Programmer's Guide

Borland InterBase Workgroup Server

Version 4.0

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R894
9495969798-9 8 7 6 5 4 3 2 1
I1

Table of Contents

Tables and Figures

Preface

This preface describes the documentation set, the printing conventions used to display information in text and in code examples, and the conventions a user should employ when specifying database objects and files by name in applications.

The InterBase Documentation Set

The InterBase documentation set is an integrated package designed for all levels of users. The InterBase server documentation consists of a five-book core set and a platform-specific installation guide. Information on the InterBase Client for Windows is provided in a single book.

The InterBase core documentation set consists of the following books:

Table 1: InterBase Core Documentation

Book	Description
Getting Started	Provides a basic introduction to InterBase and roadmap for using the documentation and a tutorial for learning basic SQL through **isql**. Introduces more advanced topics such as creating stored procedures and triggers.
Data Definition Guide	Explains how to create, alter, and delete database objects through **isql**.
Language Reference	Describes SQL and DSQL syntax and usage.
Programmer's Guide	Describes how to write embedded SQL and DSQL database applications in a host language, precompiled through **gpre**.
API Guide	Explains how to write database applications using the InterBase API.
Installing and Running on . . .	Platform-specific information on installing and running InterBase.

Additional documentation includes the following book:

Table 2: InterBase Client Documentation

Book	Description
InterBase Windows Client User's Guide	Installing and using the InterBase PC client. Using Windows **isql** and the InterBase Server Manager.

Printing Conventions

The InterBase documentation set uses different fonts to distinguish various kinds of text and syntax.

Text Conventions

The following table describes font conventions used in text, and provides examples of their use:

Table 3: Text Conventions

Convention	Purpose	Example
UPPERCASE	SQL keywords, names of all database objects such as tables, columns, indexes, stored procedures, and SQL functions.	The following SELECT statement retrieves data from the CITY column in the CITIES table.
italic	Introduces new terms, and emphasizes words. Also used for file names and host-language variables.	The *isc4.gdb* security database is *not* accessible without a valid *username* and *password*.
bold	Utility names, user-defined and host-language function names. Function names are always followed by parentheses to distinguish them from utility names.	To back up and restore a database, use **gbak** or the server manager. The **datediff()** function can be used to calculate the number of days between two dates.

Syntax Conventions

The following table describes the conventions used in syntax statements and sample code, and offers examples of their use:

Table 4: Syntax Conventions

Convention	Purpose	Example
UPPERCASE	Keywords that must be typed exactly as they appear when used.	SET TERM !!;
italic	Parameters that *cannot* be broken into smaller units. For example, a table name cannot be subdivided.	CREATE TABLE *name* (*<col>* [**,** *<col>* ...]);
<italic>	Parameters in angle brackets that *can* be broken into smaller syntactic units.	CREATE TABLE *name* (*<col>* [**,** *<col>* ...])**;**
	For example, column definitions (*<col>*) can be subdivided into a name, data type and constraint definition.	*<col> = name <datatype>* [CONSTRAINT *name <type>*]
[ ]	Square brackets enclose optional syntax.	*<col>* [**,** *<col>* ...]
...	Closely spaced ellipses indicate that a clause within brackets can be repeated as many times as necessary.	(*<col>* [**,** *<col>* ...]);
|	The pipe symbol indicates that either of two syntax clauses that it separates may be used, but not both.	SET TRANSACTION {SNAPSHOT [TABLE STABILITY] | READ COMMITTED};
	Inside curly braces, the pipe symbol separates multiple choices, one of which *must* be used.	
{ }	Curly braces indicate that one of the enclosed options *must* be included in actual statement use.	SET TRANSACTION {SNAPSHOT [TABLE STABILITY] | READ COMMITTED};

Database Object-naming Conventions

InterBase database objects, such as tables, views, and column names, appear in text and code in uppercase in the InterBase documentation set because this is the way such information is stored in a database's system tables.

When an applications programmer or end user creates a database object or refers to it by name, case is unimportant. The following limitations on naming database objects must be observed:

- Start each name with an alphabetic character (A-Z or a-z).
- Restrict object names to 31 characters, including dollar signs ($), underscores (_), 0 to 9, A to Z, and a to z. Some objects, such as constraint names, are restricted to 27 bytes in length.
- Keep object names unique. In all cases, objects of the same type, for example, tables and views, *must* be unique. In most cases, object names must also be unique within the database.

For more information about naming database objects with CREATE or DECLARE statements, see the *Language Reference*.

File-naming Conventions

InterBase is available on a wide variety of platforms. In most cases users in a heterogenous networking environment can access their InterBase database files regardless of platform differences between client and server machines if they know the target platform's file naming conventions.

Because file-naming conventions differ widely from platform to platform, and because the core InterBase documentation set is the same for each of these platforms, all file names in text and in examples are restricted to a base name with a maximum of eight characters, with a maximum extension length of three characters. For example, the example database on all servers is referred to as *employee.gdb*.

Generally, InterBase fully supports each platform's file-naming conventions, including the use of node and path names. InterBase, however, recognizes two categories of file specification in commands and statements that accept more than one file name. The first file specification is called the *primary file specification*. Subsequent file specifications are called *secondary file specifications*. Some commands and statements place restrictions on using node names with secondary file specifications.

In syntax, file specification is denoted as follows:

```
"<filespec>"
```

Primary File Specifications

InterBase syntax always supports a full file specification, including optional node name and full path, for primary file specifications. For example, the syntax notation for CREATE DATABASE appears as follows:

```
CREATE {DATABASE | SCHEMA} "<filespec>"
   [USER "username" [PASSWORD "password"]]
   [PAGE_SIZE [=] int]
   [LENGTH [=] int [PAGE[S]]]
   [DEFAULT CHARACTER SET charset]
   . . .
```

In this syntax, the <*filespec*> that follows CREATE DATABASE supports a node name and path specification, including a platform-specific drive or volume specification.

Secondary File Specifications

For InterBase syntax that supports multiple file specification, such as CREATE DATABASE, all file specifications after the first are secondary. Secondary file specifications generally cannot include a node name, but may specify a full path name. For example, the syntax notation for CREATE DATABASE appears as follows:

```
CREATE {DATABASE | SCHEMA} "<filespec>"
   [USER "username" [PASSWORD "password"]]
   [PAGE_SIZE [=] int]
   [LENGTH [=] int [PAGE[S]]]
   [DEFAULT CHARACTER SET charset]
   [<secondary_file>]

<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]

<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
   [<fileinfo>]
```

In the secondary file specification, <*filespec*> does not support specification of a node name.

CHAPTER 1

Introduction

The InterBase *Programmer's Guide* is a task-oriented explanation of how to write, preprocess, compile, and link embedded SQL and DSQL database applications using InterBase and a *host programming language*, either C or C++. This chapter describes who should read this book, and provides a brief overview of its chapters.

Who Should Use This Guide

The InterBase *Programmer's Guide* is intended for database applications programmers. It assumes a general knowledge of:

- SQL.
- Relational database programming.
- C programming.

The *Programmer's Guide* assumes little or no previous experience with InterBase.

For an introduction to InterBase and SQL, see *Getting Started*.

Topics Covered in This Guide

The following table lists the task-oriented chapters in the *Programmer's Guide*, and provides a brief description of them:

Table 1-1: *Programmer's Guide* Chapters

Chapter	Description
1: Introduction	Introduces the structure of the book and describes its intended audience.

Table 1-1: *Programmer's Guide* Chapters (Continued)

Chapter	Description
2: Application Requirements	Describes elements common to programming all SQL and DSQL applications.
3: Working With Databases	Describes using SQL statements that deal with databases.
4: Working With Transactions	Explains how to use and control transactions with SQL statements.
5: Working With Data Definition Statements	Describes how to embed SQL data definition statements in applications.
6: Working With Data	Explains how to select, insert, update, and delete standard SQL data in applications.
7: Working With Dates	Describes how to select, insert, update, and delete DATE data in applications.
8: Working With BLOB Data	Describes how to select, insert, update, and delete BLOB data in applications.
9: Using Arrays	Describes how to select, insert, update, and delete array data in applications.
10: Working With Security	Explains how to grant and revoke table and procedure privileges in applications.
11: Working With User-defined Functions	Describes how to write UDFs, how to call UDFs in applications, how to write BLOB filters, and how to create BLOB filter libraries.
12: Working With Stored Procedures	Explains how to call stored procedures in applications.
13: Working With Events	Explains how triggers interact with applications. Describes how to register interest in events, wait on them, and respond to them in applications.
14: Error Handling and Recovery	Describes how to trap and handle SQL statement errors in applications.
15: Using Dynamic SQL	Describes how to write DSQL applications.
16: Preprocessing, Compiling, and Linking	Describes how to convert source code into an executable application.

Sample Database and Applications

A sample database and sample application source code can be found in the InterBase *examples* subdirectory. The *Programmer's Guide* makes use of the sample database and source code for its examples wherever possible.

CHAPTER 2

Application Requirements

This chapter describes programming requirements for InterBase SQL and dynamic SQL (DSQL) applications. Many of these requirements may also affect developers moving existing applications to InterBase.

Requirements for All Applications

All embedded applications must include certain declarations and statements to ensure proper handling by the InterBase preprocessor, **gpre**, and to enable communication between SQL and the host language in which the application is written. Every application must:

- Declare host variables to use for data transfer between SQL and the application.
- Declare and set the databases accessed by the program.
- Create transaction handles for each non-default transaction used in the program.
- Include SQL (and, optionally, DSQL) statements.
- Provide error handling and recovery.
- Close all transactions and databases before ending the program.

Dynamic SQL applications, those applications that build SQL statements at run time, or enable users to build them, have additional requirements. For more information about DSQL requirements, see “DSQL Requirements,” in this chapter.

For more information about using **gpre**, see Chapter 16: “Preprocessing, Compiling, and Linking.”

Porting Considerations for SQL

When porting existing SQL applications to InterBase, other considerations may be necessary. For example, many SQL variants require that host variables be declared between BEGIN DECLARE SECTION and END DECLARE SECTION statements; InterBase has no such requirements, but **gpre** can correctly handle section declarations from ported applications. For additional portability, declare all host-language variables within sections.

Porting Considerations for DSQL

When porting existing DSQL applications to InterBase, statements that use another vendor's SQL descriptor area (SQLDA) must be modified to accommodate the extended SQLDA (XSQLDA) used by InterBase.

Declaring Host Variables

A *host variable* is a standard host-language variable used to hold values read from a database, to assemble values to write to a database, or to store values describing database search conditions. SQL uses host variables in the following situations:

- During data retrieval, SQL moves the values in database fields into host variables where they can be viewed and manipulated.
- When a user is prompted for information, host variables are used to hold the data until it can be passed to InterBase in an SQL INSERT or UPDATE statement.
- When specifying search conditions in a SELECT statement, conditions can be entered directly, or in a host variable. For example, both of the following SQL statement fragments are valid WHERE clauses. The second uses a host-language variable, *country*, for comparison with a column, COUNTRY:

```
. . . WHERE COUNTRY = "Mexico";

. . . WHERE COUNTRY = :country;
```

One host variable must be declared for every column of data accessed in a database. Host variables may either be declared globally like any other standard host-language variable, or may appear within an SQL section declaration with other global declarations. For more information about reading from and writing to host variables in SQL programs, see Chapter 6: "Working With Data."

Host variables used in SQL programs are declared just like standard language variables. They follow all standard host-language rules for declaration, initialization, and manipulation. For example, in C, variables must be declared before they can be used as host variables in SQL statements:

```
int empno; char fname[26], lname[26];
```

For compatibility with other SQL variants, host variables can also be declared between BEGIN DECLARE SECTION and END DECLARE SECTION statements.

Section Declarations

Many SQL implementations expect host variables to be declared between BEGIN DECLARE SECTION and END DECLARE SECTION statements. For portability and compatibility, InterBase supports section declarations using the following syntax:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      <hostvar>;
      . . .
EXEC SQL
   END DECLARE SECTION;
```

For example, the following C code fragment declares three host variables, *empno*, *fname*, and *lname*, within a section declaration:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      int empno;
      char fname[26];
      char lname[26];
EXEC SQL
   END DECLARE SECTION;
```

Additional host-language variables not used in SQL statements can be declared outside DECLARE SECTION statements.

Using BASED ON to Declare Variables

InterBase supports a declarative clause, BASED ON, for creating C language character variables based on column definitions in a database. Using BASED ON ensures that the resulting host-language variable is large enough to hold the maximum number of characters in a CHAR or VARCHAR database column, plus an extra byte for the null-terminating character expected by most C string functions.

BASED ON uses the following syntax:

```
BASED ON <dbcolumn> hostvar;
```

For example, the following statements declare two host variables, *fname*, and *lname*, based on two column definitions, FIRSTNAME, and LASTNAME, in an employee database:

```
BASED ON EMP.FIRSTNAME fname;
BASED ON EMP.LASTNAME lname;
```

Embedded in a C or C++ program, these statements generate the following host-variable declarations during preprocessing:

```
char fname[26];
char lname[26];
```

To use BASED ON, follow these steps:

1. Use SET DATABASE to specify the database from which column definitions are to be drawn.
2. Use CONNECT to attach to the database.
3. Declare a section with BEGIN DECLARE SECTION.
4. Use the BASED ON statement to declare a string variable of the appropriate type.

The following statements show the previous BASED ON declarations in context:

```
EXEC SQL
   SET DATABASE EMP = "employee.gdb";
EXEC SQL
   CONNECT EMP;
EXEC SQL
   BEGIN DECLARE SECTION;
      int empno;
      BASED ON EMP.FIRSTNAME fname;
      BASED ON EMP.LASTNAME lname;
EXEC SQL
   END DECLARE SECTION;
```

Host Variables and Host-language Data Structures

If a host language supports data structures, data fields within a structure can correspond to a collection of database columns. For example, the following C declaration creates a structure, BILLING_ADDRESS, that contains six variables, or *data members*, each of which corresponds to a similarly named column in a table:

```
struct
{
   char fname[25];
   char lname[25];
   char street[30];
   char city[20];
   char state[3];
   char zip[11];
} billing_address;
```

SQL recognizes data members in structures, but information read from or written to a structure must be read from or written to individual data members in SQL statements. For example, the following SQL statement reads data from a table into variables in the C structure, BILLING_ADDRESS:

```
EXEC SQL
   SELECT FNAME, LNAME, STREET, CITY, STATE, ZIP
      INTO :billing_address.fname, :billing_address.lname,
      :billing_address.street, :billing_address.city,
      :billing_address.state, :billing_address.zip
      FROM ADDRESSES WHERE CITY = "Brighton";
```

Declaring and Initializing Databases

An SQL program can access multiple InterBase databases at the same time. Each database used in a multiple-database program must be declared and initialized before it can be accessed in SQL transactions. Programs that access only a single database need not declare the database or assign a database handle if, instead, they specify a database on the **gpre** command line.

Important DSQL programs cannot connect to multiple databases.

InterBase supports the following SQL statements for handling databases:

- SET DATABASE declares the name of a database to access, and assigns it to a database handle.
- CONNECT opens a database specified by a handle, and allocates it system resources.

Database handles replace database names in CONNECT statements. They can also be used to qualify table names within transactions. For a complete discussion of database handling in SQL programs, see Chapter 3: “Working With Databases.”

Using SET DATABASE

The SET DATABASE statement is used to:

- Declare a database handle for each database used in an SQL program.
- Associate a database handle with an actual database name. Typically, a database handle is a mnemonic abbreviation of the actual database name.

SET DATABASE instantiates a host variable for the database handle without requiring an explicit host variable declaration. The database handle contains a pointer used to reference the database in subsequent SQL statements. To include a SET DATABASE statement in a program, use the following syntax:

```
EXEC SQL
   SET DATABASE handle = "<dbname>";
```

A separate statement should be used for each database. For example, the following statements declare a handle, DB1, for the *employee.gdb* database, and another handle, DB2, for *employee2.gdb*:

```
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
EXEC SQL
   SET DATABASE DB2 = "employee2.gdb";
```

Once a database handle is created and associated with a database, the handle can be used in subsequent SQL database and transaction statements that require it, such as CONNECT.

Note SET DATABASE also supports user name and password options. For a complete discussion of SET DATABASE options, see Chapter 3: “Working With Databases.”

Using CONNECT

The CONNECT statement attaches to a database, opens the database, and allocates system resources for it. A database must be opened before its tables can be used. To include CONNECT in a program, use the following syntax:

```
EXEC SQL
   CONNECT handle;
```

A separate statement can be used for each database, or a single statement can connect to multiple databases. For example, the following statements connect to two databases:

```
EXEC SQL
   CONNECT DB1;
EXEC SQL
   CONNECT DB2;
```

The next example uses a single CONNECT to establish both connections:

```
EXEC SQL
   CONNECT DB1, DB2;
```

Once a database is connected, its tables can be accessed in subsequent transactions. Its handle can qualify table names in SQL applications, but not in DSQL applications. For a complete discussion of CONNECT options and using database handles, see Chapter 3: “Working With Databases.”

Working With a Single Database

In single-database programs preprocessed without the **gpre -m** switch, SET DATABASE and CONNECT are optional. The **-m** switch suppresses automatic generation of transactions. Using SET DATABASE and CONNECT is strongly recommended, however, especially as a way to make program code as self-documenting as possible. If you omit these statements, take the following steps:

1. Insert a section declaration in the program code where global variables are defined. Use an empty section declaration if no host-language variables are used in the program. For example, the following declaration illustrates an empty section declaration:

   ```
   EXEC SQL
      BEGIN DECLARE SECTION;
   EXEC SQL
      END DECLARE SECTION;
   ```

2. Specify a database name on the **gpre** command line at precompile time. A database need not be specified if a program contains a CREATE DATABASE statement.

For more information about working with a single database in an SQL program, see Chapter 3: “Working With Databases.”

SQL Statements

An SQL application consists of a program written in a host language, like C or C++, into which SQL and dynamic SQL (DSQL) statements are embedded. Any SQL or DSQL statement supported by InterBase can be embedded in a host language. Each SQL or DSQL statement must be:

- Preceded by the keywords EXEC SQL.
- Ended with the statement terminator expected by the host language. For example, in C and C++, the host terminator is the semicolon (;).

For a complete list of SQL and DSQL statements supported by InterBase, see the *Language Reference*.

Error Handling and Recovery

Every time an SQL statement is executed, it returns an error code in the SQLCODE variable. SQLCODE is declared automatically for SQL programs during preprocessing with **gpre**. To catch run-time errors and recover from them when possible, SQLCODE should be examined after each SQL operation.

SQL provides the WHENEVER statement to monitor SQLCODE and direct program flow to recovery procedures. Alternatively, SQLCODE can be tested directly after each SQL statement executes. For a complete discussion of SQL error handling and recovery, see Chapter 14: “Error Handling and Recovery.”

Closing Transactions

Every transaction should be closed when it completes its tasks, or when an error occurs that prevents it from completing its tasks. Failure to close a transaction before a program ends can cause *limbo transactions*, where records are entered into the database, but are neither committed or rolled back. Limbo transactions can be cleaned up using the database administration tools provided with InterBase.

Accepting Changes

The COMMIT statement ends a transaction, makes the transaction's changes available to other users, and closes cursors. A COMMIT is used to preserve changes when all of a transaction's operations are successful. To end a transaction with COMMIT, use the following syntax:

```
EXEC SQL
   COMMIT TRANSACTION name;
```

For example, the following statement commits a transaction named MYTRANS:

```
EXEC SQL
   COMMIT TRANSACTION MYTRANS;
```

For a complete discussion of SQL transaction control, see Chapter 4: "Working With Transactions."

Undoing Changes

The ROLLBACK statement undoes a transaction's changes, ends the current transaction, and closes open cursors. Use ROLLBACK when an error occurs that prevents all of a transaction's operations from being successful. To end a transaction with ROLLBACK, use the following syntax:

```
EXEC SQL
   ROLLBACK TRANSACTION name;
```

For example, the following statement rolls back a transaction named MYTRANS:

```
EXEC SQL
   ROLLBACK TRANSACTION MYTRANS;
```

To roll back an unnamed transaction (i.e., the default transaction), use the following statement:

```
EXEC SQL
   ROLLBACK;
```

For a complete discussion of SQL transaction control, see Chapter 4: "Working With Transactions."

Closing Databases

Once a database is no longer needed, close it before the program ends, or subsequent attempts to use the database may fail or result in database corruption. There are two ways to close a database:

- Use the DISCONNECT statement to detach a database and close files.
- Use the RELEASE option with COMMIT or ROLLBACK in a program.

DISCONNECT, COMMIT RELEASE, and ROLLBACK RELEASE perform the following tasks:

- Close open database files.
- Close remote database connections.
- Release the memory that holds database descriptions and InterBase engine-compiled requests.

Note Closing databases with DISCONNECT is preferred for compatibility with the SQL-92 standard.

For a complete discussion of closing databases, see Chapter 3: “Working With Databases.”

DSQL Requirements

DSQL applications must adhere to all the requirements for all SQL applications and meet additional requirements as well. DSQL applications enable users to enter ad hoc SQL statements for processing at run time. To handle the wide variety of statements a user might enter, DSQL applications require the following additional programming steps:

- Declare as many extended SQL descriptor areas (XSQLDAs) as are needed in the application; typically a program must use one or two of these structures. Complex applications may require more.
- Declare all transaction names and database handles used in the program at compile time; names and handles are not dynamic, so enough must be declared to accommodate the anticipated needs of users at run time.
- Provide a mechanism to get SQL statements from a user.
- Prepare each SQL statement received from a user for processing. PREPARE loads statement information into the XSQLDA.

- EXECUTE each prepared statement.

EXECUTE IMMEDIATE combines PREPARE and EXECUTE in a single statement. For more information, see the *Language Reference*.

In addition, the syntax for cursors involving BLOB data differs from that of cursors for other data types. For more information about BLOB cursor statements, see the *Language Reference*.

Declaring an XSQLDA

The *extended SQL descriptor area* (XSQLDA) is used as an intermediate staging area for information passed between an application and the InterBase engine. The XSQLDA is used for either of the following tasks:

- Pass input parameters from a host-language program to SQL.
- Pass output, from a SELECT statement or stored procedure, from SQL to the host-language program.

A single XSQLDA can be used for only one of these tasks at a time. Many applications declare two XSQLDAs, one for input, and another for output.

The XSQLDA structure is defined in the InterBase header file, *ibase.h*, that is automatically included in programs when they are preprocessed with **gpre**.

Note DSQL applications written using versions of InterBase prior to 3.3 use an older SQL descriptor area, the SQLDA. For backward compatibility, the SQLDA continues to be supported. You can examine its structure in *ibase.h*. The new structure, XSQLDA, is used automatically when preprocessing an application with **gpre**. To use the old structure, specify the **gpre -sqlda old** switch. As convenient, older applications should be modified to use the XSQLDA.

To create an XSQLDA for a program, a host-language data type of the appropriate type must be set up in a section declaration. For example, the following statement creates two XSQLDA structures, *inxsqlda*, and *outxsqlda*:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      XSQLDA inxsqlda;
      XSQLDA outxsqlda;
      . . .
EXEC SQL
   END DECLARE SECTION;
. . .
```

When an application containing XSQLDA declarations is preprocessed, **gpre** automatically includes the header file, *ibase.h*, which defines the XSQLDA as a host-language data type. For a complete discussion of the structure of the XSQLDA, see Chapter 15: “Using Dynamic SQL.”

DSQL Limitations

DSQL enables programmers to create flexible applications that are capable of handling a wide variety of user requests. Even so, not every SQL statement can be handled in a completely dynamic fashion. For example, database handles and transaction names must be specified when an application is written, and cannot be changed or specified by users at run time. Similarly, while InterBase supports multiple databases and multiple simultaneous transactions in an application, the following limitations apply:

- Only a single database can be accessed at a time.
- Transactions can only operate on the currently active database.
- Users cannot specify transaction names in DSQL statements; instead, transaction names must be supplied and manipulated when an application is coded.

Using Database Handles

Database handles are always static, and can only be declared when an application is coded. Enough handles must be declared to satisfy the expected needs of users. Once a handle is declared, it can be assigned to a user-specified database at run time with SET DATABASE, as in the following C code fragment:

```
. . .
EXEC SQL
   SET DATABASE DB1 = "dummydb.gdb";
EXEC SQL
   SET DATABASE DB2 = "dummydb.gdb";
. . .
printf("Specify first database to open: ");
gets(fname1);
printf("\nSpecify second database to open: ");
gets(fname2);
EXEC SQL
   SET DATABASE DB1 = :fname1;
EXEC SQL
   SET DATABASE DB2 = :fname2;
. . .
```

For a complete discussion of SET DATABASE, see Chapter 3: “Working With Databases.”

Using the Active Database

A DSQL application can only work with one database at a time, even if the application attaches to multiple databases. All DSQL statements operate only on the currently *active database*, the last database associated with a handle in a SET DATABASE statement.

Embedded SQL statements within a DSQL application can operate on any open database. For example, all DSQL statements entered by a user at run time might operate against a single database specified by the user, but the application might also contain non-DSQL statements that record user entries in a log database.

For a complete discussion of SET DATABASE, see Chapter 3: “Working With Databases.”

Using Transaction Names

Many SQL statements support an optional transaction name parameter, used to specify the controlling transaction for a specific statement. Transaction names can be used in DSQL applications, too, but must be set up when an application is compiled. Once a name is declared, it can be directly inserted into a user statement only by the application itself.

After declaration, use a transaction name in an EXECUTE or EXECUTE IMMEDIATE statement to specify the controlling transaction, as in the following C code fragment:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION:
      long first, second; /* declare transaction names */
EXEC SQL
   END DECLARE SECTION;
. . .
first = second = 0L; /* initialize names to zero */
. . .
EXEC SQL
   SET TRANSACTION first; /* start transaction 1 */
EXEC SQL
   SET TRANSACTION second; /* start transaction 2 */
printf("\nSQL> ");
gets(userstatement);
EXEC SQL
   EXECUTE IMMEDIATE TRANSACTION first userstatement;
```

. . .

For complete information about named transactions, see Chapter 4: “Working With Transactions.”

Preprocessing Programs

After an SQL or DSQL program is written, and before it is compiled and linked, it must be preprocessed with **gpre**, the InterBase preprocessor. **gpre** translates SQL statements and variables into statements and variables that the host-language compiler accepts. For complete information about preprocessing with **gpre**, see Chapter 16: “Preprocessing, Compiling, and Linking.”

CHAPTER 3

Working With Databases

This chapter describes how to use SQL statements in embedded applications to control databases. There are three database statements that set up and open databases for access:

- SET DATABASE declares a database handle, associates the handle with an actual database file, and optionally assigns operational parameters for the database.
- SET NAMES optionally specifies the character set a client application uses for CHAR, VARCHAR, and text BLOB data. The server uses this information to transliterate from a database's default character set to the client's character set on SELECT operations, and to transliterate from a client application's character set to the database character set on INSERT and UPDATE operations.
- CONNECT opens a database, allocates system resources for it, and optionally assigns operational parameters for the database.

All databases must be closed before a program ends. A database can be closed by using DISCONNECT, or by appending the RELEASE option to the final COMMIT or ROLLBACK in a program.

Declaring a Database

Before a database can be opened and used in a program, it must first be declared with SET DATABASE to:

- Establish a database handle.
- Associate the database handle with a database file stored on a local or remote node.

A *database handle* is a unique, abbreviated alias for an actual database name. Database handles are used in subsequent CONNECT, COMMIT RELEASE, and ROLLBACK RELEASE statements to specify which databases they should affect.

Except in dynamic SQL (DSQL) applications, database handles can also be used inside transaction blocks to *qualify*, or differentiate, table names when two or more open databases contain identically named tables.

Each database handle must be unique among all variables used in a program. Database handles cannot duplicate host-language reserved words, and cannot be InterBase reserved words.

The following statement illustrates a simple database declaration:

```
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
```

This database declaration identifies the database file, *employee.gdb*, as a database the program uses, and assigns the database a handle, or alias, DB1.

If a program runs in a directory different from the directory that contains the database file, then the file name specification in SET DATABASE *must* include a full path name, too. For example, the following SET DATABASE declaration specifies the full path to *employee.gdb*:

```
EXEC SQL
   SET DATABASE DB1 = "/usr/interbase/examples/employee.gdb";
```

If a program and a database file it uses reside on different hosts, then the file name specification must also include a host name. The following declaration illustrates how a Unix host name is included as part of the database file specification:

```
EXEC SQL
   SET DATABASE DB1 = "vega:usr/interbase/examples/employee.gdb";
```

Note Host syntax is specific to each server platform on which InterBase runs. For the correct host syntax for a particular server, see the server's documentation.

Declaring Multiple Databases

An SQL program, but not a DSQL program, can access multiple databases at the same time. In multi-database programs, database handles are required. A handle is used to:

- Reference individual databases in a multi-database transaction.
- Qualify table names.
- Specify databases to open in CONNECT statements.

- Indicate databases to close with DISCONNECT, COMMIT RELEASE, and ROLLBACK RELEASE.

DSQL programs can access only a single database at a time, so database handle use is restricted to connecting to and disconnecting from a database.

In multi-database programs, each database must be declared in a separate SET DATABASE statement. For example, the following code contains two SET DATABASE statements:

```
. . .
EXEC SQL
   SET DATABASE DB2 = "employee2.gdb";
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
. . .
```

Using Handles to Differentiate Table Names

When the same table name occurs in more than one simultaneously accessed database, a database handle must be used to differentiate one table name from another. The database handle is used as a prefix to table names, and takes the form <*handle*>.<*table*>.

For example, in the following code, the database handles, TEST and EMP, are used to distinguish between two tables, each named EMPLOYEE:

```
. . .
EXEC SQL
   DECLARE IDMATCH CURSOR FOR
      SELECT TESTNO INTO :matchid FROM TEST.EMPLOYEE
         WHERE TESTNO > 100;
EXEC SQL
   DECLARE EIDMATCH CURSOR FOR
      SELECT EMPNO INTO :empid FROM EMP.EMPLOYEE
      WHERE EMPNO = :matchid;
. . .
```

Important This use of database handles applies only to embedded SQL applications. DSQL applications cannot access multiple databases simultaneously.

Using Handles With CONNECT, DISCONNECT, COMMIT, and ROLLBACK

In multi-database programs, database handles *must* be specified in CONNECT statements to identify which databases among several to open and prepare for use in subsequent transactions.

Database handles can also be used with DISCONNECT, COMMIT RELEASE, and ROLLBACK RELEASE to specify a subset of open databases to close.

To open and prepare a database with CONNECT, see "Opening a Database," in this chapter. To close a database with DISCONNECT, COMMIT RELEASE, or ROLLBACK RELEASE, see "Closing a Database," in this chapter. To learn more about using database handles in transactions, see "Accessing an Open Database," in this chapter.

Using Different Databases for Preprocessing and Run Time

Normally, each SET DATABASE statement specifies a single database file to associate with a handle. When a program is preprocessed, **gpre** uses the specified file to validate the program's table and column references. Later, when a user runs the program, the same database file is accessed. Different databases can be specified for preprocessing and run time when necessary.

Using the COMPILETIME Clause

Sometimes a program may be designed to run against any one of several identically structured databases, or the actual database that a program will use at run time is not available when a program is preprocessed and compiled. In these cases, SET DATABASE can include a COMPILETIME clause to specify a database for **gpre** to test against during preprocessing. For example, the following SET DATABASE statement declares that *employee.gdb* is to be used by **gpre** during preprocessing:

```
EXEC SQL
   SET DATABASE EMP = COMPILETIME "employee.gdb";
```

Important The file specification that follows the COMPILETIME keyword must always be a hard-coded, quoted string.

When SET DATABASE uses the COMPILETIME clause, but no RUNTIME clause, and does not specify a different database file specification in a subsequent CONNECT statement, the same database file is used both for preprocessing and run time. To specify different preprocessing and run-time databases with SET DATABASE, use both the COMPILETIME and RUNTIME clauses.

Using the RUNTIME Clause

When a database file is specified for use during preprocessing, SET DATABASE can specify a different database to use at run time by including the RUNTIME keyword and a run-time file specification:

```
EXEC SQL
   SET DATABASE EMP = COMPILETIME "employee.gdb"
      RUNTIME "employee2.gdb";
```

The file specification that follows the RUNTIME keyword can be either a hard-coded, quoted string, or a host-language variable. For example, the following C code fragment prompts the user for a database name, and stores the name in a variable that is used later in SET DATABASE:

```
. . .
char db_name[125];
. . .
printf("Enter the desired database name, including node and path):\n");
gets(db_name);
EXEC SQL
   SET DATABASE EMP = COMPILETIME "employee.gdb" RUNTIME :db_name;
. . .
```

Note Host-language variables in SET DATABASE must be preceded, as always, by a colon.

Controlling SET DATABASE Scope

By default, SET DATABASE creates a handle that is global to all modules in an application. A *global handle* is one that may be referenced in all host-language modules comprising the program. SET DATABASE provides two optional keywords to change the scope of a declaration:

- STATIC limits declaration scope to the module containing the SET DATABASE statement. No other program modules can see or use a database handle declared STATIC.
- EXTERN notifies **gpre** that a SET DATABASE statement in a module duplicates a globally-declared database in another module. If the EXTERN keyword is used, then another module must contain the actual SET DATABASE statement, or an error occurs during compilation.

The STATIC keyword is used in a multi-module program to restrict database handle access to the single module where it is declared. The following example illustrates the use of the STATIC keyword:

```
EXEC SQL
   SET DATABASE EMP = STATIC "employee.gdb";
```

The EXTERN keyword is used in a multi-module program to signal that SET DATABASE in one module is not an actual declaration, but refers to a declaration made in a different module. **gpre** uses this information during preprocessing. The following example illustrates the use of the EXTERN keyword:

```
EXEC SQL
   SET DATABASE EMP = EXTERN "employee.gdb";
```

If an application contains an EXTERN reference, then when it is used at run time, the actual SET DATABASE declaration must be processed first, and the database connected before other modules can access it.

A single SET DATABASE statement can contain either the STATIC or EXTERN keyword, but not both. A scope declaration in SET DATABASE applies to both COMPILETIME and RUNTIME databases.

Specifying a Character Set for a Client Connection

When a client application connects to a database, it may have its own character set requirements. The server providing database access to the client does not know about these requirements unless the client specifies them. The client application specifies its character set requirement using the SET NAMES statement *before* it connects to the database.

SET NAMES specifies the character set the server should use when translating data from the database to the client application. Similarly, when the client sends data to the database, the server translates the data from the client's character set to the database's default character set (or the character set for an individual column if it differs from the database's default character set).

For example, the following statements specify that the client is using the DOS437 character set, then connect to the database:

```
EXEC SQL
   SET NAMES DOS437;
EXEC SQL
   CONNECT "europe.gdb" USER "JAMES" PASSWORD "U4EEAH";
```

For more information about character sets, see the *Data Definition Guide*. For the complete syntax of SET NAMES and CONNECT, see the *Language Reference*.

Opening a Database

After a database is declared, it must be attached with a CONNECT statement before it can be used. CONNECT:

- Allocates system resources for the database.

- Determines if the database file is *local*, residing on the same host where the application itself is running, or *remote*, residing on a different host.
- Opens the database and examines it to make sure it is valid.

InterBase provides transparent access to all databases, whether local or remote. If the database structure is invalid, the on-disk structure (ODS) number does not correspond to the one required by InterBase, or if the database is corrupt, InterBase reports an error, and permits no further access.

Optionally, CONNECT can be used to specify:

- A user name and password combination that is checked against the server's security database before allowing the connect to succeed. User names can be up to 31 characters. Passwords are restricted to 8 characters.
- The size of the database buffer cache to allocate to the application when the default cache size is inappropriate.

Using Simple CONNECT Statements

In its simplest form, CONNECT requires one or more database parameters, each specifying the name of a database to open. The name of the database can be a:

- Database handle declared in a previous SET DATABASE statement.
- Host-language variable.
- Hard-coded file name.

Using a Database Handle

If a program uses SET DATABASE to provide database handles, those handles should be used in subsequent CONNECT statements instead of hard-coded names. For example,

```
. . .
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
EXEC SQL
   SET DATABASE DB2 = "employee2.gdb";
EXEC SQL
   CONNECT DB1;
EXEC SQL
   CONNECT DB2;
. . .
```

There are several advantages to using a database handle with CONNECT:

- Long file specifications can be replaced by shorter, mnemonic handles.
- Handles can be used to qualify table names in multi-database transactions. DSQL applications do not support multi-database transactions.
- Handles can be reassigned to other databases as needed.
- The number of database cache buffers can be specified as an additional CONNECT parameter.

For more information about setting the number of database cache buffers, see "Setting Database Cache Buffers," in this chapter.

Using Host-language Variables or Hard-coded Strings

Instead of using a database handle, CONNECT can use a database name supplied at run time. The database name can be supplied as either a host-language variable or a hard-coded, quoted string.

The following C code demonstrates how a program accessing only a single database might implement CONNECT using a file name solicited from a user at run time:

```
. . .
char fname[125];
. . .
printf("Enter the desired database name, including node and path):\n");
gets(fname);
. . .
EXEC SQL
   CONNECT :fname;
. . .
```

Tip This technique is especially useful for programs that are designed to work with many identically structured databases, one at a time, such as CAD/CAM or architectural databases.

Multiple Database Implementation

To use a database specified by the user as a host-language variable in a CONNECT statement in multi-database programs, follow these steps:

1. Declare a database handle using the following SET DATABASE syntax:

```
EXEC SQL
   SET DATABASE handle = COMPILETIME "dbname";
```

Here, *handle* is a hard-coded database handle supplied by the programmer, *dbname* is a quoted, hard-coded database name used by **gpre** during preprocessing.

2. Prompt the user for a database to open.
3. Store the database name entered by the user in a host-language variable.
4. Use the handle to open the database, associating the host-language variable with the handle using the following CONNECT syntax:

```
EXEC SQL
   CONNECT :variable AS handle;
```

The following C code illustrates these steps:

```
. . .
char fname[125];
. . .
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
printf("Enter the desired database name, including node and path):\n");
gets(fname);
EXEC SQL
   CONNECT :fname AS DB1;
. . .
```

In this example, SET DATABASE provides a hard-coded database file name for preprocessing with **gpre**. When a user runs the program, the database specified in the variable, *fname*, is used instead.

Using a Hard-coded Database Name in a Single-database Program

In a single-database program that omits SET DATABASE, CONNECT *must* contain a hard-coded, quoted file name in the following format:

```
EXEC SQL
   CONNECT "[host[path]]filename";
```

host is only required if a program and a database file it uses reside on different nodes. Similarly, *path* is only required if the database file does not reside in the current working directory. For example, the following CONNECT statement contains a hard-coded file name that includes both a Unix host name and a path name:

```
EXEC SQL
   CONNECT "valdez:usr/interbase/examples/employee.gdb";
```

Note Host syntax is specific to each server platform on which InterBase runs. For the correct host syntax for a particular server, see the server's documentation.

Important A program that accesses multiple databases *cannot* use this form of CONNECT.

Using a Hard-coded Database Name in a Multi-database Program

A program that accesses multiple databases must declare handles for each of them in separate SET DATABASE statements. These handles must be used in subsequent CONNECT statements to identify specific databases to open:

```
. . .
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
EXEC SQL
   SET DATABASE DB2 = "employee2.gdb";
EXEC SQL
   CONNECT DB1;
EXEC SQL
   CONNECT DB2;
. . .
```

Later, when the program closes these databases, the database handles are no longer in use. These handles can be reassigned to other databases by hard-coding a file name in a subsequent CONNECT statement. For example,

```
. . .
EXEC SQL
   DISCONNECT DB1, DB2;
EXEC SQL
   CONNECT "project.gdb" AS DB1;
. . .
```

Additional CONNECT Syntax

CONNECT supports several formats for opening databases to provide programming flexibility. The following table outlines each possible syntax, provides

descriptions and examples, and indicates whether CONNECT can be used in programs that access single or multiple databases:

Table 3-1: CONNECT Syntax Summary

Syntax	Description	Example	Single Access	Multiple Access
CONNECT "*<dbfile>*";	Open a single, hard-coded database file, *<dbfile>*.	EXEC SQL CONNECT "employee.gdb";	Yes	No
CONNECT *handle*;	Open the database file associated with a previously declared database handle. This is the preferred CONNECT syntax.	EXEC SQL CONNECT EMP;	Yes	Yes
CONNECT "*<dbfile>*" AS *handle*;	Open a hard-coded database file, *<dbfile>*, and assign a previously declared database handle to it.	EXEC SQL CONNECT "employee.gdb" AS EMP;	Yes	Yes
CONNECT :*varname* AS *handle*;	Open the database file stored in the host-language variable, *<varname>*, and assign a previously declared database handle to it.	EXEC SQL CONNECT :fname AS EMP;	Yes	Yes

For a complete discussion of CONNECT syntax and its uses, see the *Language Reference*.

Attaching to Multiple Databases With a Single CONNECT

CONNECT can attach to multiple databases. To open all databases specified in previous SET DATABASE statements, use either of the following CONNECT syntax options:

```
EXEC SQL
   CONNECT ALL;

EXEC SQL
   CONNECT DEFAULT;
```

CONNECT can also attach to a specified list of databases. Separate each database request from others with commas. For example, the following statement opens two databases specified by their handles:

```
EXEC SQL
   CONNECT DB1, DB2;
```

The next statement opens two hard-coded database files and also assigns them to previously declared handles:

```
EXEC SQL
   CONNECT "employee.gdb" AS DB1, "employee2.gdb" AS DB2;
```

Tip Opening multiple databases with a single CONNECT is most effective when a program's database access is simple and clear. In complex programs that open and close several databases, that substitute database names with host-language variables, or that assign multiple handles to the same database, use separate CONNECT statements to make program code easier to read, debug, and modify.

Handling CONNECT Errors

The WHENEVER statement should be used to trap and handle run-time errors that occur during database declaration. The following C code fragment illustrates an error-handling routine that displays error messages and ends the program in an orderly fashion:

```
. . .
EXEC SQL
   WHENEVER SQLERROR
      GOTO error_exit;
. . .
:error_exit
   isc_print_sqlerr(sqlcode, status_vector);
   EXEC SQL
      DISCONNECT ALL;
   exit(1);
. . .
```

For a complete discussion of SQL error handling, see Chapter 14: "Error Handling and Recovery."

Setting Database Cache Buffers

Besides opening a database, CONNECT can set the number of *cache buffers* assigned to a database. When a program establishes a connection to a database, InterBase allocates system memory to use as a private buffer. The buffers are used to store accessed database pages to speed performance. The number of buffers assigned for a program determine how many simultaneous database pages it can have access to in the memory pool. Buffers remain assigned until a program finishes with a database.

The default number of database cache buffers assigned to a database is 75. Use the CACHE *n* parameter with CONNECT to change the number of buffers assigned to a database.

For programs that use many databases, but access or change only a few tables in them, a smaller number of buffers can be used. The minimum number of buffers allowed is 43.

For programs that access or change many rows in many databases, performance may be improved by increasing the number of buffers. The maximum number of buffers allowed is system-dependent.

Setting Buffers For Individual Databases

The CONNECT statement's optional CACHE *n* parameter sets the number of buffers for a database, where *n* is the number of buffers to reserve. To set the number of buffers for an individual database, place CACHE *n* after the database name. The following CONNECT specifies 50 buffers for the database pointed to by the EMP handle:

```
EXEC SQL
   CONNECT EMP CACHE 50;
```

The next statement opens two databases, TEST and EMP. Because CACHE is not specified for TEST, its buffers default to 75. EMP is opened with the CACHE clause specifying 100 buffers:

```
EXEC SQL
   CONNECT TEST, EMP CACHE 100;
```

Specifying Buffers for All Databases

To specify the same number of buffers for *all* databases, use CONNECT ALL with the CACHE *n* parameter. For example, the following statements connect to two databases, EMP, and EMP2, and set the number of buffers allotted to each of them to 90:

```
. . .
EXEC SQL
   SET DATABASE EMP = "employee.gdb";
EXEC SQL
   SET DATABASE EMP2 = "test.gdb";
EXEC SQL
   CONNECT ALL CACHE 90;
. . .
```

The same effect can be achieved by specifying the same amount of cache for individual databases:

```
. . .
EXEC SQL
   CONNECT EMP CACHE 90, TEST CACHE 90;
. . .
```

Accessing an Open Database

Once a database is connected, its tables can be accessed as follows:

- One database can be accessed in a single transaction.
- One database can be accessed in multiple transactions.
- Multiple databases can be accessed in a single transaction.
- Multiple databases can be accessed in multiple transactions.

For general information about using transactions, see Chapter 4: “Working With Transactions.”

Using Database Handles to Differentiate Table Names

In SQL, using multiple databases in transactions may require extra precautions to ensure intended behavior. When two or more databases have tables that share the same name, a database handle must be prefixed to those table names to differentiate them from one another in transactions.

A table name differentiated by a database handle takes the form:

```
handle.table
```

For example, the following cursor declaration accesses an EMPLOYEE table in TEST, and another EMPLOYEE table in EMP. TEST and EMP are used as prefixes to indicate which EMPLOYEE table should be referenced:

```
. . .
EXEC SQL
   DECLARE IDMATCH CURSOR FOR
      SELECT TESTNO INTO :matchid FROM TEST.EMPLOYEE
         WHERE (SELECT EMPNO FROM EMP.EMPLOYEE WHERE EMPNO = TESTNO);
. . .
```

Note DSQL does not support access to multiple databases in a single statement.

Closing a Database

When a program is finished with a database, the database should be closed. In SQL, a database can be closed in either of the following ways:

- Issue a DISCONNECT to detach a database and close files.
- Append a RELEASE option to a COMMIT or ROLLBACK to disconnect from a database and close files.

DISCONNECT, COMMIT RELEASE, and ROLLBACK RELEASE perform the following tasks:

- Close open database files.
- Disconnect from remote database connections.
- Release the memory that holds database metadata descriptions and InterBase engine-compiled requests.

Note Closing databases with DISCONNECT is preferred for compatibility with the SQL-92 standard. Do not close a database until it is no longer needed. Once closed, a database must be reopened, and its resources reallocated, before it can be used again.

Closing Databases With DISCONNECT

To close all open databases, use the following DISCONNECT syntax:

```
EXEC SQL
   DISCONNECT {ALL | DEFAULT};
```

For example, each of the following statements closes all open databases in a program:

```
EXEC SQL
   DISCONNECT ALL;

EXEC SQL
   DISCONNECT DEFAULT;
```

To close specific databases, specify their handles as comma-delimited parameters, using the following syntax:

```
EXEC SQL
   DISCONNECT handle [, handle ...];
```

For example, the following statement disconnects from two databases:

```
EXEC SQL
   DISCONNECT DB1, DB2;
```

Note A database should not be closed until all transactions are finished with it, or it must be reopened and its resources reallocated.

Closing Databases With COMMIT and ROLLBACK

To close all open databases with COMMIT or ROLLBACK use the following syntax:

```
EXEC SQL
   {COMMIT | ROLLBACK} RELEASE;
```

For example, the following COMMIT closes all open databases:

```
EXEC SQL
   COMMIT RELEASE;
```

To close specific databases, provide their handles as parameters following the RELEASE option with COMMIT or ROLLBACK, using the following syntax:

```
EXEC SQL
   COMMIT | ROLLBACK RELEASE handle [, handle ...];
```

For example, the next ROLLBACK statement closes two databases:

```
EXEC SQL
   ROLLBACK RELEASE DB1, DB2;
```

CHAPTER 4

Working With Transactions

All SQL data definition and data manipulation statements take place within the context of a *transaction*, a set of SQL statements that works to carry out a single task. This chapter explains how to open, control, and close transactions using the following SQL transaction management statements:

Table 4-1: SQL Transaction Management Statements

Statement	Purpose
SET TRANSACTION	Starts a transaction, assigns it a name, and specifies its behavior. The following behaviors can be specified: • *Access mode* describes the actions a transaction's statements can perform. • *Lock resolution* describes how a transaction should react if a lock conflict occurs. • *Isolation level* describes the view of the database given a transaction as it relates to actions performed by other simultaneously occurring transactions. • *Table reservation*, an optional list of tables to lock for access at the start of the transaction rather than at the time of explicit reads or writes. • *Database specification*, an optional list limiting the open databases to which a transaction may have access.
COMMIT	Saves a transaction's changes to the database and ends the transaction.
ROLLBACK	Undoes a transaction's changes before they have been committed to the database, and ends the transaction.

Transaction management statements define the beginning and end of a transaction. They also control its behavior and interaction with other simultaneously running transactions that share access to the same data within and across applications.

There are two types of transactions in InterBase:

- The *default transaction*, **gds__trans**, is used by InterBase if it encounters any statement that requires a transaction without first finding a SET TRANSACTION statement. A default behavior is defined for **gds__trans**, but can be changed by starting the default transaction with SET TRANSACTION and specifying alternative behavior as parameters.

Important When using the default transaction without explicitly starting it with SET TRANSACTION, applications must be preprocessed *without* the **-m gpre** switch.

- *Named transactions* are always started with SET TRANSACTION statements. These statements provide unique names for each transaction, and usually include parameters that specify a transaction's behavior.

Except for naming conventions, and use in multi-transaction programs, both the default and named transactions offer the same control over transactions. Optional parameters to SET TRANSACTION can be used to specify its behavior (access mode, lock resolution, and isolation level).

For more information about **gpre**, see Chapter 16: "Preprocessing, Compiling, and Linking." For more information about transaction behavior, see "Specifying SET TRANSACTION Behavior," in this chapter.

Starting the Default Transaction

If a transaction is started without a specified behavior, the following default behavior is used:

```
READ WRITE WAIT ISOLATION LEVEL SNAPSHOT
```

The default transaction is especially useful for programs that use only a single transaction. It is automatically started in programs that require a transaction context where none is explicitly provided. It can also be explicitly started in a program with SET TRANSACTION.

To learn more about transaction behavior, see "Starting the Default Transaction," in this chapter.

Starting the Default Transaction Without SET TRANSACTION

Simple, single transaction programs can omit SET TRANSACTION. The following program fragment issues a SELECT statement without starting a transaction:

```
. . .
EXEC SQL
```

```
    SELECT * FROM CITIES
       WHERE POPULATION > 4000000
          ORDER BY POPULATION, CITY;
. . .
```

A programmer need only start the default transaction explicitly in a single transaction program to modify its operating characteristics or when writing a DSQL application that is preprocessed with the **gpre -m** switch.

During preprocessing, when **gpre** encounters a statement, such as SELECT, that requires a transaction context without first finding a SET TRANSACTION statement, it automatically generates a default transaction as long as the **-m** switch is not specified. A default transaction started by **gpre** uses a predefined, or default, behavior that dictates how the transaction interacts with other simultaneous transactions attempting to access the same data.

Important DSQL programs should be preprocessed with the **gpre -m** switch if they start a transaction through DSQL. In this mode, **gpre** does *not* generate the default transaction as needed, but instead reports an error if there is no transaction.

For more information about transaction behaviors that can be modified, see "Specifying SET TRANSACTION Behavior," in this chapter. For more information about using the **gpre -m** switch, see Chapter 16: "Preprocessing, Compiling, and Linking."

Starting the Default Transaction With SET TRANSACTION

SET TRANSACTION issued without parameters starts the default transaction, **gds__trans**, with the following default behavior:

```
READ WRITE WAIT ISOLATION LEVEL SNAPSHOT
```

The following table summarizes these settings:

Table 4-2: Default Transaction Default Behavior

Parameter	Setting	Purpose
Access Mode	READ WRITE	Access mode. This transaction can select, insert, update, and delete data.
Lock Resolution	WAIT	Lock resolution. This transaction waits for locked tables and rows to be released to see if it can then update them before reporting a lock conflict.
Isolation Level	ISOLATION LEVEL SNAPSHOT	This transaction receives a stable, unchanging view of the database as it is at the moment the transaction starts; it never sees changes made to the database by other active transactions.

Note Explicitly starting the default transaction is good programming practice. It makes a program's source code easier to understand.

The following statements are equivalent. They both start the default transaction with the default behavior.

```
EXEC SQL
   SET TRANSACTION;

EXEC SQL
   SET TRANSACTION NAME gds__trans READ WRITE WAIT ISOLATION LEVEL
      SNAPSHOT;
```

To start the default transaction, but change its characteristics, SET TRANSACTION must be used to specify those characteristics that differ from the default. Characteristics that do not differ from the default can be omitted. For example, the following statement starts the default transaction for READ ONLY access, WAIT lock resolution, and ISOLATION LEVEL SNAPSHOT:

```
EXEC SQL
   SET TRANSACTION READ ONLY;
```

As this example illustrates, the NAME clause can be omitted when starting the default transaction.

Important In DSQL, changing the characteristics of the default transaction is accomplished as with PREPARE and EXECUTE in a manner similar to the one described, but the program must be preprocessed using the **gpre -m** switch.

For more information about preprocessing programs with the **-m** switch, see Chapter 16: "Preprocessing, Compiling, and Linking." For more information about transaction behavior and modification, see "Specifying SET TRANSACTION Behavior," in this chapter.

Starting a Named Transaction

A single application can start simultaneous transactions. InterBase extends transaction management and data manipulation statements to support *transaction names*, unique identifiers that specify which transaction controls a given statement among those transactions that are active.

Transaction names must be used to distinguish one transaction from another in programs that use two or more transactions at a time. Each transaction started while other transactions are active requires a unique name and its own SET TRANSACTION statement. SET TRANSACTION can include optional parameters that modify a transaction's behavior.

There are four steps for using transaction names in a program:

1. Declare a unique host-language variable for each transaction name. In C and C++, transaction names should be declared as long pointers.
2. Initialize each transaction name to zero.
3. Use SET TRANSACTION to start each transaction using an available transaction name.
4. Include the transaction name in subsequent transaction management and data manipulation statements that should be controlled by a specified transaction.

Important Using named transactions in dynamic SQL statements is somewhat different. For information about named transactions in DSQL, see "Working With Multiple Transactions in DSQL," in this chapter.

For additional information about creating multiple transaction programs, see "Working With Multiple Transactions," in this chapter.

Naming Transactions

A transaction name is a programmer-supplied variable that distinguishes one transaction from another in SQL statements. If transaction names are not used in SQL statements that control transactions and manipulate data, then those statements operate only on the default transaction, **gds__trans**.

The following C code declares and initializes two transaction names. It also starts those transactions in SET TRANSACTION statements.

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      long *t1, *t2; /* declare transaction names */
EXEC SQL
   END DECLARE SECTION;
. . .
t1 = t2 = 0L; /* initialize names to zero */
. . .
EXEC SQL
   SET TRANSACTION NAME t1; /* start trans. w. default behavior */
EXEC SQL
   SET TRANSACTION NAME t2; /* start trans2. w. default behavior */
. . .
```

Each of these steps is fully described in the following sections.

A transaction name can be included as an optional parameter in any data manipulation and transaction management statement. In multi-transaction

programs, omitting a transaction name causes a statement to be executed for the default transaction, **gds__trans**.

For more information about using transaction names with data manipulation statements, see Chapter 6: “Working With Data.”

Declaring Transaction Names

Transaction names must be declared before they can be used. A name is declared as a host-language pointer. In C and C++, transaction names should be declared as long pointers.

The following code illustrates how to declare two transaction names:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      long *t1;
      long *t2;
EXEC SQL
   END DECLARE SECTION;
```

Note In this example, the transaction declaration occurs within an SQL section declaration. While InterBase does not require that host-language variables occur within a section declaration, putting them there guarantees compatibility with other SQL implementations that do require section declarations.

Transaction names are usually declared globally at the module level. If a transaction name is declared locally, ensure that:

- The transaction using the name is completely contained within the function where the name is declared. Include an error-handling routine to roll back transactions when errors occur. ROLLBACK releases a transaction name, and sets its value to NULL.
- The transaction name is not used outside the function where it is declared.

To reference a transaction name declared in another module, provide an external declaration for it. For example, in C, the external declaration for *t1* and *t2* might be as follows:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      extern long *t1, *t2;
EXEC SQL
   END DECLARE SECTION;
```

Initializing Transaction Names

Once transaction names are declared, they should be initialized to zero before being used for the first time. The following C code illustrates how to set a starting value for two declared transaction names:

```
t1 = t2 = 0L; /* initialize transaction names to zero */
```

Once a transaction name is declared and initialized, it can be used to:

- Start and name a transaction. Using a transaction name for all transactions except for the default transaction is required if a program runs multiple, simultaneous transactions.
- Specify which transactions control data manipulation statements. Transaction names are required in multi-transaction programs, unless a statement affects only the default transaction.
- Commit or roll back specific transactions in a multi-transaction program.

Specifying SET TRANSACTION Behavior

Use SET TRANSACTION to start a named transaction, and optionally specify its behavior. The syntax for starting a named transaction using default behavior is:

```
SET TRANSACTION NAME name;
```

For a summary of the default behavior for a transaction started without specifying behavior parameters, see Table 4-2. The following statements are equivalent. They both start the transaction named *t1*, using default transaction behavior.

```
EXEC SQL
   SET TRANSACTION NAME t1;

EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE WAIT ISOLATION LEVEL SNAPSHOT;
```

The following table lists the optional SET TRANSACTION parameters for specifying the behavior of the default transaction:

Table 4-3: SET TRANSACTION Parameters

Parameter	Setting	Purpose
Access Mode	READ ONLY or READ WRITE	Describes the type of access this transaction is permitted for a table. For more information about access mode, see “Access Mode,” in this chapter.

Table 4-3: SET TRANSACTION Parameters (Continued)

Parameter	Setting	Purpose
Lock Resolution	WAIT or NO WAIT	Specifies what happens when this transaction encounters a locked row during an update or delete. It either waits for the lock to be released so it can attempt to complete its actions, or it returns an immediate lock conflict error message. For more information about lock resolution, see “Lock Resolution,” in this chapter.
Isolation Level	• SNAPSHOT provides a view of the database at the moment this transaction starts, but prevents viewing changes made by other active transactions. • SNAPSHOT TABLE STABILITY prevents other transactions from making changes to tables that this transaction is reading and updating, but permits them to read rows in the table. • READ COMMITTED reads the most recently committed version of a row during updates and deletions, and allows this transaction to make changes if there is no update conflict with other transactions.	Determines this transaction’s interaction with other simultaneous transactions attempting to access the same tables. READ COMMITTED isolation level also enables a user to specify which version of a row it can read. There are two options: • RECORD_VERSION specifies that the transaction immediately read the latest committed version of a row, even if a more recent uncommitted version also resides on disk. • NO RECORD_VERSION specifies that the transaction can only read the latest version of a row. If WAIT lock resolution is also specified, then the transaction waits until the latest version of a row is committed or rolled back, and retries its read.
Table Reservation	RESERVING	RESERVING specifies a subset of available tables to lock immediately for this transaction to access.
Database Specification	USING	USING specifies a subset of available databases that this transaction can access; it cannot access any other databases. The purpose of this option is to reduce the amount of system resources used by this transaction. Note: USING is not available in DSQL.

The complete syntax of SET TRANSACTION is:

```
EXEC SQL
   SET TRANSACTION [NAME name]
      [READ WRITE| READ ONLY]
      [WAIT | NO WAIT]
      [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
         | READ COMMITTED [[NO] RECORD_VERSION]}]
```

```
        [RESERVING <reserving_clause>
           | USING dbhandle [, dbhandle ...]];

<reserving_clause> = table [, table ...]
   [FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

Transaction options are fully described in the following sections.

Access Mode

The access mode parameter specifies the type of access a transaction has for the tables it uses. There are two possible settings:

- READ ONLY specifies that a transaction can select data from a table, but cannot insert, update, or delete table data.
- READ WRITE specifies that a transaction can select, insert, update, and delete table data. This is the default setting if none is specified.

InterBase assumes that most transactions both read and write data. When starting a transaction for reading and writing, READ WRITE can be omitted from SET TRANSACTION statement. For example, the following statements start a transaction, *t1*, for READ WRITE access:

```
EXEC SQL
   SET TRANSACTION NAME t1;

EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE;
```

Tip It is good programming practice to specify a transaction's access mode, even when it is READ WRITE. It makes an application's source code easier to read and debug because the program's intentions are clearly spelled out.

Start a transaction for READ ONLY access when you only need to read data. READ ONLY *must* be specified. For example, the following statement starts a transaction, *t1*, for read-only access:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ ONLY;
```

Isolation Level

The isolation level parameter specifies the control a transaction exercises over table access. It determines the:

- View of a database the transaction can see.
- Table access allowed to this and other simultaneous transactions.

The following table describes the three isolation levels supported by InterBase:

Table 4-4: ISOLATION LEVEL Options

Isolation Level	Purpose
SNAPSHOT	The default isolation level, provides a stable, committed view of the database at the time the transaction starts. Other simultaneous transactions can UPDATE and INSERT rows, but this transaction cannot see those changes. For updated rows, this transaction sees versions of those rows as they existed at the start of the transaction. If this transaction attempts to update or delete rows changed by another transaction, an update conflict is reported.
SNAPSHOT TABLE STABILITY	Provides a transaction sole insert, update, and delete access to the tables it uses. Other simultaneous transactions may still be able to select rows from those tables.
READ COMMITTED	Enables the transaction to see all committed data in the database, and to update rows updated and committed by other simultaneous transactions without causing lost update problems.

The isolation level for most transactions should be either SNAPSHOT or READ COMMITTED. These levels enable simultaneous transactions to select, insert, update, and delete data in shared databases, and they minimize the chance for lock conflicts. Lock conflicts occur in two situations:

- When a transaction attempts to update a row already updated or deleted by another transaction. A row updated by a transaction is effectively locked for update to all other transactions until the controlling transaction commits or rolls back. READ COMMITTED transactions can read and update rows updated by simultaneous transactions after they commit.
- When a transaction attempts to insert, update, or delete a row in a table locked by another transaction with an isolation level of SNAPSHOT TABLE STABILITY. SNAPSHOT TABLE STABILITY locks entire tables for write access, although concurrent reads by other SNAPSHOT and READ COMMITTED transactions are permitted.

Using SNAPSHOT TABLE STABILITY guarantees that only a single transaction can make changes to tables, but increases the chance of lock conflicts where there are simultaneous transactions attempting to access the same tables. For more information about the likelihood of lock conflicts, see “Isolation Level Interactions,” in this chapter.

Comparing SNAPSHOT, READ COMMITTED, and SNAPSHOT TABLE STABILITY

There are five classic problems all transaction management statements must address:

- *Lost updates,* which can occur if an update is overwritten by a simultaneous transaction unaware of the last updates made by another transaction.
- *Dirty reads,* which can occur if the system allows one transaction to select uncommitted changes made by another transaction.
- *Non-reproducible reads,* which can occur if one transaction is allowed to update or delete rows that are repeatedly selected by another transaction. READ COMMITTED transactions permit non-reproducible reads by design, since they can see committed deletes made by other transactions.
- *Phantom rows,* which can occur if one transaction is allowed to select some, but not all, new rows written by another transaction. READ COMMITTED transactions do not prevent phantom rows.
- *Update side effects,* which can occur when row values are interdependent, and their dependencies are not adequately protected or enforced by locking, triggers, or integrity constraints. These conflicts occur when two or more simultaneous transactions randomly and repeatedly access and update the same data; such transactions are called *interleaved transactions*.

Except as noted, all three InterBase isolation levels control these problems. The following table summarizes how a transaction with a particular isolation level controls access to its data for other simultaneous transactions:

Table 4-5: InterBase Management of Classic Transaction Conflicts

Problem	SNAPSHOT, READ COMMITTED	SNAPSHOT TABLE STABILITY
Lost updates	Other transactions cannot update rows already updated by this transaction.	Other transactions cannot update tables controlled by this transaction.
Dirty reads	Other SNAPSHOT transactions can only read a previous version of a row updated by this transaction. Other READ COMMITTED transactions can only read a previous version, or committed updates.	Other transactions cannot access tables updated by this transaction.

Table 4-5: InterBase Management of Classic Transaction Conflicts (Continued)

Problem	SNAPSHOT, READ COMMITTED	SNAPSHOT TABLE STABILITY
Non-reproducible reads	SNAPSHOT and SNAPSHOT TABLE STABILITY transactions can only read versions of rows committed when they started. READ COMMITTED transactions must expect that reads cannot be reproduced.	SNAPSHOT and SNAPSHOT TABLE STABILITY transactions can only read-versions of rows committed when they started. Other transactions cannot access tables updated by this transaction.
Phantom rows	READ COMMITTED transactions may encounter phantom rows.	Other transactions cannot access tables controlled by this transaction.
Update side effects	Other SNAPSHOT transactions can only read a previous version of a row updated by this transaction. Other READ COMMITTED transactions can only read a previous version, or committed updates. Use triggers and integrity constraints to try to avoid any problems with interleaved transactions.	Other transactions cannot update tables controlled by this transaction. Use triggers and integrity constraints to avoid any problems with interleaved transactions.

Choosing Between SNAPSHOT and READ COMMITTED

The choice between SNAPSHOT and READ COMMITTED isolation levels depends on an application's needs. SNAPSHOT is the default InterBase isolation level. READ COMMITTED duplicates SNAPSHOT behavior, but can read subsequent changes committed by other transactions. In many cases, using READ COMMITTED reduces data contention.

SNAPSHOT transactions receive a stable view of a database as it exists the moment the transactions start. READ COMMITTED transactions can see the latest committed versions of rows. Both types of transactions can use SELECT statements unless they encounter the following conditions:

- Table locked by SNAPSHOT TABLE STABILITY transaction for UPDATE.
- Uncommitted inserts made by other simultaneous transactions. In this case, a SELECT is allowed, but changes cannot be seen.

READ COMMITTED transactions can read the latest committed version of rows. A SNAPSHOT transaction can read only a prior version of the row as it existed before the update occurred.

SNAPHOT and READ COMMITTED transactions with READ WRITE access can use INSERT, UPDATE, and DELETE unless they encounter tables locked by SNAPSHOT TABLE STABILITY transactions.

SNAPSHOT transactions cannot update or delete rows previously updated or deleted and then committed by other simultaneous transactions. Attempting to update a row previously updated or deleted by another transaction results in an update conflict error.

A READ COMMITTED READ WRITE transaction can read changes committed by other transactions, and subsequently update those changed rows.

Occasional update conflicts may occur when simultaneous SNAPSHOT and READ COMMITTED transactions attempt to update the same row at the same time. When update conflicts occur, expect the following behavior:

- For *mass* or *searched updates*, updates where a single UPDATE modifies multiple rows in a table, all updates are undone on conflict. The UPDATE can be retried. For READ COMMITTED transactions, the NO RECORD_VERSION option can be used to narrow the window between reads and updates or deletes. For more information, see "Starting a Transaction With READ COMMITTED Isolation Level," in this chapter.
- For *cursor* or *positioned updates*, where rows are retrieved and updated from an active set one row at a time, only a single update is undone. To retry the update, the cursor must be closed, then reopened, and updates resumed at the point of previous conflict.

For more information about UPDATE through cursors, see Chapter 6: "Working With Data."

Starting a Transaction With SNAPSHOT Isolation Level

InterBase assumes that the default isolation level for transactions is SNAPSHOT. Therefore, SNAPSHOT need not be specified in SET TRANSACTION to set the isolation level. For example, the following statements are equivalent. They both start a transaction, *t1*, for READ WRITE access and set isolation level to SNAPSHOT.

```
EXEC SQL
   SET TRANSACTION NAME t1;

EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE SNAPSHOT;
```

When an isolation level is specified, it must follow the access and lock resolution modes.

Tip It is good programming practice to specify a transaction's isolation level, even when it is SNAPSHOT. It makes an application's source code easier to read and debug because the program's intentions are clearly spelled out.

Starting a Transaction With READ COMMITTED Isolation Level

To start a READ COMMITTED transaction, the isolation level *must* be specified. For example, the following statement starts a named transaction, *t1*, for READ WRITE access and sets isolation level to READ COMMITTED:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE READ COMMITTED;
```

Isolation level always follows access mode. If the access mode is omitted, isolation level is the first parameter to follow the transaction name.

READ COMMITTED supports mutually exclusive optional parameters, RECORD_VERSION and NO RECORD_VERSION. They determine READ COMMITTED behavior when it encounters a row where the latest version of that row is uncommitted:

- RECORD_VERSION, specifies that the transaction immediately read the latest committed version of a row, even if a more recent uncommitted version also resides on disk.
- NO RECORD_VERSION, the default, specifies that the transaction can only read the latest version of a row. If the WAIT lock resolution option is also specified, then the transaction waits until the latest version of a row is committed or rolled back, and retries its read.

Because NO RECORD_VERSION is the default behavior, it need not be specified with READ COMITTED. For example, the following statements are equivalent. They start a named transaction, *t1*, for READ WRITE access and set isolation level to READ COMMITTED NO RECORD_VERSION.

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE READ COMMITTED;

EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE READ COMMITTED
      NO RECORD_VERSION;
```

RECORD_VERSION must always be specified when it is used. For example, the following statement starts a named transaction, *t1*, for READ WRITE access and sets isolation level to READ COMMITTED RECORD_VERSION:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE READ COMMITTED
      RECORD_VERSION;
```

Starting a Transaction With SNAPSHOT TABLE STABILITY Isolation Level

To start a SNAPSHOT TABLE STABILITY transaction, the isolation level *must* be specified. For example, the following statement starts a named transaction, *t1*,

for READ WRITE access and sets isolation level to SNAPSHOT TABLE STABILITY:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE SNAPSHOT TABLE STABILITY;
```

Isolation level always follows the optional access mode and lock resolution parameters, if they are present.

Important Use SNAPSHOT TABLE STABILITY with care. In an environment where multiple transactions share database access, SNAPSHOT TABLE STABILITY greatly increases the likelihood of lock conflicts.

Isolation Level Interactions

To determine the possibility for lock conflicts between two transactions accessing the same database, each transaction's isolation level and access mode must be considered. The following table summarizes possible combinations.

Table 4-6: Isolation Level Interaction with Read (SELECT) and WRITE (UPDATE)

		SNAPSHOT or READ COMMITTED		**SNAPSHOT TABLE STABILITY**	
		UPDATE	**SELECT**	**UPDATE**	**SELECT**
SNAPSHOT or READ COMMITTED	UPDATE	Some simultaneous updates may conflict.	—	Always conflicts.	Always conflicts.
	SELECT	—	—	—	—
SNAPSHOT TABLE STABILITY	UPDATE	Always conflicts.	—	Always conflicts.	Always conflicts.
	SELECT	Always conflicts.	—	Always conflicts.	—

As this table illustrates, SNAPSHOT and READ COMMITTED transactions offer the least chance for conflicts. For example, if *t1* is a SNAPSHOT transaction with READ WRITE access, and *t2* is a READ COMMITTED transaction with READ WRITE access, *t1* and *t2* only conflict when they attempt to update the same rows. If *t1* and *t2* have READ ONLY access, they never conflict with any other transaction.

A SNAPSHOT TABLE STABILITY transaction with READ WRITE access is guaranteed that it alone can update tables, but it conflicts with all other simultaneous transactions except for SNAPSHOT and READ COMMITTED transactions running in READ ONLY mode. A SNAPSHOT TABLE STABILITY transaction with READ ONLY access is compatible with any other read-only transaction, but conflicts with any transaction that attempts to insert, update, or delete data.

Lock Resolution

The lock resolution parameter determines what happens when a transaction encounters a lock conflict. There are two options:

- WAIT, the default, causes the transaction to wait until locked resources are released. Once the locks are released, the transaction retries its operation.
- NO WAIT immediately returns a lock conflict error without waiting for locks to be released.

Because WAIT is the default lock resolution, it need not be specified in a SET TRANSACTION statement. For example, the following statements are equivalent. They both start a transaction, *t1*, for READ WRITE access, WAIT lock resolution, and READ COMMITTED isolation level:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE READ COMMITTED;

EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE WAIT READ COMMITTED;
```

To use NO WAIT, the lock resolution parameter must be specified. For example, the following statement starts the named transaction, *t1*, for READ WRITE access, NO WAIT lock resolution, and SNAPSHOT isolation level:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE NO WAIT READ SNAPSHOT;
```

When lock resolution is specified, it follows the optional access mode, and precedes the optional isolation level parameter.

Tip It is good programming practice to specify a transaction's lock resolution, even when it is WAIT. It makes an application's source code easier to read and debug because the program's intentions are clearly spelled out.

RESERVING Clause

The optional RESERVING clause enables transactions to guarantee themselves specific levels of access to a subset of available tables at the expense of other simultaneous transactions. Reservation takes place at the start of the transaction instead of only when data manipulation statements require a particular level of access. RESERVING is only useful in an environment where simultaneous transactions share database access. It has three main purposes:

- To prevent possible deadlocks and update conflicts that can occur if locks are taken only when actually needed (the default behavior).
- To provide for *dependency locking*, the locking of tables that may be affected by triggers and integrity constraints. While explicit dependency locking is not required, it can assure that update conflicts do not occur because of indirect table conflicts.
- To change the level of shared access for one or more individual tables in a transaction. For example, a READ WRITE SNAPSHOT transaction may need exclusive update rights for a single table, and could use the RESERVING clause to guarantee itself sole write access to the table.

Important A single SET TRANSACTION statement can contain either a RESERVING or a USING clause, but not both.

To reserve tables for a transaction, use the following SET TRANSACTION syntax:

```
EXEC SQL
   SET TRANSACTION [NAME name]
      [READ WRITE| READ ONLY]
      [WAIT | NO WAIT]
      [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
         | READ COMMITTED [[NO] RECORD_VERSION]}]
      RESERVING <reserving_clause>;

<reserving_clause> = table [, table ...]
   [FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

Each table should only appear once in the RESERVING clause. Each table, or a list of tables separated by commas, must be followed by a clause describing the type of reservation requested. The following table lists these reservation options:

Table 4-7: Table Reservation Options for the RESERVING Clause

Reservation Option	Purpose
PROTECTED READ	Prevents other transactions from updating rows. All transactions can select from the table.
PROTECTED WRITE	Prevents other transactions from updating rows. SNAPSHOT and READ COMMITTED transactions can select from the table, but only this transaction can update rows.
SHARED READ	Any transaction can select from this table. Any READ WRITE transaction can update this table. This is the most liberal reservation mode.

Table 4-7: Table Reservation Options for the RESERVING Clause (Continued)

Reservation Option	Purpose
SHARED WRITE	Any SNAPSHOT or READ COMMITTED READ WRITE transaction can update this table. Other SNAPSHOT and READ COMMITTED transactions can also select from this table.

The following statement starts a SNAPSHOT transaction, *t1*, for READ WRITE access, and reserves a single table for PROTECTED WRITE access:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE WAIT SNAPSHOT
      RESERVING EMPLOYEE FOR PROTECTED WRITE;
```

The next statement starts a READ COMMITTED transaction, *t1*, for READ WRITE access, and reserves two tables, one for SHARED WRITE, and another for PROTECTED READ:

```
EXEC SQL
   SET TRANSACTION NAME t1 READ WRITE WAIT READ COMMITTED
      RESERVING EMPLOYEES FOR SHARED WRITE, EMP_PROJ
         FOR PROTECTED READ;
```

SNAPSHOT and READ COMMITTED transactions use RESERVING to implement more restrictive access to tables for other simultaneous transactions. SNAPSHOT TABLE STABILITY transactions use RESERVING to reduce the likelihood of deadlock in critical situations.

USING Clause

Every time a transaction is started, InterBase reserves system resources for each database currently attached for program access. In a multi-transaction, multi-database program, the USING clause can be used to preserve system resources by restricting the number of open databases to which a transaction has access. USING restricts a transaction's access to tables to a listed subset of all open databases using the following syntax:

```
EXEC SQL
   SET TRANSACTION [NAME name]
      [READ WRITE | READ ONLY]
      [WAIT | NO WAIT]
      [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
         | READ COMMITTED [[NO] RECORD_VERSION]}]
      USING dbhandle> [, dbhandle ...];
```

Important A single SET TRANSACTION statement can contain either a USING or a RESERVING clause, but not both.

The following C program fragment opens three databases, *test.gdb*, *research.gdb*, and *employee.gdb*, assigning them to the database handles TEST, RESEARCH, and EMP, respectively. Then it starts the default transaction, and restricts its access to TEST and EMP:

```
. . .
EXEC SQL
   SET DATABASE ATLAS = "test.gdb";
EXEC SQL
   SET DATABASE RESEARCH = "research.gdb";
EXEC SQL
   SET DATABASE EMP = "employee.gdb";
EXEC SQL
   CONNECT TEST, RESEARCH, EMP; /* Open all databases */
EXEC SQL
      SET TRANSACTION USING TEST, EMP;
. . .
```

Using Transaction Names in Data Statements

Once named transactions are started, use their names in INSERT, UPDATE, DELETE, and OPEN statements to specify which transaction controls the statement. For example, the following C code fragment declares two transaction handles, *mytrans1*, and *mytrans2*, initializes them to zero, starts the transactions, and then uses the transaction names to qualify the data manipulation statements that follow:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
   long *mytrans1, *mytrans2;
   char city[26];
EXEC SQL
   END DECLARE SECTION;
mytrans1 = 0L;
mytrans2 = 0L;
. . .
EXEC SQL
   SET DATABASE ATLAS = "atlas.gdb";
EXEC SQL
   CONNECT;
EXEC SQL
   DECLARE CITYLIST CURSOR FOR
      SELECT CITY FROM CITIES
         WHERE COUNTRY = "Mexico";
EXEC SQL
   SET TRANSACTION NAME mytrans1;
EXEC SQL
```

```
   SET TRANSACTION mytrans2 READ ONLY READ COMMITTED;
. . .
printf("Mexican city to add to database: ");
gets(city);
EXEC SQL
   INSERT TRANSACTION mytrans1 INTO CITIES (CITY, COUNTRY)
      VALUES :city, "Mexico";
EXEC SQL
   COMMIT mytrans1;
EXEC SQL
   OPEN TRANSACTION mytrans2 CITYLIST;
EXEC SQL
   FETCH CITYLIST INTO :city;
while (!SQLCODE)
{
   printf("%s\n", city);
   EXEC SQL
      FETCH CITYLIST INTO :city;
}
EXEC SQL
   CLOSE CITYLIST;
EXEC SQL
   COMMIT;
EXEC SQL
   DISCONNECT;
. . .
```

As this example illustrates, a transaction name cannot appear in a DECLARE CURSOR statement. To use a name with a cursor declaration, include the transaction name in the cursor's OPEN statement. The transaction name is not required in subsequent FETCH and CLOSE statements for that cursor.

Note The DSQL EXECUTE and EXECUTE IMMEDIATE statements also support transaction names.

For more information about using transaction names with data manipulation statements, see Chapter 6: "Working With Data." For more information about transaction names and the COMMIT statement, see "Using COMMIT," in this chapter. For more information about using transaction names with DSQL statements, see "Working With Multiple Transactions in DSQL," in this chapter.

Ending a Transaction

When a transaction's tasks are complete, or an error prevents a transaction from completing, the transaction must be ended to set the database to a consistent state. There are two statements that end transactions:

- COMMIT makes a transaction's changes permanent in the database. It signals that a transaction completed all its actions successfully.
- ROLLBACK undoes a transaction's changes, returning the database to its previous state, before the transaction started. ROLLBACK is typically used when one or more errors occur that prevent a transaction from completing successfully.

Both COMMIT and ROLLBACK close the record streams associated with the transaction, reinitialize the transaction name to zero, and release system resources allocated for the transaction. Freed system resources are available for subsequent use by any application or program.

COMMIT and ROLLBACK have additional benefits. They clearly indicate program logic and intention, make a program easier to understand, and most importantly, assure that a transaction's changes are handled as intended by the programmer.

ROLLBACK is frequently used inside error-handling routines to clean up transactions when errors occur. It can also be used to roll back a partially completed transaction prior to retrying it, and it can be used to restore a database to its prior state if a program encounters an unrecoverable error.

Important If the program ends before a transaction ends, a transaction is automatically rolled back, but databases are not closed. If a program ends without closing the database, data loss or corruption is possible. Therefore, open databases should always be closed by issuing explicit DISCONNECT, COMMIT RELEASE, or ROLLBACK RELEASE statements.

For more information about DISCONNECT, COMMIT RELEASE, and ROLLBACK RELEASE, see Chapter 3: "Working With Databases."

Using COMMIT

Use COMMIT to write transaction changes permanently to a database. COMMIT closes the record streams associated with the transaction, resets the transaction name to zero, and frees system resources assigned to the transaction for other uses. The complete syntax for COMMIT is:

```
EXEC SQL
   COMMIT [TRANSACTION name] [RETAIN [SNAPSHOT] | RELEASE dbhandle
      [, dbhandle ...]]
```

For example, the following C code fragment contains a complete transaction. It gives all employees who have worked since December 31, 1992, a 4.3% cost-of-living salary increase. If all qualified employee records are successfully updated,

the transaction is committed, and the changes are actually applied to the database.

```
. . .
EXEC SQL
   SET TRANSACTION SNAPSHOT TABLE STABILITY;
EXEC SQL
   UPDATE EMPLOYEE
      SET SALARY = SALARY * 1.043
      WHERE HIRE_DATE < "1-JAN-1993";
EXEC SQL
   COMMIT;
. . .
```

By default, COMMIT affects only the default transaction, **gds__trans**. To commit another transaction, use its transaction name as a parameter to COMMIT.

Tip Even READ ONLY transactions that do not change a database should be ended with a COMMIT rather than ROLLBACK. The database is not changed, but the overhead required to start subsequent transactions is greatly reduced.

Specifying Transaction Names for COMMIT

To commit changes for transactions other than the default transaction, specify a transaction name as a COMMIT parameter. For example, the following C code fragment starts two transactions using names, and commits them:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      long *TR1, *TR2;
EXEC SQL
   END DECLARE SECTION;
TR1 = 0L;
TR2 = 0L;
. . .
EXEC SQL
   SET TRANSACTION NAME TR1;
EXEC SQL
   SET TRANSACTION NAME TR2;
. . .
/* do actual processsing here */
. . .
EXEC SQL
   COMMIT TRANSACTION TR1;
EXEC SQL
   COMMIT TRANSACTION TR2;
. . .
```

Important In multi-transaction programs, transaction names must always be specified for COMMIT except when committing the default transaction.

Committing Updates Without Freeing a Transaction

To write transaction changes to the database without establishing a new *transaction context*—the names, system resources, and current state of cursors used in a transaction—use the RETAIN option with COMMIT. In a busy, multi-user environment, maintaining the transaction context for each user speeds up processing and uses fewer system resources than closing and starting a new transaction for each action. The syntax for the RETAIN option is:

```
EXEC SQL
   COMMIT [TRANSACTION name] RETAIN [SNAPSHOT];
```

COMMIT RETAIN writes all pending changes to the database, ends the current transaction *without* closing its record stream and cursors and without freeing its system resources, then starts a new transaction and assigns the existing record streams and system resources to the new transaction.

For example, the following C code fragment updates the POPULATION column by user-specified amounts for cities in the CITIES table that are in a country also specified by the user. Each time a qualified row is updated, a COMMIT with the RETAIN option is issued, preserving the current cursor status and system resources.

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char country[26], city[26], asciimult[10];
      int multiplier;
      long pop;
EXEC SQL
   END DECLARE SECTION;
. . .
main ()
{
   EXEC SQL
      DECLARE CHANGEPOP CURSOR FOR
         SELECT CITY, POPULATION
            FROM CITIES
            WHERE COUNTRY = :country;
   printf("Enter country with city populations needing adjustment: ");
   gets(country);
   EXEC SQL
      SET TRANSACTION;
   EXEC SQL
      OPEN CHANGEPOP;
   EXEC SQL
```

```
            FETCH CHANGEPOP INTO :city, :pop;
        while(!SQLCODE)
        {
            printf("City: %s Population: %ld\n", city, pop);
            printf("\nPercent change (100%% to -100%%:");
            gets(asciimult);
            multiplier = atoi(asciimult);
            EXEC SQL
                UPDATE CITIES
                    SET POPULATION  = POPULATION * (1 + :multiplier / 100)
                    WHERE CURRENT OF CHANGEPOP;
            EXEC SQL
                COMMIT RETAIN; /* commit changes, save current state */
            EXEC SQL
                FETCH CHANGEPOP INTO :city, :pop;
            if (SQLCODE && (SQLCODE != 100))
            {
                isc_print_sqlerror(SQLCODE, isc_$status);
                EXEC SQL
                    ROLLBACK;
                EXEC SQL
                    DISCONNECT;
                exit(1);
            }
        }
        EXEC SQL
            COMMIT;
        EXEC SQL
            DISCONNECT;
    }
```

A ROLLBACK executed after a COMMIT RETAIN can only roll back updates and writes occurring *after* the COMMIT RETAIN.

Important In multi-transaction programs, a transaction name *must* always be specified for COMMIT RETAIN except when retaining the state of the default transaction. For more information about transaction names, see "Naming Transactions," in this chapter.

Using ROLLBACK

Use ROLLBACK to restore the database to its condition prior to the start of the transaction. ROLLBACK also closes the record streams associated with the transaction, resets the transaction name to zero, and frees system resources assigned to the transaction for other uses. ROLLBACK typically appears in error-handling routines. The syntax for ROLLBACK is:

```
EXEC SQL
    ROLLBACK [TRANSACTION name] [RELEASE [dbhandle [, dbhandle ...]]];
```

For example, the following C code fragment contains a complete transaction. It gives all employees who have worked since December 31, 1992, a 4.3% cost-of-living salary adjustment. If all qualified employee records are successfully updated, the transaction is committed, and the changes are actually applied to the database. If an error occurs, all changes made by the transaction are undone, and the database is restored to its condition prior to the start of the transaction.

```
. . .
EXEC SQL
   SET TRANSACTION SNAPSHOT TABLE STABILITY;
EXEC SQL
   UPDATE EMPLOYEES
      SET SALARY = SALARY * 1.043
      WHERE HIRE_DATE < "1-JAN-1993";
if (SQLCODE && (SQLCODE != 100))
{
   isc_print_sqlerror(SQLCODE, isc_$status);
   EXEC SQL
      ROLLBACK;
   EXEC SQL
      DISCONNECT;
   exit(1);
}
EXEC SQL
   COMMIT;
EXEC SQL
   DISCONNECT;
. . .
```

By default, ROLLBACK affects only the default transaction, **gds__trans**. To roll back other transactions, use their transaction names as parameters to ROLLBACK.

Working With Multiple Transactions

Because InterBase provides support for transaction names, a program can use as many transactions at once as necessary to carry out its work. Each simultaneous transaction in a program requires its own name. A transaction's name distinguishes it from other active transactions. The name can also be used in data manipulation and transaction management statements to specify which transaction controls the statement. For more information about declaring and using transaction names, see "Starting a Named Transaction," in this chapter.

There are four steps for using named transactions in a program:

1. Declare a unique host-language variable for each transaction name.

2. Initialize each transaction variable to zero.
3. Use SET TRANSACTION to start each transaction using an available transaction name.
4. Use the transaction names as parameters in subsequent transaction management and data manipulation statements that should be controlled by a specified transaction.

Multi-transaction Programs and the Default Transaction

In multi-transaction programs, it is good programming practice to supply a transaction name for every transaction a program defines. One transaction in a multi-transaction program can be the default transaction, **gds__trans**. When the default transaction is used in multi-transaction programs, it, too, should be started explicitly and referenced by name in data manipulation statements.

If the transaction name is omitted from a transaction management or data manipulation statement, InterBase assumes the statement affects the default transaction. If the default transaction has not been explicitly started with a SET TRANSACTION statement, then during preprocessing, **gpre** inserts a statement to start it.

Important DSQL programs must be preprocessed with the **gpre -m** switch. In this mode, **gpre** does *not* generate the default transaction automatically, but instead reports an error. DSQL programs require that all transactions be explicitly started.

Using Cursors in Multi-transaction Programs

DECLARE CURSOR does not support transaction names. Instead, to associate a named transaction with a cursor, include the transaction name as an optional parameter in the cursor's OPEN statement. A cursor can only be associated with a single transaction. For example, the following statements declare a cursor, and open it, associating it with the transaction, *t1*:

```
. . .
EXEC SQL
   DECLARE S CURSOR FOR
      SELECT COUNTRY, CUST_NO, SUM(QTY_ORDERED)
      FROM SALES
      GROUP BY CUST_NO
         WHERE COUNTRY = "Mexico";
EXEC SQL
   SET TRANSACTION t1 READ ONLY READ COMMITTED;
. . .
EXEC SQL
```

```
   OPEN TRANSACTION t1 S;
. . .
```

An OPEN statement without the optional transaction name parameter operates under control of the default transaction, **gds__trans**.

Once a named transaction is associated with a cursor, subsequent cursor statements automatically operate under control of that transaction. Therefore, it does not support a transaction name parameter. For example, the following statements illustrate a FETCH and CLOSE for the S cursor after it is associated with the named transaction, *t2*:

```
. . .
EXEC SQL
   OPEN TRANSACTION t2 S;
EXEC SQL
   FETCH S INTO :country, :cust_no, :qty;
while (!SQLCODE)
{
       printf("%s %d %d\n", country, cust_no, qty);
   EXEC SQL
      FETCH S INTO :country, :cust_no, :qty;
}
EXEC SQL
   CLOSE S;
. . .
```

Multiple cursors can be controlled by a single transaction, or each transaction can control a single cursor according to a program's needs.

A Multi-transaction Example

The following C code illustrates the steps required to create a simple multi-transaction program. It declares two transaction handles, *mytrans1*, and *mytrans2*, initializes them to zero, starts the transactions, and then uses the transaction names to qualify the data manipulation statements that follow. It also illustrates the use of a cursor with a named transaction.

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
   long *mytrans1 = 0L, *mytrans2 = 0L;
   char city[26];
EXEC SQL
   END DECLARE SECTION;
. . .
EXEC SQL
   DECLARE CITYLIST CURSOR FOR
      SELECT CITY FROM CITIES
         WHERE COUNTRY = "Mexico";
```

```
EXEC SQL
   SET TRANSACTION NAME mytrans1;
EXEC SQL
   SET TRANSACTION mytrans2 READ ONLY READ COMMITTED;
. . .
printf("Mexican city to add to database: ");
gets(city);
EXEC SQL
   INSERT TRANSACTION mytrans1 INTO CITIES
      VALUES :city, "Mexico", NULL, NULL, NULL, NULL;
EXEC SQL
   COMMIT mytrans1;
EXEC SQL
   OPEN TRANSACTION mytrans2 CITYLIST;
EXEC SQL
   FETCH CITYLIST INTO :city;
while (!SQLCODE)
{
   printf("%s\n", city);
   EXEC SQL
      FETCH CITYLIST INTO :city;
}
EXEC SQL
   CLOSE CITYLIST;
EXEC SQL
   COMMIT;
EXEC SQL
   DISCONNECT
. . .
```

Working With Multiple Transactions in DSQL

In InterBase, DSQL applications can also use multiple transactions, but with the following limitations:

- Programs must be preprocessed with the **gpre -m** switch.
- Transaction names must be declared statically. They cannot be defined through user-modified host variables at run time.
- Transaction names must be initialized to zero before appearing in DSQL statements.
- All transactions must be started with explicit SET TRANSACTION statements.
- No data definition language (DDL) can be used in the context of a named transaction in an embedded program; DDL must always occur in the context of the default transaction, **gds__trans**.

- As long as a transaction name parameter is not specified with a SET TRANSACTION statement, it can follow a PREPARE statement to modify the behavior of a subsequently named transaction in an EXECUTE or EXECUTE IMMEDIATE statement. This enables a user to modify transaction behaviors at run time.

Transaction names are fixed for all InterBase programs during preprocessing, and cannot be dynamically assigned. A user can still modify DSQL transaction behavior at run time. It is up to the programmer to anticipate possible transaction behavior modification and plan for it. The following section describes how users can modify transaction behavior.

Modifying Transaction Behavior With SET TRANSACTION

The number and name of transactions available to a DSQL program is fixed when the program is preprocessed with **gpre**, the InterBase preprocessor. The programmer determines both the named transactions that control each DSQL statement in a program, and the default behavior of those transactions. A user can change a named transaction's behavior at run time.

In DSQL programs, a user enters an SQL statement into a host-language string variable, then the host variable is processed in a PREPARE statement or EXECUTE IMMEDIATE statement. PREPARE:

- Checks the statement in the variable for errors.
- Loads the statement into an XSQLDA for a subsequent EXECUTE statement.

EXECUTE IMMEDIATE:

- Checks the statement for errors.
- Loads the statement into the XSQLDA.
- Executes the statement.

Both EXECUTE and EXECUTE IMMEDIATE operate within the context of a programmer-specified transaction, which can be a named transaction. If the transaction name is omitted, these statements are controlled by the default transaction, **gds__trans**.

The transaction behavior for an EXECUTE or EXECUTE IMMEDIATE can be modified by:

- Enabling a user to enter a SET TRANSACTION statement into a host variable.

- Executing the SET TRANSACTION statement before the EXECUTE or EXECUTE IMMEDIATE whose transaction context should be modified.

In this context, a SET TRANSACTION statement changes the behavior of the next transaction, named or default, until another SET TRANSACTION occurs.

For example, the following C code fragment provides the user the option of specifying a new transaction behavior, applies the behavior change, executes the next user statement in the context of that changed transaction, then restores the transaction's original behavior.

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char usertrans[512], query[1024];
      char deftrans[] = {"SET TRANSACTION READ WRITE WAIT SNAPSHOT"};
EXEC SQL
   END DECLARE SECTION;
. . .
printf("\nEnter SQL statement: ");
gets(query);
printf("\nChange transaction behavior (Y/N)? ");
gets(usertrans);
if (usertrans[0] == "Y" || usertrans[0] == "y")
{
   printf("\nEnter \"SET TRANSACTION\" and desired behavior: ");
   gets(usertrans);
   EXEC SQL
      COMMIT usertrans;
   EXEC SQL
      EXECUTE IMMEDIATE usertrans;
}
else
{
   EXEC SQL
      EXECUTE IMMEDIATE deftrans;
}
EXEC SQL
   EXECUTE IMMEDIATE query;
EXEC SQL
   EXECUTE IMMEDIATE deftrans;
. . .
```

Important As this example illustrates, you must commit or roll back any previous transactions before you can execute SET TRANSACTION.

CHAPTER 5

Working With Data Definition Statements

This chapter discusses how to create, modify, and delete databases, tables, views, and indexes in SQL applications. A database's tables, views, and indexes make up most of its underlying structure, or *metadata*.

Important The discussion in this chapter applies equally to dynamic SQL (DSQL) applications, except that users enter DSQL data definition statements at run time, and do not preface those statements with EXEC SQL.

The preferred method for creating, modifying, and deleting metadata is through the InterBase interactive SQL tool, **isql**, but in some instances, it may be necessary or desirable to embed some data definition capabilities in an SQL application. Both SQL and DSQL applications can use the following subset of data definition statements:

Table 5-1: Data Definition Statements Supported for Embedded Applications

CREATE Statement	ALTER Statement	DROP Statement
CREATE DATABASE	ALTER DATABASE	—
CREATE DOMAIN	ALTER DOMAIN	DROP DOMAIN
CREATE GENERATOR	SET GENERATOR	—
CREATE INDEX	ALTER INDEX	DROP INDEX
CREATE SHADOW	ALTER SHADOW	DROP SHADOW
CREATE TABLE	ALTER TABLE	DROP TABLE
CREATE VIEW	—	DROP VIEW
DECLARE EXTERNAL	—	DROP EXTERNAL
DECLARE FILTER	—	DROP FILTER

DSQL also supports creating, altering, and dropping stored procedures, triggers, and exceptions. DSQL is especially powerful for data definition because it enables users to enter any supported data definition statement at run time. For example, **isql** itself is a DSQL application. For more information about using

isql to define stored procedures, triggers, and exceptions, see the *Data Definition Guide*. For a complete discussion of DSQL programming, see Chapter 15: "Using Dynamic SQL."

Creating Metadata

SQL data definition statements are used in applications the sole purpose of which is to create or modify databases or tables. Typically the expectation is that these applications will be used only once by any given user, then discarded, or saved for later modification by a database designer who can read the program code as a record of a database's structure. If data definition changes must be made, editing a copy of existing code is easier than starting over.

Note Use the InterBase interactive SQL tool, **isql**, to create and alter data definitions whenever possible. For more information about **isql**, see the *Windows Client User's Guide*.

The SQL CREATE statement is used to make new databases, domains, tables, views, or indexes. A COMMIT statement must follow every CREATE so that subsequent CREATE statements can use previously defined metadata upon which they may rely. For example, domain definitions must be committed before the domain can be referenced in subsequent table definitions.

Important Applications that mix data definition and data manipulation must be preprocessed using the **gpre -m** switch. Such applications must explicitly start every transaction with SET TRANSACTION.

Creating a Database

CREATE DATABASE establishes a new database and its *system tables*, tables that describe the internal structure of the database. InterBase uses the system tables whenever an application accesses a database. SQL programs can read the data in most of these tables just like any user-created table.

In its most elementary form, the syntax for CREATE DATABASE is:

```
EXEC SQL
   CREATE DATABASE "<filespec>";
```

CREATE DATABASE must appear before any other CREATE statements. It requires one parameter, the name of a database to create. For example, the following statement creates a database named *employee.gdb*:

```
EXEC SQL
   CREATE DATABASE "employee.gdb";
```

Note The database name can include a full file specification, including both host or node names, and a directory path to the location where the database file should be created. For information about file specifications for a particular operating system, see the operating system manuals.

Important Although InterBase enables access to remote databases, when a database is created, it should only be created directly on the machine where it is to reside.

There are optional parameters for CREATE DATABASE. For example, when an application running on a client attempts to connect to an InterBase server in order to create a database, it may be expected to provide USER and PASSWORD parameters before the connection is established. Other parameters specify the database page size, the number and size of multi-file databases, and the default character set for the database.

For more information about specifying a default character set for a database, see "Specifying a Default Character Set for a Database," in this chapter. For a complete discussion of all CREATE DATABASE parameters, see the *Data Definition Guide*. For the complete syntax of CREATE DATABASE, see the *Language Reference*.

Important An application that creates a database must be preprocessed with the **gpre -m** switch. It must also create at least one table. If a database is created without a table, it cannot be successfully opened by another program. Applications that perform both data definition and data manipulation must declare tables with DECLARE TABLE before creating and populating them. For more information about table creation, see "Creating a Table," in this chapter.

Specifying a Default Character Set for a Database

A database's default character set designation specifies the character set the server uses to transliterate and store CHAR, VARCHAR, and text BLOB data in the database when no other character set information is provided. A default character set should always be specified for a database when it is created with CREATE DATABASE.

To specify a default character set, use the DEFAULT CHARACTER SET clause of CREATE DATABASE. For example, the following statement creates a database that uses the ISO8859_1 character set:

```
EXEC SQL
   CREATE DATABASE "europe.gdb" DEFAULT CHARACTER SET ISO8859_1;
```

Important If you do not specify a character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption

for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot later move that data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.

For a complete description of the DEFAULT CHARACTER SET clause and a list of the character sets supported by InterBase, see the *Data Definition Guide*.

Creating a Domain

CREATE DOMAIN creates a column definition that is global to the database, and that can be used to define columns in subsequent CREATE TABLE statements. CREATE DOMAIN is especially useful when many tables in a database contain identical column definitions. For example, in an employee database, several tables might define columns for employees' first and last names.

At its simplest, the syntax for CREATE DOMAIN is:

```
EXEC SQL
   CREATE DOMAIN name AS <datatype>;
```

The following statements create two domains, FIRSTNAME, and LASTNAME.

```
EXEC SQL
   CREATE DOMAIN FIRSTNAME AS VARCHAR(15);
EXEC SQL
   CREATE DOMAIN LASTNAME AS VARCHAR(20);
EXEC SQL
   COMMIT;
```

Once a domain is defined and committed, it can be used in CREATE TABLE statements to define columns. For example, the following CREATE TABLE fragment illustrates how the FIRSTNAME and LASTNAME domains can be used in place of column definitions in the EMPLOYEE table definition.

```
EXEC SQL
   CREATE TABLE EMPLOYEE
   (
      . . .
      FIRST_NAME FIRSTNAME NOT NULL,
      LAST_NAME LASTNAME NOT NULL;
      . . .
   );
```

A domain definition can also specify a default value, a NOT NULL attribute, a CHECK constraint that limits inserts and updates to a range of values, a character set, and a collation order.

For more information about creating domains and using them during table creation, see the *Data Definition Guide*. For the complete syntax of CREATE DOMAIN, see the *Language Reference*.

Creating a Table

The CREATE TABLE statement defines a new database table and the columns and integrity constraints within that table. Each column can include a character set specification and a collation order specification. CREATE TABLE also automatically imposes a default SQL security scheme on the table. The person who creates a table becomes its owner. A table's owner is assigned all privileges for it, including the right to grant privileges to other users.

A table can only be created for a database that already exists. At its simplest, the syntax for CREATE TABLE is as follows:

```
EXEC SQL
   CREATE TABLE name (<col_def> | <table_constraint>
      [, <col_def> | <table_constraint> ...]);
```

<*col_def*> defines a column using the following syntax:

```
col {<datatype> | COMPUTED [BY] (<expr>) | domain} <col_constraint>
   COLLATE collation
```

col must be a column name unique within the table definition.

<*datatype*> specifies the SQL data type to use for column entries. COMPUTED BY can be used to define a column whose value is computed from an expression when the column is accessed at run time.

Note <*col_constraint*> is an optional integrity constraint to apply to a column. <*tableconstraint*> is an optional integrity constraint to apply to an entire table. Integrity constraints are used to ensure data entered in a table meets specific requirements, to specify that data entered in a table or column is unique, or to enforce referential integrity with other tables in the database.

The following code fragment contains SQL statements that create a database, *employee.gdb*, and create a table, EMPLOYEE_PROJECT, with three columns, EMP_NO, PROJ_ID, and DUTIES:

```
EXEC SQL
   CREATE DATABASE "employee.gdb";
EXEC SQL
   CREATE TABLE EMPLOYEE_PROJECT
   (
      EMP_NO SMALLINT NOT NULL,
      PROJ_ID CHAR(5) NOT NULL,
      DUTIES BLOB SUBTYPE 1 SEGMENT SIZE 240
```

```
    );
EXEC SQL
    COMMIT;
```

An application can create multiple tables, but duplicating an existing table name is not permitted.

For more information about SQL data types and integrity constraints, see the *Data Definition Guide*. For more information about CREATE TABLE syntax, see the *Language Reference*. For more information about changing or assigning table privileges, see Chapter 10: "Working With Security."

Creating a Computed Column

A *computed column* is one whose value is calculated when the column is accessed at run time. The value can be derived from any valid SQL expression that results in a single, non-array value.

To create a computed column, use the following column declaration syntax in CREATE TABLE:

```
col COMPUTED [BY] (<expr>)
```

The expression can reference previously defined columns in the table. For example, the following statement creates a computed column, FULL_NAME, by concatenating two other columns, LAST_NAME, and FIRST_NAME:

```
EXEC SQL
    CREATE TABLE EMPLOYEE
    (
        . . .
        FIRST_NAME VARCHAR(10) NOT NULL,
        LAST_NAME VARCHAR(15) NOT NULL,
        . . .
        FULL_NAME COMPUTED BY (LAST_NAME || ", " || FIRST_NAME)
    );
```

For more information about COMPUTED BY, see the *Data Definition Guide*.

Declaring and Creating a Table

In programs that mix data definition and data manipulation, the DECLARE TABLE statement must be used to describe a table's structure to the InterBase preprocessor, **gpre**, before that table can be created. During preprocessing, if **gpre** encounters a DECLARE TABLE statement, it stores the table's description for later reference. When **gpre** encounters a CREATE TABLE statement for the previously declared table, it verifies that the column descriptions in the CREATE

statement match those in the DECLARE statement. If they do not match, **gpre** reports the errors and cancels preprocessing so that the error can be fixed.

When used, DECLARE TABLE must come before the CREATE TABLE statement it describes. For example, the following code fragment declares a table, EMPLOYEE_PROJ, then creates it:

```
EXEC SQL
   DECLARE EMPLOYEE_PROJECT TABLE
   (
      EMP_NO SMALLINT,
      PROJ_ID CHAR(5),
      DUTIES BLOB(240, 1)
   );
EXEC SQL
   CREATE TABLE EMPLOYEE_PROJECT
   (
      EMP_NO SMALLINT,
      PROJ_ID CHAR(5),
      DUTIES BLOB(240, 1)
   );
EXEC SQL
   COMMIT;
```

For more information about DECLARE TABLE, see the *Language Reference*.

Creating a View

A *view* is a virtual table that is based on a subset of one or more actual tables in a database. Views are used to:

- Restrict user access to data by presenting only a subset of available data.
- Rearrange and present data from two or more tables in a manner especially useful to the program.

Unlike a table, a view is not stored in the database as raw data. Instead, when a view is created, the definition of the view is stored in the database. When a program uses the view, InterBase reads the view definition and quickly generates the output as if it were a table.

To make a view, use the following CREATE VIEW syntax:

```
EXEC SQL
   CREATE VIEW name [(view_col [, view_col ...)] AS
      <select> [WITH CHECK OPTION];
```

The name of the view, *name*, must be unique within the database.

To give each column displayed in the view its own name, independent of its column name in an underlying table, enclose a list of *view_col* parameters in parentheses. Each column of data returned by the view's SELECT statement is assigned sequentially to a corresponding view column name. If a list of view column names is omitted, column names are assigned directly from the underlying table.

Listing independent names for columns in a view ensures that the appearance of a view does not change if its underlying table structures are modified.

Note A view column name must be provided for each column of data returned by the view's SELECT statement, or else no view column names should be specified.

The <*select*> is a standard SELECT statement that specifies the selection criteria for rows to include in the view. A SELECT in a view may not include an ORDER BY clause. In DSQL, the UNION clause is also forbidden.

The optional WITH CHECK OPTION is used to restrict inserts, updates, and deletes in a view that can be updated. For more information about views allowing update, and the WITH CHECK OPTION, see "Creating a View for Update," in this chapter.

To create a read-only view, a view's creator must have SELECT privilege for the table or tables underlying the view. To create a view for update requires ALL privilege for the table or tables underlying the view. For more information about SQL privileges, see Chapter 10: "Working With Security."

Creating a View for SELECT

Many views combine data from multiple tables or other views. A view based on multiple tables or other views can be read, but not updated. For example, the following statement creates a read-only view, PHONE_LIST, because it joins two tables, EMPLOYEE, and DEPARTMENT:

```
EXEC SQL
   CREATE VIEW PHONE_LIST AS
      SELECT EMP_NO, FIRST_NAME, LAST_NAME, LOCATION, PHONE_NO
         FROM EMPLOYEE, DEPARTMENT
         WHERE EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO;
EXEC SQL
   COMMIT;
```

Important Only a view's creator initially has access to it. To assign read access to others, use GRANT. For more information about GRANT, see Chapter 10: "Working With Security."

Creating a View for Update

An *updatable view* is one that enables privileged users to insert, update, and delete information in the view's base table. To be updatable, a view must meet the following conditions:

- It derives its columns from a single table or updatable view.
- It does *not* define a self-join of the base table.
- It does *not* reference columns derived from arithmetic expressions.
- The view's SELECT statement does *not* contain:
  - A WHERE clause that uses the DISTINCT predicate.
  - A HAVING clause.
  - Functions.
  - Nested queries.
  - Stored procedures.

For example, the following view, HIGH_CITIES, is an updatable view. It selects all cities in the CITIES table with altitudes greater than or equal to a half mile.

```
EXEC SQL
   CREATE VIEW HIGH_CITIES AS
      SELECT CITY, COUNTRY_NAME, ALTITUDE FROM CITIES
      WHERE ALTITUDE >= 2640;
EXEC SQL
   COMMIT;
```

Users who have INSERT and UPDATE privileges for this view can change rows in or add new rows to the view's underlying table, CITIES. They can even insert or update rows that cannot be displayed by the HIGH_CITIES view. The following INSERT adds a record for Santa Cruz, California, altitude 23 feet, to the CITIES table:

```
EXEC SQL
   INSERT INTO HIGH_CITIES (CITY, COUNTRY_NAME, ALTITUDE)
   VALUES ("Santa Cruz", "United States", "23");
```

To restrict inserts and updates through a view to only those rows that can be selected by the view, use the WITH CHECK OPTION in the view definition. For example, the following statement defines the view, HIGH_CITIES, to use the WITH CHECK OPTION. Users with INSERT and UPDATE privileges will only be able to enter rows for cities with altitudes greater than or equal to a half mile.

```
EXEC SQL
   CREATE VIEW HIGH_CITIES AS
      SELECT CITY, COUNTRY_NAME, ALTITUDE FROM CITIES
      WHERE ALTITUDE > 2640 WITH CHECK OPTION;
```

Creating an Index

SQL provides CREATE INDEX for establishing user-defined database indexes. An index, based on one or more columns in a table, is used to speed data retrieval for queries that access those columns. The syntax for CREATE INDEX is:

```
EXEC SQL
   CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX <index> ON
      table (col [, col ...]);
```

For example, the following statement defines an index, NAMEX, for the LAST_NAME and FIRST_NAME columns in the EMPLOYEE table:

```
EXEC SQL
   CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Note InterBase automatically generates system-level indexes when tables are defined using UNIQUE and PRIMARY KEY constraints. For more information about constraints, see the *Data Definition Guide*.

For more information about CREATE INDEX syntax, see the *Language Reference*.

Preventing Duplicate Index Entries

To define an index that eliminates duplicate entries, include the UNIQUE keyword in CREATE INDEX. The following statement creates a unique index, PRODTYPEX, on the PROJECT table:

```
EXEC SQL
   CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

Important After a unique index is defined, users cannot insert or update values in indexed columns if those values already exist there. For unique indexes defined on multiple columns, like PRODTYPEX in the previous example, the same value can be entered within individual columns, but the combination of values entered in all columns defined for the index must be unique.

Specifying Index Sort Order

By default, SQL stores an index in ascending order. To make a descending sort on a column or group of columns more efficient, use the DESCENDING keyword to define the index. For example, the following statement creates an index, CHANGEX, based on the CHANGE_DATE column in the SALARY_HISTORY table:

```
EXEC SQL
   CREATE DESCENDING INDEX CHANGEX ON SALARY_HISTORY (CHANGE_DATE);
```

Note To retrieve indexed data in descending order, use ORDER BY in the SELECT statement to specify retrieval order.

Creating Generators

A *generator* is a monotonically increasing or decreasing numeric value that is inserted in a field either directly by an SQL statement in an application or through a trigger. Generators are often used to produce unique values to insert into a column used as a primary key.

To create a generator for use in an application, use the following CREATE GENERATOR syntax:

```
EXEC SQL
   CREATE GENERATOR name;
```

The following statement creates a generator, EMP_NO_GEN, to specify a unique employee number:

```
EXEC SQL
   CREATE GENERATOR EMP_NO_GEN;
EXEC SQL
   COMMIT;
```

Once a generator is created, the starting value for a generated number can be specified with SET GENERATOR. To insert a generated number in a field, use the InterBase library GEN_ID() function in an assignment statement. For more information about GEN_ID(), CREATE GENERATOR, and SET GENERATOR, see the *Data Definition Guide*.

Dropping Metadata

SQL supports several statements for deleting existing metadata:

- DROP TABLE, to delete a table from a database
- DROP VIEW, to delete a view definition from a database
- DROP INDEX, to delete a database index
- ALTER TABLE, to delete columns from a table

For more information about deleting columns with ALTER TABLE, see "Altering a Table," in this chapter.

Dropping an Index

To delete an index, use DROP INDEX. An index can only be dropped by its creator, the SYSDBA, or a user with root privileges. If an index is in use when the drop is attempted, the drop is postponed until the index is no longer in use. The syntax of DROP INDEX is:

```
EXEC SQL
   DROP INDEX name;
```

name is the name of the index to delete. For example, the following statement drops the index, NEEDX:

```
EXEC SQL
   DROP INDEX NEEDX;
EXEC SQL
   COMMIT;
```

Deletion fails if the index is on a UNIQUE, PRIMARY KEY, or FOREIGN KEY integrity constraint. To drop an index on a UNIQUE, PRIMARY KEY, or FOREIGN KEY integrity constraint, first drop the constraints, the constrained columns, or the table.

For more information about DROP INDEX and dropping integrity constraints, see the *Data Definition Guide*.

Dropping a View

To delete a view, use DROP VIEW. A view can only be dropped by its owner, the SYSDBA, or a user with root privileges. If a view is in use when a drop is

attempted, the drop is postponed until the view is no longer in use. The syntax of DROP VIEW is:

```
EXEC SQL
   DROP VIEW name;
```

name is the name of the view to delete. For example, the following statement drops the EMPLOYEE_SALARY view:

```
EXEC SQL
   DROP VIEW EMPLOYEE_SALARY;
EXEC SQL
   COMMIT;
```

Deleting a view fails if a view is used in another view, a trigger, or a computed column. To delete a view that meets any of these conditions:

1. Delete the other view, trigger, or computed column.
2. Delete the view.

For more information about DROP VIEW, see the *Data Definition Guide.*

Dropping a Table

To remove a table from a database, use DROP TABLE. A table can only be dropped by its owner, the SYSDBA, or a user with root privileges. If a table is in use when a drop is attempted, the drop is postponed until the table is no longer in use. The syntax of DROP TABLE is:

```
EXEC SQL
   DROP TABLE name;
```

name is the name of the table to drop. For example, the following statement drops the EMPLOYEE table:

```
EXEC SQL
   DROP TABLE EMPLOYEE;
EXEC SQL
   COMMIT;
```

Deleting a table fails if a table is used in a view, a trigger, or a computed column. A table cannot be deleted if a UNIQUE or PRIMARY KEY integrity constraint is defined for it, and the constraint is also referenced by a FOREIGN KEY in another table. To drop the table, first drop the FOREIGN KEY constraints in the other table, then drop the table.

Note Columns within a table can be dropped without dropping the rest of the table. For more information, see "Dropping an Existing Column From a Table," in this chapter.

For more information about DROP TABLE, see the *Data Definition Guide*.

Altering Metadata

Most changes to data definitions are made at the table level, and involve adding new columns to a table, or dropping obsolete columns from it. SQL provides ALTER TABLE to add new columns to a table and to drop existing columns. A single ALTER TABLE can carry out a single operation, or both operations.

Making changes to views and indexes always requires two separate statements:

1. Drop the existing definition.
2. Create a new definition.

If current metadata cannot be dropped, replacement definitions cannot be added. Dropping metadata can fail for the following reasons:

- The person attempting to drop metadata is not the metadata's creator.
- SQL integrity constraints are defined for the metadata and referenced in other metadata.
- The metadata is used in another view, trigger, or computed column.

For more information about dropping metadata, see "Dropping Metadata," in this chapter.

Altering a Table

ALTER TABLE enables the following changes to an existing table:

- Adding new column definitions
- Adding new table constraints
- Dropping existing column definitions
- Dropping existing table constraints
- Changing column definitions by dropping existing definitions, and adding new ones

- Changing existing table constraints by dropping existing definitions, and adding new ones

The simple syntax of ALTER TABLE is as follows:

```
EXEC SQL
   ALTER TABLE name {ADD colname <datatype> [NOT NULL]
   | DROP colname | ADD CONSTRAINT constraintname tableconstraint
   | DROP CONSTRAINT constraintname};
```

Note For information about adding, dropping, and modifying constraints at the table level, see the *Data Definition Guide*.

For the complete syntax of ALTER TABLE, see the *Language Reference*.

Adding a New Column to a Table

To add another column to an existing table, use ALTER TABLE. A table can only be modified by its creator. The syntax for adding a column with ALTER TABLE is:

```
EXEC SQL
   ALTER TABLE name ADD colname <datatype> colconstraint
      [, ADD colname datatype colconstraint ...];
```

For example, the following statement adds a column, EMP_NO, to the EMPLOYEE table:

```
EXEC SQL
   ALTER TABLE EMPLOYEE ADD EMP_NO EMPNO NOT NULL;
EXEC SQL
   COMMIT;
```

Note This example makes use of a domain, EMPNO, to define a column. For more information about domains, see the *Data Definition Guide*.

Multiple columns can be added to a table at the same time. Separate column definitions with commas. For example, the following statement adds two columns, EMP_NO, and FULL_NAME, to the EMPLOYEE table. FULL_NAME is a computed column, a column that derives it values from calculations based on other columns:

```
EXEC SQL
   ALTER TABLE EMPLOYEE
      ADD EMP_NO EMPNO NOT NULL,
      ADD FULL_NAME COMPUTED BY (LAST_NAME || ', ' || FIRST_NAME);
EXEC SQL
   COMMIT;
```

Note This example creates a column using a value computed from two other columns already defined for the EMPLOYEE table. For more information about creating computed columns, see the *Data Definition Guide.*

New columns added to a table may be defined with integrity constraints. For more information about adding columns with integrity constraints to a table, see the *Data Definition Guide.*

Dropping an Existing Column From a Table

To delete a column definition and its data from a table, use ALTER TABLE. A column can only be dropped by the owner of the table, the SYSDBA, or a user with root privileges. If a table is in use when a column is dropped, the drop is postponed until the table is no longer in use. The syntax for dropping a column with ALTER TABLE is:

```
EXEC SQL
   ALTER TABLE name DROP colname [, colname ...];
```

For example, the following statement drops the EMP_NO column from the EMPLOYEE table:

```
EXEC SQL
   ALTER TABLE EMPLOYEE DROP EMP_NO;
EXEC SQL
   COMMIT;
```

Multiple columns can be dropped with a single ALTER TABLE. The following statement drops the EMP_NO and FULL_NAME columns from the EMPLOYEE table:

```
EXEC SQL
   ALTER TABLE EMPLOYEE
      DROP EMP_NO,
      DROP FULL_NAME;
EXEC SQL
   COMMIT;
```

Deleting a column fails if the column is part of a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. To drop the column, first drop the constraint, then the column.

Deleting a column also fails if the column is used by a CHECK constraint for another column. To drop the column, first drop the CHECK constraint, then drop the column.

For more information about integrity constraints, see the *Data Definition Guide.*

Modifying a Column

An existing column definition can be modified using ALTER TABLE, but if data already stored in that column is not preserved *before* making changes, it will be lost.

Preserving data entered in a column and modifying the definition for a column, is a six-step process:

1. Adding a new, temporary column to the table that mirrors the current metadata of the column to be changed.
2. Copying the data from the column to be changed to the newly created temporary column.
3. Dropping the column to change.
4. Adding a new column definition, giving it the same name that the previously dropped column had.
5. Copying data from the temporary column to the redefined column.
6. Dropping the temporary column.

For example, suppose the EMPLOYEE table contains a column, OFFICE_NO, defined to hold a data type of CHAR(3), and suppose that the size of the column needs to be increased by one. The following numbered sequence describes each step and provides sample code:

1. First, create a temporary column to hold the data in OFFICE_NO during the modification process:

```
EXEC SQL
   ALTER TABLE EMPLOYEE ADD TEMP_NO CHAR(3);
EXEC SQL
   COMMIT;
```

2. Move existing data from OFFICE_NO to TEMP_NO to preserve it:

```
EXEC SQL
   UPDATE EMPLOYEE
      SET TEMP_NO = OFFICE_NO;
```

3. After the data is moved, drop the OFFICE_NO column:

```
EXEC SQL
   ALTER TABLE DROP OFFICE_NO;
EXEC SQL
   COMMIT;
```

4. Add a new column definition for OFFICE_NO, specifying the data type and new size:

```
EXEC SQL
   ALTER TABLE ADD OFFICE_NO CHAR (4);
EXEC SQL
   COMMIT;
```

5. Move the data from TEMP_NO to OFFICE_NO:

```
EXEC SQL
   UPDATE EMPLOYEE
      SET OFFICE_NO = TEMP_NO;
```

6. Finally, drop the TEMP_NO column:

```
EXEC SQL
   ALTER TABLE DROP TEMP_NO;
EXEC SQL
   COMMIT;
```

For more information about dropping column definitions, see “Dropping an Existing Column From a Table,” in this chapter. For more information about adding column definitions, see “Adding a New Column to a Table,” in this chapter.

Altering a View

To change the information provided by a view, follow these steps:

1. Drop the current view definition.
2. Create a new view definition and give it the same name as the dropped view.

For example, the following view is defined to select employee salary information:

```
EXEC SQL
   CREATE VIEW EMPLOYEE_SALARY AS
      SELECT EMP_NO, LAST_NAME, CURRENCY, SALARY
      FROM EMPLOYEE, COUNTRY
      WHERE EMPLOYEE.COUNTRY_CODE = COUNTRY.CODE;
```

Suppose the full name of each employee should be displayed instead of the last name. First, drop the current view definition:

```
EXEC SQL
   DROP EMPLOYEE_SALARY;
EXEC SQL
   COMMIT;
```

Then create a new view definition that displays each employee's full name:

```
EXEC SQL
   CREATE VIEW EMPLOYEE_SALARY AS
      SELECT EMP_NO, FULL_NAME, CURRENCY, SALARY
      FROM EMPLOYEE, COUNTRY
      WHERE EMPLOYEE.COUNTRY_CODE = COUNTRY.CODE;
EXEC SQL
   COMMIT;
```

For more information about dropping a view, see "Dropping a View," in this chapter. For more information about creating a view, see "Creating a View," in this chapter.

Altering an Index

To change the definition of an index, follow these steps:

1. Use ALTER INDEX to make the current index inactive.
2. Drop the current index.
3. Create a new index and give it the same name as the dropped index.

An index is usually modified to change the combination of columns that are indexed, to prevent or allow insertion of duplicate entries, or to specify index sort order. For example, given the following definition of the NAMEX index:

```
EXEC SQL
   CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Suppose there is an additional need to prevent duplicate entries with the UNIQUE keyword. First, make the current index inactive, then drop it:

```
EXEC SQL
   ALTER INDEX NAMEX INACTIVE;
EXEC SQL
   DROP INDEX NAMEX;
EXEC SQL
   COMMIT;
```

Then create a new index, NAMEX, based on the previous definition, that also includes the UNIQUE keyword:

```
EXEC SQL
   CREATE UNIQUE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
EXEC SQL
   COMMIT
```

ALTER INDEX can be used directly to change an index's sort order, or to add the ability to handle unique or duplicate entries. For example, the following statement changes the NAMEX index to permit duplicate entries:

```
EXEC SQL
   ALTER INDEX NAMEX DUPLICATE;
```

Important Be careful when altering an index directly. For example, changing an index from supporting duplicate entries to one that requires unique entries without disabling the index and recreating it can reduce index performance.

For more information about dropping an index, see "Dropping an Index," in this chapter. For more information about creating an index, see "Creating an Index," in this chapter.

CHAPTER 6

Working With Data

The majority of SQL statements in an embedded program are devoted to reading or modifying existing data, or adding new data to a database. This chapter describes the types of data recognized by InterBase, and how to retrieve, modify, add, or delete data in a database using SQL expressions and the following statements.

- SELECT statements *query* a database, that is, read or retrieve existing data from a database. Variations of the SELECT statement make it possible to retrieve:
  - A single row, or part of a row, from a table. This operation is referred to as a *singleton select*.
  - Multiple rows, or parts of rows, from a table using a SELECT within a DECLARE CURSOR statement.
  - Related rows, or parts of rows, from two or more tables into a *virtual table*, or *results table*. This operation is referred to as a *join*.
  - All rows, or parts of rows, from two or more tables into a virtual table. This operation is referred to as a *union*.
- INSERT statements write new rows of data to a table.
- UPDATE statements modify existing rows of data in a table.
- DELETE statements remove existing rows of data from a table.

To learn how to use the SELECT statement to retrieve data, see “Understanding Data Retrieval With SELECT,” in this chapter. For information about retrieving a single row with SELECT, see “Selecting a Single Row,” in this chapter. For information about retrieving multiple rows, see “Selecting Multiple Rows,” in this chapter.

For information about using INSERT to write new data to a table, see “Inserting Data,” in this chapter. To modify data with UPDATE, see “Updating Data,” in this chapter. To remove data from a table with DELETE, see “Deleting Data,” in this chapter.

Supported Data Types

To query or write to a table, it is necessary to know the structure of the table, what columns it contains, and what data types are defined for those columns. InterBase supports ten fundamental data types, described in the following table:

Table 6-1: Data Types Supported by InterBase

Name	Size	Range/Precision	Description
BLOB	Variable	None. BLOB segment size is limited to 64K.	Binary large object. Stores large data, such as graphics, text, and digitized voice. Basic structural unit: segment. BLOB subtype describes BLOB contents.
CHAR(*n*)	*n* characters	1 to 32,767 bytes. Character set character size determines the maximum number of characters that can fit in 32K.	Fixed length CHAR or text string type. Alternate keyword: CHARACTER.
DATE	64 bits	1 Jan 100 to 11 Dec 5941.	Also includes time information.
DECIMAL (*precision*, *scale*)	Variable	*precision* = 1 to 15. Specifies at least *precision* digits of precision to store. *scale* = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	Number with a decimal point *scale* digits from the right. For example, DECIMAL(10, 3) holds numbers accurately in the following format: ppppppp.sss
DOUBLE PRECISION	64 bits‡	1.7×10^{-308} to 1.7×10^{308}.	Scientific: 15 digits of precision.
FLOAT	32 bits	3.4×10^{-38} to 3.4×10^{38}.	Single precision: 7 digits of precision.
INTEGER	32 bits	-2,147,483,648 to 2,147,483,647.	Signed long (longword).
NUMERIC (*precision*, *scale*)	Variable	*precision* = 1 to 15. Specifies exactly *precision* digits of precision to store. *scale* = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format: ppppppp.sss
SMALLINT	16 bits	-32,768 to 32,767.	Signed short (word).

Table 6-1: Data Types Supported by InterBase (Continued)

Name	Size	Range/Precision	Description
VARCHAR (*n*)	*n* characters	1 to 32,765 bytes. Character set character size determines the maximum number of characters that can fit in 32K.	Variable length CHAR or text string type. Alternate keywords: CHAR VARYING, CHARACTER VARYING.

‡ **Actual size of DOUBLE is platform-dependent. Most platforms support the 64 bit size.**

A BLOB is used to store very large data objects of indeterminate and variable size, such as bitmapped graphics images, vector drawings, sound files, chapter or book-length documents, or any other kind of multimedia information. Because a BLOB can hold different kinds of information, it requires special processing for reading and writing. For more information about BLOB handling, see Chapter 8: "Working With BLOB Data."

The DATE data type may require conversion to and from InterBase when entered or manipulated in a host-language program. For more information about retrieving and writing dates, see Chapter 7: "Working With Dates."

InterBase also supports arrays of most data types. An *array* is a matrix of individual items, all of any single InterBase data type, except BLOB, that can be handled either as a single entity, or manipulated item by item. To learn more about the flexible data access provided by arrays, see Chapter 9: "Using Arrays."

For a complete discussion of InterBase data types, see the *Data Definition Guide*.

Understanding SQL Expressions

All SQL data manipulation statements support *SQL expressions*, SQL syntax for comparing and evaluating columns, constants, and host-language variables to produce a single value.

In the SELECT statement, for example, the WHERE clause is used to specify a *search condition* that determines if a row qualifies for retrieval. That search condition is an SQL expression. DELETE and UPDATE also support search condition expressions. Typically, when an expression is used as a search condition, the expression evaluates to a Boolean value that is True, False, or Unknown.

SQL expressions can also appear in the INSERT statement VALUE clause and the UPDATE statement SET clause to specify or calculate values to insert into a column. When inserting or updating a numeric value via an expression, the expression is usually arithmetic, such as multiplying one number by another to produce a new number which is then inserted or updated in a column. When

inserting or updating a string value, the expression may *concatenate*, or combine, two strings to produce a single string for insertion or updating.

The following table describes the elements that can be used in expressions:

Table 6-2: Elements of SQL Expressions

Element	Description
Column names	Columns from specified tables, against which to search or compare values, or from which to calculate values.
Host-language variables	Program variables containing changeable values. Host-language variables must be preceded by a colon (:).
Constants	Hard-coded numbers or quoted strings, like 507 or “Tokyo”.
Concatenation operator	||, used to combine character strings.
Arithmetic operators	+, -, *, and /, used to calculate and evaluate values.
Logical operators	Keywords, NOT, AND, and OR, used within simple search conditions, or to combine simple search conditions to make complex searches. A logical operation evaluates to true or false. Usually used only in search conditions.
Comparison operators	<, >, <=, >=, =, and <>, used to compare a value on the left side of the operator to another on the right. A comparative operation evaluates to true or false. Other, more specialized comparison operators include ALL, ANY, BETWEEN, CONTAINING, EXISTS, IN, IS, LIKE, NULL, SINGULAR, SOME, and STARTING WITH. These operators can evaluate to True, False, or Unknown. Usually used only in search conditions.
COLLATE clause	Comparisons of CHAR and VARCHAR values can sometimes take advantage of a COLLATE clause to force the way text values are compared.
Stored procedures	Reusable SQL statement blocks that can receive and return parameters, and that are stored as part of a database’s metadata. For more information about stored procedures in queries, see Chapter 12: “Working With Stored Procedures.”
Subqueries	A SELECT statement, usually nested within the WHERE clause to return or calculate values against which rows searched by the main SELECT statement are compared. For more information about subqueries, see “Using Subqueries,” in this chapter.
Parentheses	Group related parts of expressions that should be processed separately to produce a single value which is then used within the expression. Parenthetical expressions can be nested.

Complex expressions can be constructed by combining simple expressions in different ways. For example the following WHERE clause uses a column name, three constants, three comparison operators, and a set of grouping parentheses to retrieve only those rows for employees with salaries between $60,000 and $120,000:

```
WHERE DEPARTMENT = "Publications" AND
   (SALARY > 60000 AND SALRAY < 120000)
```

As another example, search conditions in WHERE clauses often contain nested SELECT statements, or *subqueries*. In the following query, the WHERE clause contains a subquery that uses the aggregate function, AVG(), to retrieve a list of all departments with bigger than average salaries:

```
EXEC SQL
   DECLARE WELL_PAID CURSOR FOR
      SELECT DEPT_NO
         INTO :wellpaid
         FROM DEPARTMENT
      WHERE SALARY > (SELECT AVG(SALARY) FROM DEPARTMENT);
```

For more information about using subqueries to specify search conditions, see "Using Subqueries," in this chapter. For more information about aggregate functions, see "Retrieving Aggregate Column Information," in this chapter.

Using the String Operator in Expressions

The string operator, ||, also referred to as a *concatenation operator*, enables a single character string to be built from two or more character strings. Character strings can be constants or values retrieved from a column. For example,

```
char strbuf[80];
. . .
EXEC SQL
   SELECT LAST_NAME || " is the manager of publications."
      INTO :strbuf
      FROM DEPARTMENT, EMPLOYEE
      WHERE DEPT_NO = 5900 AND MNGR_NO = EMP_NO;
```

The string operator can also be used in INSERT or UPDATE statements:

```
EXEC SQL
   INSERT INTO DEPARTMENT (MANAGER_NAME)
      VALUES(:fname || :lname);
```

Using Arithmetic Operators in Expressions

To calculate numeric values in expressions, InterBase recognizes four arithmetic operators listed in the following table:

Table 6-3: Arithmetic Operators

Operator	Purpose	Precedence	Operator	Purpose	Precedence
*	Multiplication	1	+	Addition	3
/	Division	2	-	Subtraction	4

Arithmetic operators are evaluated from left to right, except when ambiguities arise. In these cases, InterBase evaluates operations according to the precedence specified in the table (for example, multiplications are performed before divisions, and divisions are performed before subtractions).

Arithmetic operations are always calculated before comparison and logical operations. To change or force the order of evaluation, group operations in parentheses. InterBase calculates operations within parentheses first. If parentheses are nested, the equation in the innermost set is the first evaluated, and the outermost set is evaluated last. For more information about precedence and using parentheses for grouping, see "Determining Precedence of Operators," in this chapter.

The following example illustrates a WHERE clause search condition that uses an arithmetic operator to combine the values from two columns, then uses a comparison operator to determine if that value is greater than 10:

```
DECLARE RAINCITIES CURSOR FOR
   SELECT CITYNAME, COUNTRYNAME
      INTO :cityname, :countryname
      FROM CITIES
      WHERE JANUARY_RAIN + FEBRUARY_RAIN > 10;
```

Using Logical Operators in Expressions

Logical operators calculate a Boolean value, True, False, or Unknown, based on comparing previously calculated simple search conditions immediately to the left and right of the operator. InterBase recognizes three logical operators, NOT, AND, and OR.

NOT reverses the search condition in which it appears, while AND and OR are used to combine simple search conditions. For example, the following query returns any employee whose last name is not "Smith":

```
DECLARE NOSMITH CURSOR FOR
```

```
SELECT LAST_NAME
   INTO :lname
   FROM EMPLOYEE
   WHERE NOT LNAME = "Smith";
```

When AND appears between search conditions, both search conditions must be true if a row is to be retrieved. The following query returns any employee whose last name is neither “Smith” nor “Jones”:

```
DECLARE NO_SMITH_OR_JONES CURSOR FOR
   SELECT LAST_NAME
      INTO :lname
      FROM EMPLOYEE
      WHERE NOT LNAME = "Smith" AND NOT LNAME = "Jones";
```

OR stipulates that one search condition or the other must be true. For example, the following query returns any employee named “Smith” or “Jones”:

```
DECLARE ALL_SMITH_JONES CURSOR FOR
   SELECT LAST_NAME, FIRST_NAME
      INTO :lname, :fname
      FROM EMPLOYEE
      WHERE LNAME = "Smith" OR LNAME = "Jones";
```

The order in which combined search conditions are evaluated is dictated by the precedence of the operators that connect them. A NOT condition is evaluated before AND, and AND is evaluated before OR. Parentheses can be used to change the order of evaluation. For more information about precedence and using parentheses for grouping, see “Determining Precedence of Operators,” in this chapter.

Using Comparison Operators in Expressions

Comparison operators evaluate to a Boolean value: True, False, or Unknown, based on a test for a specific relationship between a value to the left of the operator, and a value or range of values to the right of the operator. Values compared must evaluate to the same data type, unless the CAST() function is used to translate one data type to a different one for comparison. Values can be columns, constants, or calculated values.

The following table lists operators that can be used in statements, describes how they are used, and provides samples of their use:

Note Comparisons evaluate to Unknown if a NULL value is encountered.

For more information about CAST(), see “Using CAST( ) for Data Type Conversions,” in this chapter.

InterBase also supports comparison operators that compare a value on the left of the operator to the results of a subquery to the right of the operator. The following table lists these operators, and describes how they are used:

Table 6-4: InterBase Comparison Operators Requiring Subqueries

Operator	Purpose
ALL	Determines if a value is equal to all values returned by a subquery.
ANY and SOME	Determines if a value is equal to any values returned by a subquery.
EXISTS	Determines if a value exists in *at least one* value returned by a subquery.
SINGULAR	Determines if a value exists in *exactly one* value returned by a subquery.

For more information about using subqueries, see "Using Subqueries," in this chapter.

Using BETWEEN

BETWEEN tests whether a value falls within a range of values. The complete syntax for the BETWEEN operator is:

```
<value> [NOT] BETWEEN <value> AND <value>
```

For example, the following cursor declaration retrieves LAST_NAME and FIRST_NAME columns for employees with salaries between $100,000 and $250,000, inclusive:

```
EXEC SQL
   DECLARE LARGE_SALARIES CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE SALARY BETWEEN 100000 AND 250000;
```

Use NOT BETWEEN to test whether a value falls outside a range of values. For example, the following cursor declaration retrieves the names of employees with salaries less than $30,000 and greater than $150,000:

```
EXEC SQL
   DECLARE EXTREME_SALARIES CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE SALARY NOT BETWEEN 30000 AND 150000;
```

Using CONTAINING

CONTAINING tests to see if an ASCII string value contains a quoted ASCII string supplied by the program. String comparisons are case-insensitive; “String”, “STRING”, and “string” are equivalent values for CONTAINING. The complete syntax for CONTAINING is:

```
<value> [NOT] CONTAINING "<string>"
```

For example, the following cursor declaration retrieves the names of all employees whose last names contain the three-letter combination, “las” (and “LAS” or “Las”):

```
EXEC SQL
   DECLARE LAS_EMP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME CONTAINING "las";
```

Use NOT CONTAINING to test for strings that exclude a specified value. For example, the following cursor declaration retrieves the names of all employees whose last names do not contain “las” (also “LAS” or “Las”):

```
EXEC SQL
   DECLARE NOT_LAS_EMP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME NOT CONTAINING "las";
```

Tip CONTAINING can be used to search a BLOB segment by segment for an occurrence of a quoted string.

Using IN

IN tests that a known value equals at least one value in a list of values. A list is a set of values separated by commas and enclosed by parentheses. The values in the list must be parenthesized and separated by commas. If the value being compared to a list of values is NULL, IN returns Unknown.

The syntax for IN is:

```
<value> [NOT] IN (<value> [, <value> ...])
```

For example, the following cursor declaration retrieves the names of all employees in the accounting, payroll, and human resources departments:

```
EXEC SQL
   DECLARE ACCT_PAY_HR CURSOR FOR
      SELECT DEPARTMENT, LAST_NAME, FIRST_NAME, EMP_NO
```

```
            FROM EMPLOYEE EMP, DEPTARTMENT DEP
            WHERE EMP.DEPT_NO = DEP.DEPT_NO AND
               DEPARTMENT IN ("Accounting", "Payroll", "Human Resources")
            GROUP BY DEPARTMENT;
```

Use NOT IN to test that a value does not occur in a set of specified values. For example, the following cursor declaration retrieves the names of all employees not in the accounting, payroll, and human resources departments:

```
EXEC SQL
   DECLARE NOT_ACCT_PAY_HR CURSOR FOR
      SELECT DEPARTMENT, LAST_NAME, FIRST_NAME, EMP_NO
         FROM EMPLOYEE EMP, DEPTARTMENT DEP
         WHERE EMP.DEPT_NO = DEP.DEPT_NO AND
            DEPARTMENT NOT IN ("Accounting", "Payroll",
                "Human Resources")
         GROUP BY DEPARTMENT;
```

IN can also be used to compare a value against the results of a subquery. For example, the following cursor declaration retrieves all cities in Europe:

```
EXEC SQL
   DECLARE NON_JFG_CITIES CURSOR FOR
      SELECT C.COUNTRY, C.CITY, C.POPULATION
         FROM CITIES C
         WHERE C.COUNTRY NOT IN (SELECT O.COUNTRY FROM COUNTRIES O
            WHERE O.CONTINENT <> "Europe")
         GROUP BY C.COUNTRY;
```

For more information about subqueries, see “Using Subqueries,” in this chapter.

Using LIKE

LIKE is a case-sensitive operator that tests a string value against a string containing *wildcards*, symbols that substitute for a single, variable character, or a string of variable characters. LIKE recognizes two wildcard symbols:

- % (percent) substitutes for a string of zero or more characters.
- _ (underscore) substitutes for a single character.

The syntax for LIKE is:

```
<value> [NOT] LIKE <value> [ESCAPE "symbol"]
```

For example, this cursor retrieves information about any employee whose last names contain the three letter combination “ton” (but not “Ton”):

```
EXEC SQL
   DECLARE TON_EMP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, EMP_NO
```

```
        FROM EMPLOYEE
        WHERE LAST_NAME LIKE "%ton%";
```

To test for a string that contains a percent or underscore character:

1. Precede the % or _ with another symbol (for example, @), in the quoted comparison string.
2. Use the ESCAPE clause to identify the symbol (@, in this case) preceding % or _ as a literal symbol. A *literal symbol* tells InterBase that the next character should be included as is in the search string.

For example, this cursor retrieves all table names in RDB$RELATIONS that have underscores in their names:

```
EXEC SQL
   DECLARE UNDER_TABLE CURSOR FOR
      SELECT RDB$RELATION_NAME
         FROM RDB$RELATIONS
         WHERE RDB$RELATION_NAME LIKE "%@_%" ESCAPE "@";
```

Use NOT LIKE to retrieve rows that do not contain strings matching those described. For example, the following cursor retrieves all table names in RDB$RELATIONS that do not have underscores in their names:

```
EXEC SQL
   DECLARE NOT_UNDER_TABLE CURSOR FOR
      SELECT RDB$RELATION_NAME
         FROM RDB$RELATIONS
         WHERE RDB$RELATION_NAME NOT LIKE "%@_%" ESCAPE "@";
```

Using IS NULL

IS NULL tests for the absence of a value in a column. The complete syntax of the IS NULL clause is:

```
<value> IS [NOT] NULL
```

For example, the following cursor retrieves the names of employees who do not have phone extensions:

```
EXEC SQL
   DECLARE MISSING_PHONE CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE PHONE_EXT IS NULL;
```

Use IS NOT NULL to test that a column contains a value. For example, the following cursor retrieves the phone numbers of all employees that have phone extensions:

```
EXEC SQL
   DECLARE PHONE_LIST CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
         FROM EMPLOYEE
         WHERE PHONE_EXT IS NOT NULL
         ORDER BY LAST_NAME, FIRST_NAME;
```

Using STARTING WITH

STARTING WITH is a case-sensitive operator that tests a string value to see if it begins with a stipulated string of characters. To support international character set conversions, STARTING WITH follows byte-matching rules for the specified collation order. The complete syntax for STARTING WITH is:

```
<value> [NOT] STARTING WITH <value>
```

For example, the following cursor retrieves employee last names that start with "To":

```
EXEC SQL
   DECLARE TO_EMP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME STARTING WITH "To";
```

Use NOT STARTING WITH to retrieve information for columns that do not begin with the stipulated string. For example, the following cursor retrieves all employees except those whose last names start with "To":

```
EXEC SQL
   DECLARE NOT_TO_EMP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME NOT STARTING WITH "To";
```

For more information about collation order and byte-matching rules, see the *Data Definition Guide*.

Using ALL

ALL tests that a value is true when compared to every value in a list returned by a subquery. The complete syntax for ALL is:

```
<value> <comparison_operator> ALL (<subquery>)
```

For example, the following cursor retrieves information about employees whose salaries are larger than that of the vice president of channel marketing:

```
EXEC SQL
   DECLARE MORE_THAN_VP CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, SALARY
         FROM EMPLOYEE
         WHERE SALARY > ALL (SELECT SALARY FROM EMPLOYEE
            WHERE DEPT_NO = 7734);
```

ALL returns Unknown if the subquery returns a NULL value. It can also return Unknown if the value to be compared is NULL and the subquery returns any non-NULL data. If the value is NULL and the subquery returns an empty set, ALL evaluates to True.

For more information about subqueries, see "Using Subqueries," in this chapter.

Using ANY and SOME

ANY and SOME test that a value is true if it matches any value in a list returned by a subquery. The complete syntax for ANY is:

```
<value> <comparison_operator> ANY | SOME (<subquery>)
```

For example, the following cursor retrieves information about salaries that are larger than at least one salary in the channel marketing department:

```
EXEC SQL
   DECLARE MORE_CHANNEL CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, SALARY
         FROM EMPLOYEE
         WHERE SALARY > ANY (SELECT SALARY FROM EMPLOYEE
            WHERE DEPT_NO = 7734);
```

ANY and SOME return Unknown if the subquery returns a NULL value. They can also return Unknown if the value to be compared is NULL and the subquery returns any non-NULL data. If the value is NULL and the subquery returns an empty set, ANY and SOME evaluate to False.

For more information about subqueries, see "Using Subqueries," in this chapter.

Using EXISTS

EXISTS tests that for a given value there is *at least one* qualifying row meeting the search condition specified in a subquery. The SELECT clause in the subquery must use the * (asterisk) to select all columns. The complete syntax for EXISTS is:

```
[NOT] EXISTS (SELECT * FROM <tablelist> WHERE <search_condition>)
```

For example, the following cursor retrieves all countries with rivers:

```
EXEC SQL
   DECLARE RIVER_COUNTRIES CURSOR FOR
      SELECT COUNTRY
         FROM COUNTRIES C
         WHERE EXISTS (SELECT * FROM RIVERS R
            WHERE R.COUNTRY = C.COUNTRY);
```

Use NOT EXISTS to retrieve rows that do not meet the qualifying condition specified in the subquery. For example, the following cursor retrieves all countries without rivers:

```
EXEC SQL
   DECLARE NON_RIVER_COUNTRIES COUNTRIES FOR
      SELECT COUNTRY
         FROM COUNTRIES C
         WHERE NOT EXISTS (SELECT * FROM RIVERS R
            WHERE R.COUNTRY = C.COUNTRY);
```

EXISTS always returns either True or False, even when handling NULL values.

For more information about subqueries, see "Using Subqueries," in this chapter.

Using SINGULAR

SINGULAR tests that for a given value there is *exactly one* qualifying row meeting the search condition specified in a subquery. The SELECT clause in the subquery must use the * (asterisk) to select all columns. The complete syntax for SINGULAR is:

```
[NOT] SINGULAR (SELECT * FROM <tablelist> WHERE <search_condition>)
```

For example, the following cursor retrieves all countries with a single capital:

```
EXEC SQL
   DECLARE SINGLE_CAPITAL CURSOR FOR
      SELECT COUNTRY
         FROM COUNTRIES COU
         WHERE SINGULAR (SELECT * FROM CITIES CIT
            WHERE CIT.CITY = COU.CAPITAL);
```

Use NOT SINGULAR to retrieve rows that do not meet the qualifying condition specified in the subquery. For example, the following cursor retrieves all countries with more than one capital:

```
EXEC SQL
   DECLARE MULTI_CAPITAL CURSOR FOR
      SELECT COUNTRY
         FROM COUNTRIES COU
         WHERE NOT SINGULAR (SELECT * FROM CITIES CIT
            WHERE CIT.CITY = COU.CAPITAL);
```

For more information about subqueries, see “Using Subqueries,” in this chapter.

Determining Precedence of Operators

The order in which operators and the values they affect are evaluated in a statement is called *precedence*. There are two levels of precedence for SQL operators:

- Precedence among operators of different types.
- Precedence among operators of the same type.

Precedence Among Operators of Different Types

The following table lists the evaluation order of different InterBase operator types, from first evaluated (highest precedence) to last evaluated (lowest precedence):

Table 6-5: Operator Precedence By Operator Type

Operator Type	Precedence	Explanation
String	Highest	Strings are always concatenated before all other operations take place.
Mathematical	⇓	Math is performed after string concatenation, but before comparison and logical operations.
Comparison	⇓	Comparison operations are evaluated after string concatenation and math, but before logical operations.
Logical	Lowest	Logical operations are evaluated after all other operations.

Precedence Among Operators of the Same Type

When an expression contains several operators of the same type, those operators are evaluated from left to right unless there is a conflict where two operators of the same type affect the same values.

For example, in the mathematical equation, 3 + 2 * 6, both the addition and multiplication operators work with the same value, 2. Evaluated from left to right, the equation evaluates to 30: 3+ 2 = 5; 5 * 6 = 30. InterBase follows standard mathematical rules for evaluating mathematical expressions, that stipulate multiplication is performed before addition: 2 *6 = 12; 3 + 12 = 15.

The following table lists the evaluation order for all mathematical operators, from highest to lowest:

Table 6-6: Mathematical Operator Precedence

Operator	Precedence	Explanation
*	Highest	Multiplication is performed before all other mathematical operations.
/	⇓	Division is performed before addition and subtraction.
+	⇓	Addition is performed before subtraction.
-	Lowest	Subtraction is performed after all other mathematical operations.

InterBase also follows rules for determining the order in which comparison operators are evaluated when conflicts arise during normal left to right evaluation. The next table describes the evaluation order for comparison operators, from highest to lowest:

Table 6-7: Comparison Operator Precedence

Operator	Precedence	Explanation
=, ==	Highest	Equality operations are evaluated before all other comparison operations.
<>, !=, ~=, ^=	⇓	
>	⇓	
<	⇓	
>=	⇓	
<=	⇓	
!>, ~>, ^>	⇓	
!<, ~<, ^<	Lowest	Not less than operations are evaluated after all other comparison operations.

ALL, ANY, BETWEEN, CONTAINING, EXISTS, IN, LIKE, NULL, SINGULAR, SOME, and STARTING WITH are evaluated after all listed comparison operators when they conflict with other comparison operators during normal left to right evaluation. When they conflict with one another they are evaluated strictly from left to right.

When logical operators conflict during normal left to right processing, they, too, are evaluated according to a hierarchy, detailed in the following table:

Table 6-8: Logical Operator Precedence

Operator	Precedence	Explanation
NOT	Highest	NOT operations are evaluated before all other logical operations.
AND	⇓	AND operations are evaluated after NOT operations, and before OR operations.
OR	Lowest	OR operations are evaluated after all other logical operations.

Changing Evaluation Order of Operators

To change the evaluation order of operations in an expression, use parentheses to group operations that should be evaluated as a unit, or that should derive a single value for use in other operations. For example, without parenthetical grouping, 3 + 2 * 6 evaluates to 15. To cause the addition to be performed before the multiplication, use parentheses:

```
(3 + 2) * 6 = 30
```

Tip Always use parentheses to group operations in complex expressions, even when default order of evaluation is desired. Explicitly grouped expressions are easier to understand and debug.

Using CAST( ) for Data Type Conversions

Normally, only similar data types can be compared or evaluated in expressions. The CAST() function can be used in expressions to translate one data type into another for comparison purposes. The syntax for CAST() is:

```
CAST (<value> | NULL AS datatype)
```

For example, in the following WHERE clause, CAST() is used to translate a CHAR data type, INTERVIEW_DATE, to a DATE data type to compare against a DATE data type, HIRE_DATE:

```
. . . WHERE HIRE_DATE = CAST(INTERVIEW_DATE AS DATE);
```

CAST() can be used to compare columns with different data types in the same table, or across tables. You can convert one data type to another as shown in the following table:

Table 6-9: Compatible Data Types for CAST()

From Data Type	To Data Type
NUMERIC	CHARACTER, DATE
CHARACTER	NUMERIC, DATE
DATE	CHARACTER, NUMERIC

An error results if a given data type cannot be converted into the data type specified in CAST().

Using UPPER( ) on Text Data

The UPPER() function can be used in SELECT, INSERT, UPDATE, or DELETE operations to force character and BLOB text data to uppercase. For example, an application that prompts a user for a department name might want to ensure that all department names are stored in uppercase to simplify data retrieval later. The following code illustrates how UPPER() would be used in the INSERT statement to guarantee a user's entry is uppercase:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      char response[26];
EXEC SQL
   END DECLARE SECTION;
. . .
printf("Enter new department name: ");
response[0] = '\0';
gets(response);
if (response)
   EXEC SQL
      INSERT INTO DEPARTMENT(DEPT_NO, DEPARTMENT)
         VALUES(GEN_ID(GDEPT_NO, 1), UPPER(:response));
. . .
```

The next statement illustrates how UPPER() can be used in a SELECT statement to affect both the appearance of values retrieved, and to affect its search condition:

```
EXEC SQL
   SELECT DEPT_NO, UPPER(DEPARTMENT)
      FROM DEPARTMENT
      WHERE UPPER(DEPARTMENT) STARTING WITH 'A';
```

Understanding Data Retrieval With SELECT

The SELECT statement handles all queries in SQL. SELECT can retrieve one or more rows from a table, and can return entire rows, or a subset of columns from each row, often referred to as a *projection*. Optional SELECT syntax can be used to specify search criteria that restrict the number of rows returned, to select rows with unknown values, to select rows through a view, and to combine rows from two or more tables.

At a minimum, every SELECT statement must:

- List which columns to retrieve from a table. The column list immediately follows the SELECT keyword.
- Name the table to search in a FROM clause.

Singleton selects must also include both an INTO clause to specify the host variables into which retrieved values should be stored, and a WHERE clause to specify the search conditions that cause only a single row to be returned.

The following SELECT retrieves three columns from a table and stores the values in three host-language variables:

```
EXEC SQL
   SELECT EMP_NO, FIRSTNAME, LASTNAME
      INTO :emp_no, :fname, :lname
      FROM EMPLOYEE WHERE EMP_NO = 1888;
```

Important Host variables must be declared in a program before they can be used in SQL statements. For more information about declaring host variables, see Chapter 2: "Application Requirements."

The following table lists all SELECT statement clauses, in the order that they are used, and prescribes their use in singleton and multi-row selects:

Table 6-10: SELECT Statement Clauses

Clause	Purpose	Singleton SELECT	Multi-row SELECT
SELECT	Lists columns to retrieve.	Required	Required
INTO	Lists host variables for storing retrieved columns.	Required	Not allowed
FROM	Identifies the tables to search for values.	Required	Required
WHERE	Specifies the search conditions used to restrict retrieved rows to a subset of all available rows. A WHERE clause can contain its own SELECT statement, referred to as a *subquery*.	Optional	Optional

Table 6-10: SELECT Statement Clauses (Continued)

Clause	Purpose	Singleton SELECT	Multi-row SELECT
GROUP BY	Groups related rows based on common column values. Used in conjunction with HAVING.	Optional	Optional
HAVING	Restricts rows generated by GROUP BY to a subset of those rows.	Optional	Optional
UNION	Combines the results of two or more SELECT statements to produce a single, dynamic table without duplicate rows.	Optional	Optional
PLAN	Specifies the query plan that should be used by the query optimizer instead of one it would normally choose.	Optional	Optional
ORDER BY	Specifies the sort order of rows returned by a SELECT, either ascending (ASC), the default, or descending (DESC).	Optional	Optional
FOR UPDATE	Specifies columns listed after the SELECT clause of a DECLARE CURSOR statement that can be updated using a WHERE CURRENT OF clause.	Not allowed	Optional

Using each of these clauses with SELECT is described in the following sections, after which using SELECT directly to return a single row, and using SELECT within a DECLARE CURSOR statement to return multiple rows are described in detail. For a complete overview of SELECT syntax, see the *Language Reference*.

Listing Columns to Retrieve With SELECT

A list of columns to retrieve must always follow the SELECT keyword in a SELECT statement. The SELECT keyword and its column list is called a *SELECT clause*.

Retrieving a List of Columns

To retrieve a subset of columns for a row of data, list each column by name, in the order of desired retrieval, and separate each column name from the next by a comma. Operations that retrieve a subset of columns are often called *projections*.

For example, the following SELECT retrieves three columns:

```
EXEC SQL
   SELECT EMP_NO, FIRSTNAME, LASTNAME
      INTO :emp_no, :fname, :lname
      FROM EMPLOYEE WHERE EMP_NO = 2220;
```

Retrieving All Columns

To retrieve all columns of data, use an asterisk (*) instead of listing any columns by name. For example, the following SELECT retrieves every column of data for a single row in the EMPLOYEE table:

```
EXEC SQL
   SELECT *
      INTO :emp_no, :fname, :lname, :phone_ext, :hire, :dept_no,
         :job_code, :job_grade, :job_country, :salary, :full_name
      FROM EMPLOEE WHERE EMP_NO = 1888;
```

Important One host variable must be provided for each column returned by a query.

Eliminating Duplicate Columns With DISTINCT

In a query returning multiple rows, it may be desirable to eliminate duplicate columns. For example, the following query, meant to determine if the EMPLOYEE table contains employees with the last name, SMITH, might locate many such rows:

```
EXEC SQL
   DECLARE SMITH CURSOR FOR
      SELECT LAST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME = "Smith";
```

To eliminate duplicate columns in such a query, use the DISTINCT keyword with SELECT. For example, the following SELECT yields only a single instance of “Smith”:

```
EXEC SQL
   DECLARE SMITH CURSOR FOR
      SELECT DISTINCT LAST_NAME
         FROM EMPLOYEE
         WHERE LAST_NAME = "Smith";
```

DISTINCT affects all columns listed in a SELECT statement.

Retrieving Aggregate Column Information

SELECT can include *aggregate functions*, functions that calculate or retrieve a single, collective numeric value for a column or expression based on each qualify-

ing row in a query rather than retrieving each value separately. The following table lists the aggregate functions supported by InterBase:

Table 6-11: Aggregate Functions in SQL

Function	Purpose
AVG()	Calculates the average numeric value for a set of values.
MIN()	Retrieves the minimum value in a set of values.
MAX()	Retrieves the maximum value in a set of values.
SUM()	Calculates the total of numeric values in a set of values.
COUNT()	Calculates the number of rows that satisfy the query's search condition (specified in the WHERE clause).

For example, the following query returns the average salary for all employees in the EMPLOYEE table:

```
EXEC SQL
   SELECT AVG(SALARY)
      INTO :avg_sal
      FROM EMPLOYEE;
```

The following SELECT returns the number of qualifying rows it encounters in the EMPLOYEE table, both the maximum and minimum employee number of employees in the table, and the total salary of all employees in the table:

```
EXEC SQL
   SELECT COUNT(*), MAX(EMP_NO), MIN(EMP_NO), SUM(SALARY)
      INTO :counter, :maxno, :minno, :total_salary
      FROM EMPLOYEE;
```

If a field value involved in an aggregate calculation is NULL or unknown, the entire row is automatically excluded from the calculation. Automatic exclusion prevents averages from being skewed by meaningless data.

Note Aggregate functions can also be used to calculate values for groups of rows. The resulting value is called a *group aggregate*. For more information about using group aggregates, see "Grouping Rows With GROUP BY," in this chapter.

Qualifying Column Names in Multi-table SELECT Statements

When data is retrieved from multiple tables, views, and select procedures, the same column name may appear in more than one table. In these cases, the SELECT statement must contain enough information to distinguish like-named columns from one another.

To distinguish column names in multiple tables, precede those columns with one of the following qualifiers in the SELECT clause:

- The name of the table, followed by a period. For example, EMPLOYEE.EMP_NO identifies a column named EMP_NO in the EMPLOYEE table.
- A table correlation name (alias) followed by a period. For example, if the correlation name for the EMPLOYEE table is EMP, then EMP.EMP_NO identifies a column named EMP_NO in the EMPLOYEES table.

Correlation names can be declared for tables, views, and select procedures in the FROM clause of the SELECT statement. For more information about declaring correlation names, and for examples of their use, see "Declaring and Using Correlation Names," in this chapter.

Specifying Transaction Names in a SELECT

InterBase enables an SQL application to run many simultaneous transactions if:

- Each transaction is first named with a SET TRANSACTION statement.
- Each data manipulation statement (SELECT, INSERT, UPDATE, DELETE) specifies a TRANSACTION clause that identifies the name of the transaction under which it operates.
- SQL statements are not dynamic.

In SELECT, the TRANSACTION clause intervenes between the SELECT keyword and the column list, as in the following syntax fragment:

```
SELECT TRANSACTION name <col> [, <col> ...]
```

The TRANSACTION clause is optional in single-transaction programs or in programs where only one transaction is open at a time. It must be used in a multi-transaction program. For example, the following SELECT is controlled by the transaction, T1:

```
EXEC SQL
   SELECT TRANSACTION T1:
      COUNT(*), MAX(EMP_NO), MIN(EMP_NO), SUM(SALARY)
         INTO :counter, :maxno, :minno, :total_salary
         FROM EMPLOYEE;
```

For a complete discussion of transaction handling and naming, see Chapter 4: "Working With Transactions."

Specifying Host Variables With INTO

A singleton select returns data to a list of host-language variables specified by an INTO clause in the SELECT statement. The INTO clause immediately follows the list of table columns from which data is to be extracted. Each host variable in the list must be preceded by a colon (:) and separated from the next by a comma.

The host-language variables in the INTO clause must already have been declared before they can be used. The number, order, and data type of host-language variables must correspond to the number, order, and data type of the columns retrieved. Otherwise, overflow or data conversion errors may occur.

For example, the following C program fragment declares three host variables, *lname*, *fname*, and *salary*. Two, *lname*, and *fname*, are declared as character arrays; *salary* is declared as a long integer. The SELECT statement specifies that three columns of data are to be retrieved, while the INTO clause specifies the host variables into which the data should be read.

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
long salary;
char lname[20], fname[15];
EXEC SQL
   END DECLARE SECTION;
. . .
   EXEC SQL
      SELECT LAST_NAME, FIRST_NAME, SALARY
         INTO :lanem, :fname, :salary
         FROM EMPLOYEE
         WHERE LNAME = "Smith";
. . .
```

Note In a multi-row select, the INTO clause is part of the FETCH statement, *not* the SELECT statement. For more information about the INTO clause in FETCH, see "Fetching Rows With a Cursor," in this chapter.

Listing Tables to Search With FROM

The FROM clause is required in a SELECT statement. It identifies the tables, views, or select procedures from which data is to be retrieved. The complete syntax of the FROM clause is:

```
FROM table | view | procedure [alias] [, table | view | procedure
   [alias] ...]
```

There must be at least one table, view, or select procedure name following the FROM keyword. When retrieving data from multiple sources, each source must be listed, assigned an alias, and separated from the next with a comma. For more information about select procedures, see Chapter 12: “Working With Stored Procedures.”

Listing a Single Table or View

The FROM clause in the following SELECT specifies a single table, EMPLOYEE, from which to retrieve data:

```
EXEC SQL
   SELECT LAST_NAME, FIRST_NAME, SALARY
      INTO :lanem, :fname, :salary
      FROM EMPLOYEE
      WHERE LNAME = "Smith";
```

Use the same INTO clause syntax to specify a view or select procedure as the source for data retrieval instead of a table. For example, the following SELECT specifies a select procedure, MVIEW, from which to retrieve data. MVIEW returns information for all managers whose last names begin with the letter “M,” and the WHERE clause narrows the rows returned to a single row where the DEPT_NO column is 430:

```
EXEC SQL
   SELECT DEPT_NO, LAST_NAME, FIRST_NAME, SALARY
      INTO :lname, :fname, :salary
      FROM MVIEW
      WHERE DEPT_NO = 430;
```

For more information about select procedures, see Chapter 12: “Working With Stored Procedures.”

Listing Multiple Tables

To retrieve data from multiple tables, views, or select procedures, include all sources in the FROM clause, separating sources from one another by commas.

There are two different possibilities to consider when working with multiple data sources:

1. The name of each referenced column is unique across all tables.
2. The names of one or more referenced columns exist in two or more tables.

In the first case, just use the column names themselves to reference the columns. For example, the following query returns data from two tables, DEPARTMENT, and EMPLOYEE:

```
EXEC SQL
   SELECT DEPARTMENT, DEPT_NO, LAST_NAME, FIRST_NAME, EMP_NO
      INTO :dept_name, :dept_no, :lname, :fname, :empno
      FROM DEPARTMENT, EMPLOYEE
      WHERE DEPT_NO = "Publications" AND MNGR_NO = EMP_NO;
```

In the second case, column names that occur in two or more tables must be distinguished from one another by preceding each column name with its table name and a period in the SELECT clause. For example, if an EMP_NO column exists in both the DEPARTMENT and EMPLOYEE then the previous query must be recast as follows:

```
EXEC SQL
   SELECT DEPARTMENT, DEPT_NO, LAST_NAME, FIRST_NAME,
      EMLOYEE.EMP_NO
      INTO :dept_name, :dept_no, :lname, :fname, :empno
      FROM DEPARTMENT, EMPLOYEE
      WHERE DEPT_NO = "Publications" AND
         DEPARTMENT.EMP_NO = EMPLOYEE.EMP_NO;
```

For more information about the SELECT clause, see “Listing Columns to Retrieve With SELECT,” in this chapter.

Important For queries involving joins, column names can be qualified by correlation names, brief alternate names, or aliases, that are assigned to each table in a FROM clause and substituted for them in other SELECT statement clauses when qualifying column names. Even when joins are not involved, assigning and using correlation names can reduce the length of complex queries.

Declaring and Using Correlation Names

A *correlation name*, or *alias*, is a temporary variable that represents a table name. It can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_), but must always start with an alphabetic character. Using brief correlation names reduces typing of long queries. Correlation names must be substituted for actual table names in joins, and can be substituted for them in complex queries.

A correlation name is associated with a table in the FROM clause; it replaces table names to qualify column names everywhere else in the statement. For example, to associate the correlation name, DEPT with the DEPARTMENT table, and EMP, with the EMPLOYEES table, a FROM clause might appear as:

```
FROM DEPARTMENT DEPT, EMPLOYEE EMP
```

Like an actual table name, a correlation name is used to qualify column names wherever they appear in a SELECT statement. For example, the following query employs the correlation names, DEPT, and EMP, previously described:

```
EXEC SQL
   SELECT DEPARTMENT, DEPT_NO, LAST_NAME, FIRST_NAME,
      EMLOYEE.EMP_NO
      INTO :dept_name, :dept_no, :lname, :fname, :empno
      FROM DEPARTMENT DEPT, EMPLOYEE EMP
      WHERE DEPT_NO = "Publications" AND DEPT.EMP_NO = EMP.EMP_NO;
```

For more information about the SELECT clause, see “Listing Columns to Retrieve With SELECT,” in this chapter.

Restricting Row Retrieval With WHERE

In a query, the WHERE clause specifies the data a row must (or must not) contain to be retrieved.

In singleton selects, where a query must only return one row, WHERE is mandatory unless a select procedure specified in the FROM clause returns only one row itself.

In SELECT statements within DECLARE CURSOR statements, the WHERE clause is optional. If the WHERE clause is omitted, a query returns all rows in the table. To retrieve a subset of rows in a table, a cursor declaration must include a WHERE clause.

The simple syntax for WHERE is:

```
WHERE <search_condition>
```

For example, the following simple WHERE clause tests a row to see if the DEPARTMENT column is “Publications”:

```
WHERE DEPARTMENT = "Publications"
```

What is a Search Condition?

Because the WHERE clause specifies the type of data a query is searching for it is often called a *search condition*. A query examines each row in a table to see if it meets the criteria specified in the search condition. If it does, the row qualifies for retrieval.

When a row is compared to a search condition, one of three values is returned:

- *True*: A row meets the conditions specified in the WHERE clause.

- *False*: A row fails to meet the conditions specified in the WHERE clause.
- *Unknown*: A column tested in the WHERE clause contains an unknown value that could not be evaluated because of a NULL comparison.

Most search conditions, no matter how complex, evaluate to True or False. An expression that evaluates to True or False—like the search condition in the WHERE clause—is called a *Boolean expression*.

Structure of a Search Condition

A typical simple search condition compares a value in one column against a constant or a value in another column. For example, the following WHERE clause tests a row to see if a field equals a hard-coded constant:

```
WHERE DEPARTMENT = "Publications"
```

This search condition has three elements: a column name, a comparison operator (the equal sign), and a constant. Most search conditions are more complex than this. They involve additional elements and combinations of simple search conditions. The following table describes expression elements that can be used in search conditions:

Table 6-12: Elements of WHERE Clause SEARCH Conditions

Element	Description
Column names	Columns from tables listed in the FROM clause, against which to search or compare values.
Host-language variables	Program variables containing changeable values. When used in a SELECT, host-language variables must be preceded by a colon (:).
Constants	Hard-coded numbers or quoted strings, like 507 or “Tokyo”.
Concatenation operators	||, used to combine character strings.
Arithmetic operators	+, -, *, and /, used to calculate and evaluate search condition values.
Logical operators	Keywords, NOT, AND, and OR, used within simple search conditions, or to combine simple search conditions to make complex searches. A logical operation evaluates to true or false.

Table 6-12: Elements of WHERE Clause SEARCH Conditions (Continued)

Element	Description
Comparison operators	<, >, <=, >=, =, and <>, used to compare a value on the left side of the operator to another on the right. A comparative operation evaluates to True or False. Other, more specialized comparison operators include ALL, ANY, BETWEEN, CONTAINING, EXISTS, IN, IS, LIKE, NULL, SINGULAR, SOME, and STARTING WITH. These operators can evaluate to True, False, or Unknown.
COLLATE clause	Comparisons of CHAR and VARCHAR values can sometimes take advantage of a COLLATE clause to force the way text values are compared.
Stored procedures	Reusable SQL statement blocks that can receive and return parameters, and that are stored as part of a database's metadata. For more information about stored procedures in queries, see Chapter 12: "Working With Stored Procedures."
Subqueries	A SELECT statement nested within the WHERE clause to return or calculate values against which rows searched by the main SELECT statement are compared. For more information about subqueries, see "Using Subqueries," in this chapter.
Parentheses	Group related parts of search conditions which should be processed separately to produce a single value which is then used to evaluate the search condition. Parenthetical expressions can be nested.

Complex search conditions can be constructed by combining simple search conditions in different ways. For example, the following WHERE clause uses a column name, three constants, three comparison operators, and a set of grouping parentheses to retrieve only those rows for employees with salaries between $60,000 and $120,000:

```
WHERE DEPARTMENT = "Publications" AND
   (SALARY > 60000 AND SALARY < 120000)
```

Search conditions in WHERE clauses often contain nested SELECT statements, or *subqueries*. For example, in the following query, the WHERE clause contains a subquery that uses the aggregate function, AVG(), to retrieve a list of all departments with bigger-than-average salaries:

```
EXEC SQL
   DECLARE WELL_PAID CURSOR FOR
      SELECT DEPT_NO
         INTO :wellpaid
         FROM DEPARTMENT
      WHERE SALARY > (SELECT AVG(SALARY) FROM DEPARTMENT);
```

For a general discussion of building search conditions from SQL expressions, see "Understanding SQL Expressions," in this chapter. For more information about using subqueries to specify search conditions, see "Using Subqueries," in this chapter. For more information about aggregate functions, see "Retrieving Aggregate Column Information," in this chapter.

Specifying Collation Order in a Comparison Operation

When CHAR or VARCHAR values are compared in a WHERE clause, it can be necessary to specify a collation order for the comparisons if the values being compared use different collation orders.

To specify the collation order to use for a value during a comparison, include a COLLATE clause after the value. For example, in the following WHERE clause fragment from an embedded application, the value to the left of the comparison operator is forced to be compared using a specific collation:

```
WHERE LNAME COLLATE FR_CA = :lname_search;
```

For more information about collation order and a list of collations available to InterBase, see the *Data Definition Guide*.

Sorting Rows With ORDER BY

By default, a query retrieves rows in the exact order it finds them in a table, and because internal table storage is unordered, retrieval, too, is likely to be unordered. To specify the order in which rows are returned by a query, use the optional ORDER BY clause at the end of a SELECT statement.

ORDER BY retrieves rows based on a column list. Every column in the ORDER BY clause must also appear somewhere in the SELECT clause at the start of the statement. Each column can optionally be ordered in ascending order (ASC, the default), or descending order (DESC). The complete syntax of ORDER BY is:

```
ORDER BY col [COLLATE collation] [ASC | DESC]
   [,col [COLLATE collation] [ASC | DESC] ...];
```

For example, the following cursor declaration orders output based on the LAST_NAME column. Because DESC is specified in the ORDER BY clause, employees are retrieved from Z to A:

```
EXEC SQL
   DECLARE PHONE_LIST CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
         FROM EMPLOYEE
         WHERE PHONE_EXT IS NOT NULL
         ORDER BY LAST_NAME DESC, FIRST_NAME;
```

If more than one column is specified in an ORDER BY clause, rows are first arranged by the values in the first column. Then rows that contain the same first-column value are arranged according to the values in the second column, and so on. Each ORDER BY column can include its own sort order specification.

Important In multi-column sorts, after a sort order is specified, it applies to all subsequent columns until another sort order is specified, as in the previous example. This attribute is sometimes called *sticky sort order*. For example, the following cursor declaration orders retrieval by LAST_NAME in descending order, then refines it alphabetically within LAST_NAME groups by FIRST_NAME in ascending order:

```
EXEC SQL
   DECLARE PHONE_LIST CURSOR FOR
      SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
         FROM EMPLOYEE
         WHERE PHONE_EXT IS NOT NULL
         ORDER BY LAST_NAME DESC, FIRST_NAME ASC;
```

Specifying Collation Order in an ORDER BY Clause

When CHAR or VARCHAR columns are ordered in a SELECT statement, it can be necessary to specify a collation order for the ordering, especially if columns used for ordering use different collation orders.

To specify the collation order to use for ordering a column in the ORDER BY clause, include a COLLATE clause after the column name. For example, in the following ORDER BY clause, a different collation order for each of two columns is specified:

```
. . .
ORDER BY LNAME COLLATE FR_CA, FNAME COLLATE FR_FR;
```

For more information about collation order and a list of available collations in InterBase, see the *Data Definition Guide*.

Grouping Rows With GROUP BY

The optional GROUP BY clause enables a query to return summary information about groups of rows that share column values instead of returning each qualifying row. The complete syntax of GROUP BY is:

```
GROUP BY col [COLLATE collation] [, col [COLLATE collation] ...]
```

For example, consider two cursor declarations. The first declaration returns the names of all employees each department, and arranges retrieval in ascending alphabetic order by department and employee name.

```
EXEC SQL
   DECLARE DEPT_EMP CURSOR FOR
      SELECT DEPARTMENT, LAST_NAME, FIRST_NAME
         FROM DEPARTMENT D, EMPLOYEE E
         WHERE D.DEPT_NO = E.DEPT_NO"
         ORDER BY DEPARTMENT, LAST_NAME, FIRST_NAME;
```

In contrast, the next cursor illustrates the use of aggregate functions with GROUP BY to return results known as *group aggregates*. It returns the average salary of all employees in each department. The GROUP BY clause assures that average salaries are calculated and retrieved based on department names, while the ORDER BY clause arranges retrieved rows alphabetically by department name.

```
EXEC SQL
   DECLARE AVG_DEPT_SAL CURSOR FOR
      SELECT DEPARTMENT, AVG(SALARY)
         FROM DEPARTMENT D, EMPLOYEE E
         WHERE D.DEPT_NO = E.DEPT_NO
         GROUP BY DEPARTMENT
         ORDER BY DEPARTMENT;
```

Specifying Collation Order in a GROUP BY Clause

When CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation orders.

To specify the collation order to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name. For example, in the following GROUP BY clause, the collation order for two columns is specified:

```
. . .
GROUP BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For more information about collation order and a list of collation orders available in InterBase, see the *Data Definition Guide*.

Limitations of GROUP BY

When using GROUP BY, be aware of the following limitations:

- Each column name that appears in a GROUP BY clause must also be specified in the SELECT clause.
- GROUP BY cannot specify a column whose values are derived from a mathematical, aggregate, or user-defined function.

- GROUP BY cannot be used in SELECT statements that:
  - Contain an INTO clause (singleton selects).
  - Use a subquery with a FROM clause which references a view whose definition contains a GROUP BY or HAVING clause.
- For each SELECT clause in a query, including subqueries, there can only be one GROUP BY clause.

Restricting Grouped Rows With HAVING

Just as a WHERE clause reduces the number of rows returned by a SELECT clause, the HAVING clause can be used to reduce the number of rows returned by a GROUP BY clause. The syntax of HAVING is:

```
HAVING <search_condition>
```

HAVING uses search conditions that are like the search conditions that can appear in the WHERE clause, but with the following restrictions:

- Each search condition usually corresponds to an aggregate function used in the SELECT clause.
- The FROM clause of a subquery appearing in a HAVING clause cannot name any table or view specified in the main query's FROM clause.
- A correlated subquery cannot be used in a HAVING clause.

For example, the following cursor declaration returns the average salary for all employees in each department. The GROUP BY clause assures that average salaries are calculated and retrieved based on department names. The HAVING clause restricts retrieval to those groups where the average salary is greater than 60,000, while the ORDER BY clause arranges retrieved rows alphabetically by department name.

```
EXEC SQL
   DECLARE SIXTY_THOU CURSOR FOR
      SELECT DEPARTMENT, AVG(SALARY)
         FROM DEPARTMENT D, EMPLOYEE E
         WHERE D.DEPT_NO = E.DEPT_NO
         GROUP BY DEPARTMENT
         HAVING AVG(SALARY) > 60000
         ORDER BY DEPARTMENT;
```

Note HAVING can also be used without GROUP BY. In this case, all rows retrieved by a SELECT are treated as a single group, and each column named in the SELECT clause is normally operated on by an aggregate function.

For more information about search conditions, see “Restricting Row Retrieval With WHERE,” in this chapter. For more information about subqueries, see “Using Subqueries,” in this chapter.

Specifying a Query Plan With PLAN

To process a SELECT statement, InterBase uses an internal algorithm, called the *query optimizer*, to determine the most efficient plan for retrieving data. Usually the most efficient retrieval plan also results in the fastest retrieval time. Occasionally the optimizer may choose a plan that is less efficient. For example, when the number of rows in a table grows sufficiently large, or when many duplicate rows are inserted or deleted from indexed columns in a table, but the index’s selectivity is not recomputed, the optimizer might choose a less efficient plan.

For these occasions, SELECT provided an optional PLAN clause that enables a knowledgeable programmer to specify a retrieval plan. A query plan is built around the availability of indexes, the way indexes are joined or merged, and a chosen access method.

To specify a query plan, use the following PLAN syntax:

```
PLAN <plan_expr>

<plan_expr> =
   [JOIN | [SORT] MERGE] (<plan_item> | <plan_expr>
   [, <plan_item> | <plan_expr> ...])

<plan_item> = {table | alias}
   NATURAL | INDEX ( <index> [, <index> ...]) | ORDER <index>
```

The PLAN syntax enables specifying a single table, or a join of two or more tables in a single pass. Plan expressions can be nested in parentheses to specify any combination of joins.

During retrieval, information from different tables is joined to speed retrieval. If indexes are defined for the information to be joined, then these indexes are used to perform a join. The optional JOIN keyword can be used to document this type of operation. When no indexes exist for the information to join, retrieval speed can be improved by specifying SORT MERGE instead of JOIN.

A <*plan_item*> is the name of a table to search for data. If a table is used more than once in a query, aliases must be used to distinguish them in the PLAN clause. Part of the <*plan_item*> specification indicates the way that rows should be accessed. The following choices are possible:

- NATURAL, the default order, specifies that rows are accessed sequentially in no defined order. For unindexed items, this is the only option.

- INDEX specifies that one or more indexes should be used to access items. All indexes to be used must be specified. If any Boolean or join terms remain after all indexes are used, they will be evaluated without benefit of an index. If any indexes are specified that cannot be used, an error is returned.
- ORDER specifies that items are to be sorted based on a specified index.

Selecting a Single Row

An operation that retrieves a single row of data is called a *singleton select*. To retrieve a single row from a table, to retrieve a column defined with a unique index, or to select an aggregate value like COUNT() or AVG() from a table, use the following SELECT statement syntax:

```
SELECT <col> [, <col> ...]
   INTO :variable [, :variable ...]
   FROM table
   WHERE <search_condition>;
```

The mandatory INTO clause specifies the host variables where retrieved data is copied for use in the program. Each host variable's name must be preceded by a colon (:). For each column retrieved, there must be one host variable of a corresponding data type. Columns are retrieved in the order they are listed in the SELECT clause, and are copied into host variables in the order the variables are listed in the INTO clause.

The WHERE clause must specify a search condition that guarantees that only one row is retrieved. If the WHERE clause does not reduce the number of rows returned to a single row, the SELECT fails.

Important To select data from a table, a user must have SELECT privilege for a table, or a stored procedure invoked by the user's application must have SELECT privileges for the table.

For example, the following SELECT retrieves information from the DEPARTMENT table for the department, Publications:

```
EXEC SQL
   SELECT DEPARTMENT, DEPT_NO, HEAD_DEPT, BUDGET, LOCATION, PHONE_NO
      INTO :deptname, :dept_no, :manager, :budget, :location, :phone
      FROM DEPARTMENT
      WHERE DEPARTMENT = "Publications";
```

When SQL retrieves the specified row, it copies the value in DEPARTMENT to the host variable, *deptname*, copies the value in DEPT_NO to :*dept_no*, copies the value in HEAD_DEPT to :*manager*, and so on.

Selecting Multiple Rows

Most queries specify search conditions that retrieve more than one row. For example, a query that asks to see all employees in a company that make more than $60,000 can retrieve many employees.

Because host variables can only hold a single column value at a time, a query that returns multiple rows must build a temporary table in memory, called a *results table*, from which rows can then be extracted and processed, one at a time, in sequential order. SQL keeps track of the next row to process in the results table by establishing a pointer to it, called a *cursor*.

Important In dynamic SQL (DSQL), the process for creating a query and retrieving data is somewhat different. For more information about multi-row selection in DSQL, see "Selecting Multiple Rows in DSQL," in this chapter.

To retrieve multiple rows into a results table, establish a cursor into the table, and process individual rows in the table, SQL provides the following sequence of statements:

1. DECLARE CURSOR
   - Establishes a name for the cursor.
   - Specifies the query to perform.
2. OPEN executes the query, builds the results table, and positions the cursor at the start of the table.
3. FETCH retrieves a single row at a time from the results table into host variables for program processing.
4. CLOSE releases system resources when all rows are retrieved.

Important To select data from a table, a user must have SELECT privilege for a table, or a stored procedure invoked by the user's application must have SELECT privilege for it.

Declaring a Cursor

To declare a cursor and specify rows of data to retrieve, use the DECLARE CURSOR statement. DECLARE CURSOR is a descriptive, non-executable statement. InterBase uses the information in the statement to prepare system resources for the cursor when it is opened, but does not actually perform the query. Because DECLARE CURSOR is non-executable, SQLCODE is not assigned when this statement is used.

The syntax for DECLARE CURSOR is:

```
DECLARE cursorname CURSOR FOR
   SELECT <col> [, <col> ...]
      FROM table [, <table> ...]
      WHERE <search_condition>
      [GROUP BY col [, col ...]]
      [HAVING <search_condition>]
      [ORDER BY col [ASC | DESC] [, col ...] [ASC | DESC]
         | FOR UPDATE OF col [, col ...]];
```

The *cursorname* is used in subsequent OPEN, FETCH, and CLOSE statements to identify the active cursor.

With the following exceptions, the SELECT statement inside a DECLARE CURSOR is similar to a stand-alone SELECT:

- A SELECT in a DECLARE CURSOR cannot include an INTO clause.
- A SELECT in a DECLARE CURSOR can optionally include either an ORDER BY clause or a FOR UPDATE clause.

For example, the following statement declares a cursor:

```
EXEC SQL
   DECLARE TO_BE_HIRED CURSOR FOR
      SELECT D.DEPARTMENT, D.LOCATION, P.DEPARTMENT
      FROM DEPARTMENT D, DEPARTMENT P
      WHERE D.MNGR_NO IS NULL
         AND D.HEAD_DEPT = P.DEPT_NO;
```

Permitting Updates Through Cursors With FOR UPDATE

In many applications, data retrieval and update may be interdependent. DECLARE CURSOR supports an optional FOR UPDATE clause that optionally lists columns in retrieved rows that can be modified. For example, the following statement declares such a cursor:

```
EXEC SQL
   DECLARE H CURSOR FOR
      SELECT CUST_NO
      FROM CUSTOMER
      WHERE ON_HOLD = "*"
      FOR UPDATE OF ON_HOLD;
```

If a column list after FOR UPDATE is omitted, all columns retrieved for each row may be updated. For example, the following query enables updating for two columns:

```
EXEC SQL
   DECLARE H CURSOR FOR
```

```
SELECT CUST_NAME CUST_NO
FROM CUSTOMER
WHERE ON_HOLD = "*";
```

For more information about updating columns through a cursor, see "Updating Multiple Rows," in this chapter.

Opening a Cursor

Before data selected by a cursor can be accessed, the cursor must be opened with the OPEN statement. OPEN activates the cursor and builds a results table. It builds the results table based on the selection criteria specified in the DECLARE CURSOR statement. The rows in the results table comprise the *active set* of the cursor.

For example, the following statement opens a previously declared cursor called DEPT_EMP:

```
EXEC SQL
   OPEN DEPT_EMP;
```

When InterBase executes the OPEN statement, the cursor is positioned at the start of the first row in the results table.

Fetching Rows With a Cursor

Once a cursor is opened, rows can be retrieved, one at a time, from the results table by using the FETCH statement. FETCH:

1. Retrieves the next available row from the results table.
2. Copies those rows into the host variables specified in the INTO clause of the FETCH statement.
3. Advances the cursor to the start of the next available row or sets SQLCODE to 100, indicating the cursor is at the end of the results table and there are no more rows to retrieve.

The complete syntax of the FETCH statement in SQL is:

```
FETCH <cursorname> INTO :variable [[INDICATOR] :variable]
   [, :variable [[INDICATOR] :variable>] ...];
```

Important In dynamic SQL (DSQL) multi-row select processing, a different FETCH syntax is used. For more information about retrieving multiple rows in DSQL, see "Fetching Rows With a DSQL Cursor," in this chapter.

For example, the following statement retrieves a row from the results table for the DEPT_EMP cursor, and copies its column values into the host-language variables, *deptname*, *lname*, and *fname*:

```
EXEC SQL
   FETCH DEPT_EMP
      INTO :deptname, :lname, :fname;
```

To process each row in a results table in the same manner, enclose the FETCH statement in a host-language looping construct. For example, the following C code fetches and prints each row defined for the DEPT_EMP cursor:

```
. . .
EXEC SQL
   FETCH DEPT_EMP
      INTO :deptname, :lname, :fname;
while (!SQLCODE)
{
   printf("%s %s works in the %s department.\n", fname,
      lname, deptname);
   EXEC SQL
      FETCH DEPT_EMP
         INTO :deptname, :lname, :fname;
}
EXEC SQL
   CLOSE DEPT_EMP;
. . .
```

Every FETCH statement should be tested to see if the end of the active set is reached. The previous example operates in the context of a while loop that continues processing as long as SQLCODE is zero. If SQLCODE is 100, it indicates that there are no more rows to retrieve. If SQLCODE is less than zero, it indicates that an error occurred.

Retrieving Indicator Status

Any column can have a NULL value, except those defined with the NOT NULL or UNIQUE integrity constraints. Rather than store a value for the column, InterBase sets a flag indicating the column has no assigned value.

To determine if a value returned for a column is NULL, follow each variable named in the INTO clause with the INDICATOR keyword and the name of a short integer variable, called an *indicator variable*, where InterBase should store the status of the NULL value flag for the column. If the value retrieved is:

- NULL, the indicator variable is set to -1.
- Not NULL, the indicator parameter is set to 0.

For example, the following C code declares three host-language variables, *department*, *manager*, and *missing_manager*, then retrieves column values into *department*, *manager*, and a status flag for the column retrieved into *manager*, *missing_manager*, with a FETCH from a previously declared cursor, GETCITY:

```
. . .
char department[26];
char manager[36];
short missing_manager;
. . .
FETCH GETCITY INTO :department, :manager INDICATOR :missing_manager;
```

The optional INDICATOR keyword can be omitted:

```
FETCH GETCITY INTO :department, :manager :missing_manager;
```

Often, the space between the variable that receives the actual contents of a column and the variable that holds the status of the NULL value flag is also omitted:

```
FETCH GETCITY INTO :department, :manager:missing_manager;
```

Note While InterBase enforces the SQL requirement that the number of host variables in a FETCH must equal the number of columns specified in DECLARE CURSOR, indicator variables in a FETCH statement are *not* counted toward the column count.

Refetching Rows With a Cursor

The only supported cursor movement is forward in sequential order through the active set.

To revisit previously fetched rows, close the cursor and then reopen it with another OPEN statement. For example, the following statements close the DEPT_EMP cursor, then recreate it, effectively repositioning the cursor at the start of the DEPT_EMP results table:

```
EXEC SQL
   CLOSE DEPT_EMP;
EXEC SQL
   OPEN DEPT_EMP;
```

Closing the Cursor

When the end of a cursor's active set is reached, a cursor should be closed to free up system resources. To close a cursor, use the CLOSE statement. For example, the following statement closes the DEPT_EMP cursor:

```
EXEC SQL
   CLOSE DEPT_EMP;
```

Programs can check for the end of the active set by examining SQLCODE, which is set to 100 to indicate there are no more rows to retrieve.

A Complete Cursor Example

The following program declares a cursor, opens the cursor, and then loops through the cursor's active set, fetching and printing values. The program closes the cursor when all processing is finished or an error occurs.

```
#include <stdio.h>
EXEC SQL
   BEGIN DECLARE SECTION;
      char deptname[26];
      char lname[16];
      char fname[11];
EXEC SQL
   END DECLARE SECTION;

main ()
{
   EXEC SQL
      WHENEVER SQLERROR GO TO abend;
   EXEC SQL
      DECLARE DEPT_EMP CURSOR FOR
         SELECT DEPARTMENT, LAST_NAME, FIRST_NAME
            FROM DEPARTMENT D, EMPLOYEE E
            WHERE D.DEPT_NO = E.DEPT_NO"
            ORDER BY DEPARTMENT, LAST_NAME, FIRST_NAME;
   EXEC SQL
      OPEN DEPT_EMP;
   EXEC SQL
      FETCH DEPT_EMP
         INTO :deptname, :lname, :fname;
   while (!SQLCODE)
   {
      printf("%s %s works in the %s department.\n",fname,
         lname, deptname);
      EXEC SQL
         FETCH DEPT_EMP
            INTO :deptname, :lname, :fname;
   }
   EXEC SQL
      CLOSE DEPT_EMP;
   exit();

   abend:
   if (SQLCODE)
```

```
    {
       isc_print_sqlerror();
       EXEC SQL
          ROLLBACK;
       EXEC SQL
          CLOSE_DEPT_EMP;
       EXEC SQL
          DISCONNECT ALL;
       exit(1)
    }
    else
    {
       EXEC SQL
          COMMIT;
       EXEC SQL
          DISCONNECT ALL;
       exit()
    }
}
```

Selecting Rows With NULL Values

Any column can have NULL values, except those defined with the NOT NULL or UNIQUE integrity constraints. Rather than store a value for the column, InterBase sets a flag indicating the column has no assigned value.

Use IS NULL in a WHERE clause search condition to query for NULL values. For example, some rows in the DEPARTMENT table do not have a value for the BUDGET column. Departments with no stored budget have the NULL value flag set for that column. The following cursor declaration retrieves rows for departments without budgets for possible update:

```
EXEC SQL
   DECLARE NO_BUDGET CURSOR FOR
      SELECT DEPARTMENT, BUDGET
      FROM DEPARTMENT
      WHERE BUDGET IS NULL
      FOR UPDATE OF BUDGET;
```

Note To determine if a column has a NULL value, use an indicator variable. For more information about indicator variables, see “Retrieving Indicator Status,” in this chapter.

A direct query on a column containing a NULL value returns zero for numbers, blanks for characters, and 17 November 1858 for dates. For example, the following cursor declaration retrieves all department budgets, even those with NULL values, which are reported as zero:

```
EXEC SQL
   DECLARE ALL_BUDGETS CURSOR FOR
```

```
SELECT DEPARTMENT, BUDGET
FROM DEPARTMENT
ORDER BY BUDGET DESCENDING;
```

Limitations on NULL Values

Because InterBase treats NULL values as non-values, the following limitations on NULL values in queries should be noted:

- Rows with NULL values are sorted after all other rows.
- NULL values are skipped by all aggregate operations, except for COUNT(*).
- NULL values cannot be elicited by a negated test in a search condition.
- NULL values cannot satisfy a join condition.

NULL values can be tested in comparisons. If a value on either side of a comparison operator is NULL, the result of the comparison is Unknown.

For the Boolean operators (NOT, AND, and OR), the following considerations are made:

- NULL values with NOT always returns Unknown.
- NULL values with AND return Unknown unless one operand for AND is false. In this latter case, False is returned.
- NULL values with OR return Unknown unless one operand for OR is true. In this latter case, True is returned.

For information about defining alternate NULL values, see the *Data Definition Guide*.

Selecting Rows Through a View

To select a subset of rows available through a view, substitute the name of the view for a table name in the FROM clause of a SELECT. For example, the following cursor produces a list of employee phone numbers based on the PHONE_VIEW view:

```
EXEC SQL
   DECLARE PHONE_LIST CURSOR FOR
      SELECT FIRST_NAME, LAST_NAME, PHONE_EXT
      FROM PHONE_VIEW
      WHERE EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO;
```

A view can be a join. Views can also be used in joins, themselves, in place of tables. For more information about views in joins, see “Joining Tables,” in this chapter.

Selecting Multiple Rows in DSQL

In DSQL users are usually permitted to specify queries at run time. To accommodate any type of query the user supplies, DSQL requires the use of extended SQL descriptor areas (XSQLDAs) where a query’s input and output can be prepared and described. For queries returning multiple rows, DSQL supports variations of the DECLARE CURSOR, OPEN, and FETCH statements that make use of the XSQLDA.

To retrieve multiple rows into a results table, establish a cursor into the table, and process individual rows in the table. DSQL provides the following sequence of statements:

1. PREPARE establishes the user-defined query specification in the XSQLDA structure used for output.
2. DECLARE CURSOR:
   - Establishes a name for the cursor.
   - Specifies the query to perform.
3. OPEN executes the query, builds the results table, and positions the cursor at the start of the table.
4. FETCH retrieves a single row at a time from the results table for program processing.
5. CLOSE releases system resources when all rows are retrieved.

The following three sections describe how to declare a DSQL cursor, how to open it, and how to fetch rows using the cursor. For more information about creating and filling XSQLDA structures, and preparing DSQL queries with PREPARE, see Chapter 15: “Using Dynamic SQL.” For more information about closing a cursor, see “Closing the Cursor,” in this chapter.

Declaring a DSQL Cursor

DSQL must declare a cursor based on a user-defined SELECT statement. Usually, DSQL programs:

- Prompt the user for a query (SELECT).
- Store the query in a host-language variable.
- Issue a PREPARE statement that uses the host-language variable to describe the query results in an XSQLDA.
- Declare a cursor using the query alias.

The complete syntax for DECLARE CURSOR in DSQL is:

```
DECLARE cursorname CURSOR FOR queryname;
```

For example, the following C code fragment declares a string variable, *querystring*, to hold the user-defined query, gets a query from the user and stores it in *querystring*, uses *querystring* to PREPARE a query called QUERY, then declares a cursor, C, that uses QUERY:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char querystring [512];
      XSQLDA *InputSqlda, *OutputSqlda;
EXEC SQL
   END DECLARE SECTION;
. . .
printf("Enter query: "); /* prompt for query from user */
gets(querystring); /* get the string, store in querystring */
. . .
EXEC SQL
   PREPARE QUERY INTO OutputSqlda FROM :querystring;
. . .
EXEC SQL
   DECLARE C CURSOR FOR QUERY;
```

For more information about creating and filling XSQLDA structures, and preparing DSQL queries with PREPARE, see Chapter 15: “Using Dynamic SQL.”

Opening a DSQL Cursor

The OPEN statement in DSQL establishes a results table from the input parameters specified in a previously declared and populated XSQLDA. A cursor must be opened before data can be retrieved. The syntax for a DSQL OPEN is:

```
OPEN cursorname USING DESCRIPTOR sqldaname;
```

For example, the following statement opens the cursor, C, using the XSQLDA, *InputSqlda*:

```
EXEC SQL
   OPEN C USING DESCRIPTOR InputSqlda;
```

Fetching Rows With a DSQL Cursor

DSQL uses the FETCH statement to retrieve rows from a results table. The rows are retrieved according to specifications provided in a previously established and populated extended SQL descriptor area (XSQLDA) that describes the user's request. The syntax for the DSQL FETCH statement is:

```
FETCH cursorname USING DESCRIPTOR descriptorname;
```

For example, the following C code fragment declares XSQLDA structures for input and output, and illustrates how the output structure is used in a FETCH statement:

```
. . .
XSQLDA *InputSqlda, *OutputSqlda;
. . .
EXEC SQL
   FETCH C USING DESCRIPTOR OutputSqlda;
. . .
```

For more information about creating and filling XSQLDA structures, and preparing DSQL queries with PREPARE, see Chapter 15: "Using Dynamic SQL."

Joining Tables

Joins enable retrieval of data from two or more tables in a database with a single SELECT. The tables from which data is to be extracted are listed in the FROM clause. Optional syntax in the FROM clause can reduce the number of rows returned, and additional WHERE clause syntax can further reduce the number of rows returned.

From the information in a SELECT that describes a join, InterBase builds a table that contains the results of the join operation, the *results table*, sometimes also called a *dynamic* or *virtual table*.

InterBase supports two basic types of joins:

- *Inner joins* link rows in tables based on specified join conditions, and return only those rows that match the join conditions. There are three types of inner joins:
  - *Equi-joins* link rows based on common values or equality relationships in the join columns.
  - Joins that link rows based on comparisons other than equality in the join columns. There is not an officially recognized name for these

types of joins, but for simplicity's sake they may be categorized as *comparative joins*, or *non-equi-joins*.

- *Reflexive* or *self-joins*, compare values within a column of a single table.

- *Outer joins* link rows in tables based on specified join conditions and return both rows that match the join conditions, and all other rows from one or more tables even if they do not match the join condition.

The most commonly used joins are inner joins, because they both restrict the data returned, and show a clear relationship between two or more tables. Outer joins, however, are useful for viewing joined rows against a background of rows that do not meet the join conditions.

Choosing Join Columns

Regardless of join types, to create a useful join, the columns that are compared should be a PRIMARY KEY or a FOREIGN KEY. Joined columns need not have the same names, but they must be of compatible data types. For example, INTEGER, DECIMAL, NUMERIC, and FLOAT data types can be compared to one another because they are all numbers. String values, like CHAR and VARCHAR, can only be compared to other string values unless they contain ASCII values that are all numbers. The CAST() function can be used to force translation of one InterBase data type to another for comparisons. For more information about CAST(), see "Using CAST( ) for Data Type Conversions," in this chapter.

Important If a joined column contains a NULL value for a given row, InterBase does *not* include that row in the results table unless performing an outer join.

Using Inner Joins

InterBase supports two methods for creating inner joins. For portability and compatibility with existing SQL applications, InterBase continues to support the old SQL method for specifying joins. In older versions of SQL, there is no explicit join language. An inner join is specified by listing tables to join in the FROM clause of a SELECT, and the columns to compare in the WHERE clause.

For example, the following join returns all departments and department managers where the manager's salary is at least 20% of a department's total salary:

```
EXEC SQL
   DECLARE BIG_SAL CURSOR FOR
      SELECT D.DEPARTMENT, D.MANAGER, E.SALARY
```

```
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.MNGR_NO = E.EMP_NO.NAME AND E.SALARY/5
   >= (SELECT AVG(S.SALARY) FROM S EMPLOYEE
      WHERE D.DEPT_NO = S.DEPT_NO)
ORDER BY D.DEPARTMENT;
```

InterBase also implements new, explicit join syntax based on SQL-92:

```
SELECT col [, col ...] | *
   FROM <tablerefleft> [INNER] JOIN <tablerefright>
      [ON <searchcondition>]
   [WHERE <searchcondition>];
```

The join is explicitly declared in the FROM clause using the JOIN keyword. The table reference appearing to the left of the JOIN keyword is called the *left table*, while the table to the right of the JOIN is called the *right table*. Search conditions based on a column in the right table can be specified in an optional ON clause following the right table reference. For example, using the new join syntax, the previously described query can be rewritten as:

```
EXEC SQL
   DECLARE BIG_SAL CURSOR FOR
      SELECT D.DEPARTMENT, D.MANAGER, E.SALARY
         FROM DEPARTENT D JOIN EMPLOYEE E ON D.MNGR_NO = E.EMP_NO
            AND E.SALARY/5 >= (SELECT AVG(S.SALARY) FROM S EMPLOYEE
            WHERE D.DEPT_NO = S.DEPT_NO)
         ORDER BY D.DEPARTMENT;
```

The new join syntax offers several advantages. An explicit join declaration makes the intention of the program clear when reading its source code.

The ON clause enables join search conditions to be expressed in the FROM clause. The search condition that follows the ON clause is the only place where retrieval of rows can be restricted based on columns appearing in the right table. The WHERE clause can be used to further restrict rows based solely on columns in the left table.

The FROM clause also permits the use of table references, parenthetical, nested joins whose result tables are created and then processed as if they were actual tables stored in a database. For more information about nested joins, see “Using Nested Joins,” in this chapter.

Creating Equi-joins

An inner join that matches values in join columns is called an *equi-join*. Equi-joins are among the most common join operations. The ON clause in an equi-join always takes the form:

```
ON t1.column = t2.column
```

For example, the following join returns a list of cities around the world if the capital cities also appear in the CITIES table, and also returns the populations of those cities:

```
EXEC SQL
   DECLARE CAPPOP CURSOR FOR
      SELECT COU.NAME, COU.CAPITAL, CIT.POPULATION
         FROM COUNTRIES COU JOIN CITIES CIT ON CIT.NAME = COU.CAPITAL
         WHERE COU.CAPITAL NOT NULL
         ORDER BY COU.NAME;
```

In this example, the ON clause specifies that the CITIES table must contain a city name that matches a capital name in the COUNTRIES table if a row is to be returned. Note that the WHERE clause restricts rows retrieved from the COUNTRIES table to those where the CAPITAL column contains a value.

Creating Joins Based on Non-equality Comparison Operators

Inner joins can compare values in join columns using other comparison operators besides the equality operator. For example, a join might be based on a column in one table having a value less than the value in a column in another table. The ON clause in a comparison join always takes the form:

```
ON t1.column <operator> t2.column
```

where <*operator*> is a valid comparison operator. For a list of valid comparison operators, see “Using Comparison Operators in Expressions,” in this chapter.

For example, the following join returns information about provinces in Canada that are larger than the state of Alaska in the United States:

```
EXEC SQL
   DECLARE BIGPROVINCE CURSOR FOR
      SELECT S.STATE_NAME, S.AREA, P.PROVINCE_NAME, P.AREA
         FROM STATES S JOIN PROVINCE P ON P.AREA > S.AREA AND
            P.COUNTRY = "Canada"
         WHERE S.STATE_NAME = "Alaska";
```

In this example, the first comparison operator in the ON clause tests to see if the area of a province is greater than the area of any state (the WHERE clause restricts final output to display only information for provinces that are larger in area than the state of Alaska).

Creating Self-joins

A *self-join* is an inner join where a table is joined to itself to correlate columns of data. For example, the RIVERS table lists rivers by name, and, for each river, lists

the river into which it flows. Not all rivers, of course, flow into other rivers. To discover which rivers flow into other rivers, and what their names are, the RIVERS table must be joined to itself:

```
EXEC SQL
   DECLARE RIVERSTORIVERS CURSOR FOR
      SELECT R1.RIVER, R2.RIVER
         FROM RIVERS R1 JOIN RIVERS R2 ON R2.OUTFLOW = R1.RIVER
         ORDER BY R1.RIVER, R2.SOURCE;
```

As this example illustrates, when a table is joined to itself, each invocation of the table *must* be assigned a unique correlation name (R1 and R2 are correlation names in the example). For more information about assigning and using correlation names, see "Declaring and Using Correlation Names," in this chapter.

Using Outer Joins

Outer joins produce a results table that contains columns from every row in one table, and a subset of rows from another table. Actually, one type of outer join returns all rows from each table, but this type of join is used less frequently than other types. Outer join syntax is very similar to that of inner joins:

```
SELECT col [, col ...] | *
   FROM <tablerefleft> {LEFT | RIGHT | FULL} [OUTER] JOIN
      <tablerefright> [ON <searchcondition>]
   [WHERE <searchcondition>];
```

Outer join syntax requires that you specify the type of join to perform. There are three possibilities:

- A *left outer join* retrieves all rows from the left table in a join, and retrieves any rows from the right table that match the search condition specified in the ON clause.
- A *right outer join* retrieves all rows from the right table in a join, and retrieves any rows from the left table that match the search condition specified in the ON clause.
- A *full outer join* retrieves all rows from both the left and right tables in a join regardless of the search condition specified in the ON clause.

Outer joins are useful for comparing a subset of data to the background of all data from which it is retrieved. For example, when listing those countries which contain the sources of rivers, it may be interesting to see those countries which are not the sources of rivers as well.

Using a Left Outer Join

The left outer join is more commonly used than other types of outer joins. The following left outer join retrieves those countries that contain the sources of rivers, and identifies those countries that do not have NULL values in the R.RIVERS column:

```
EXEC SQL
   DECLARE RIVSOURCE CURSOR FOR
      SELECT C.COUNTRY, R.RIVER
         FROM COUNTRIES C LEFT JOIN RIVERS R ON R.SOURCE = C.COUNTRY
         ORDER BY C.COUNTRY;
```

The ON clause enables join search conditions to be expressed in the FROM clause. The search condition that follows the ON clause is the only place where retrieval of rows can be restricted based on columns appearing in the right table. The WHERE clause can be used to further restrict rows based solely on columns in the left (outer) table.

Using a Right Outer Join

A right outer join retrieves all rows from the right table in a join, and only those rows from the left table that match the search condition specified in the ON clause. The following right outer join retrieves a list of rivers and their countries of origin, but also reports those countries that are not the source of any river:

```
EXEC SQL
   DECLARE RIVSOURCE CURSOR FOR
      SELECT R.RIVER, C.COUNTRY
         FROM RIVERS.R RIGHT JOIN COUNTRIES C ON C.COUNTRY = R.SOURCE
         ORDER BY C.COUNTRY;
```

Tip Most right outer joins can be rewritten as left outer joins by reversing the order in which tables are listed.

Using a Full Outer Join

A full outer join returns all selected columns that do not contain NULL values from each table in the FROM clause without regard to search conditions. It is useful to consolidate similar data from disparate tables.

For example, several tables in a database may contain city names. Assuming triggers have not been created that ensure that a city entered in one table is also entered in the others to which it also applies, one of the only ways to see a list of all cities in the database is to use full outer joins. The following example uses

two full outer joins to retrieve the name of every city listed in three tables, COUNTRIES, CITIES, and NATIONAL_PARKS:

```
EXEC SQL
   DECLARE ALLCITIES CURSOR FOR
      SELECT DISTINCT CIT.CITY, COU.CAPITAL, N.PARKCITY
         FROM (CITIES CIT FULL JOIN COUNTRIES COU) FULL
            JOIN NATIONAL_PARKS N;
```

This example uses a *nested* full outer join to process all rows from the CITIES and COUNTRIES tables. The result table produced by that operation is then used as the left table of the full outer join with the NATIONAL_PARKS table. For more information about using nested joins, see "Using Nested Joins," in this chapter.

Note In most databases where tables share similar or related information, triggers are usually created to ensure that all tables are updated with shared information. For more information about triggers, see the *Data Definition Guide*.

Using Nested Joins

The SELECT statement FROM clause can be used to specify any combination of available tables or *table references*, parenthetical, nested joins whose results tables are created and then processed as if they were actual tables stored in the database. Table references are flexible and powerful, enabling the succinct creation of complex joins in a single location in a SELECT.

For example, the following statement contains a parenthetical outer join that creates a results table with the names of every city in the CITIES table even if the city is not associated with a country in the COUNTRIES table. The results table is then processed as the left table of an inner join that returns only those cities that have professional sports teams of any kind, the name of the team, and the sport the team plays.

```
DECLARE SPORTSCITIES CURSOR FOR
   SELECT COU.COUNTRY, C.CITY, T.TEAM, T.SPORT
      FROM (CITIES CIT LEFT JOIN COUNTRIES COU ON COU.COUNTRY =
         CIT.COUNTRY) INNER JOIN TEAMS T ON T.CITY = C.CITY
      ORDER BY COU.COUNTRY;
```

For more information about left joins, see "Using Outer Joins," in this chapter.

Appending Tables

Sometimes two or more tables in a database are identically structured, or have columns that contain similar data. Where table structures overlap, information from those tables can be combined to produce a single results table that returns a projection for every qualifying row in both tables. The UNION clause retrieves all rows from each table, appends one table to the end of another, and eliminates duplicate rows.

Unions are commonly used to perform aggregate operations on tables.

The syntax for UNION is:

```
UNION SELECT col [, col ...] | * FROM <tableref> [, <tableref> ...]
```

For example, three tables, CITIES, COUNTRIES, and NATIONAL_PARKS, each contain the names of cities. Assuming triggers have not been created that ensure that a city entered in one table is also entered in the others to which it also applies, UNION can be used to retrieve the names of all cities that appear in any of these tables.

```
EXEC SQL
   DECLARE ALLCITIES CURSOR FOR
      SELECT CIT.CITY FROM CITIES CIT
         UNION SELECT COU.CAPITAL FROM COUNTRIES COU
         UNION SELECT N.PARKCITY FROM NATIONAL_PARKS N;
```

Tip If two or more tables share entirely identical structures—similarly named columns, identical data types, and similar data values in each column—UNION can return all rows for each table by substituting an asterisk (*) for specific column names in the SELECT clauses of the UNION.

Using Subqueries

A *subquery* is a parenthetical SELECT statement nested inside the WHERE clause of another SELECT statement, where it functions as a search condition to restrict the number of rows returned by the outer, or *parent*, query. A subquery can refer to the same table or tables as its parent query, or to other tables.

The elementary syntax for a subquery is:

```
SELECT [DISTINCT] col [, col ...]
   FROM <tableref> [, <tableref> ...]
      WHERE {expression {[NOT] IN | comparison_operator}
      | [NOT] EXISTS} (SELECT [DISTINCT] col [, col ...]
            FROM <tableref> [, <tableref> ...]
```

```
WHERE <search_condition>);
```

Because a subquery is a search condition, it is usually evaluated before its parent query, which then uses the result to determine whether or not a row qualifies for retrieval. The only exception is the *correlated subquery*, where the parent query provides values for the subquery to evaluate. For more information about correlated subqueries, see "Correlated Subqueries," in this chapter.

A subquery determines the search condition for a parent's WHERE clause in one of the following ways:

- Produces a list of values for evaluation by an IN operator in the parent query's WHERE clause, or where a comparison operator is modified by the ALL, ANY, or SOME operators.
- Returns a single value for use with a comparison operator.
- Tests whether or not data meets conditions specified by an EXISTS operator in the parent query's WHERE clause.

Subqueries can be nested within other subqueries as search conditions, establishing a chain of parent/child queries.

Simple Subqueries

A subquery is especially useful for extracting data from a single table when a self-join is inadequate. For example, it is impossible to retrieve a list of those countries with a larger than average area by joining the COUNTRIES table to itself. A subquery, however, can easily return that information.

```
EXEC SQL
   DECLARE LARGECOUNTRIES CURSOR FOR
      SELECT COUNTRY, AREA
         FROM COUNTRIES
         WHERE AREA > (SELECT AVG(AREA) FROM COUNTRIES);
         ORDER BY AREA;
```

In this example, both the query and subquery refer to the same table. Queries and subqueries can refer to different tables, too. For example, the following query refers to the CITIES table, and includes a subquery that refers to the COUNTRIES table:

```
EXEC SQL
   DECLARE EUROCAPPOP CURSOR FOR
      SELECT CIT.CITY, CIT.POPULATION
         FROM CITIES CIT
         WHERE CIT.CITY IN (SELECT COU.CAPITAL FROM COUNTRIES COU
            WHERE COU.CONTINENT = "Europe")
         ORDER BY CIT.CITY;
```

This example uses correlation names to distinguish between tables even though the query and subquery reference separate tables. Correlation names are only necessary when both a query and subquery refer to the same tables and those tables share column names, but it is good programming practice to use them. For more information about using correlation names, see "Declaring and Using Correlation Names," in this chapter.

Correlated Subqueries

A *correlated subquery* is a subquery that depends on its parent query for the values it evaluates. Because each row evaluated by the parent query is potentially different, the subquery is executed once for each row presented to it by the parent query.

For example, the following query lists each country for which there are three or more cities stored in the CITIES table. For each row in the COUNTRIES table, a country name is retrieved in the parent query, then used in the comparison operation in the subquery's WHERE clause to verify if a city in the CITIES table should be counted by the COUNT() function. If COUNT() exceeds 2 for a row, the row is retrieved.

```
EXEC SQL
   DECLARE TRICITIES CURSOR FOR
      SELECT COUNTRY
         FROM COUNTRIES COU
         WHERE 3 <= (SELECT COUNT (*)
            FROM CITIES CIT
            WHERE CIT.CITY = COU.CAPITAL);
```

Simple and correlated subqueries can be nested and mixed to build complex queries. For example, the following query retrieves the country name, capital city, and largest city of countries whose areas are larger than the average area of countries that have at least one city within 30 meters of sea level:

```
EXEC SQL
   DECLARE SEACOUNTRIES CURSOR FOR
      SELECT CO1.COUNTRY, C01.CAPITAL, CI1.CITY
         FROM COUNTRIES C01, CITIES CI1
         WHERE CO1.COUNTRY = CI1.COUNTRY AND CI1.POPULATION =
         (SELECT MAX(CI2.POPULATION)
            FROM CITIES CI2 WHERE CI2.COUNTRY = CI1.COUNTRY)
               AND CO1.AREA >
            (SELECT AVG (CO2.AREA)
               FROM COUNTRIES C02 WHERE EXISTS
               (SELECT *
               FROM CITIES CI3 WHERE CI3.COUNTRY = CO2.COUNTRY
               AND CI3.ALTITUDE <= 30));
```

When a table is separately searched by queries and subqueries, as in this example, each invocation of the table must establish a separate correlation name for the table. Using correlation names is the only method to assure that column references are associated with appropriate instances of their tables. For more information about correlation names, see "Declaring and Using Correlation Names," in this chapter.

Inserting Data

New rows of data are added to one table at a time with the INSERT statement. To insert data, a user or stored procedure must have INSERT privilege for a table.

The INSERT statement enables data insertion from two different sources:

- A VALUES clause that contains a list of values to add, either through hard-coded values, or host-language variables.
- A SELECT statement that retrieves values from one table to add to another.

The syntax of INSERT is as follows:

```
INSERT [TRANSACTION name] INTO table [(col [, col ...])]
   {VALUES (<val>[:ind] [, <val>[:ind] ...])
   | SELECT <clause>};
```

The list of columns into which to insert values is optional in DSQL applications. If it is omitted, then values are inserted into a table's columns according to the order in which the columns were created. If there are more columns than values, the remaining columns are filled with zeros.

Inserting Columns With VALUES

Use the VALUES clause to add a row of specific values to a table, or to add values entered by a user at run time. The list of values that follows the keyword can come from either from host-language variables, or from hard-coded assignments.

For example, the following statement adds a new row to the DEPARTMENT table using hard-coded value assignments:

```
EXEC SQL
   INSERT INTO DEPARTMENT (DEPT_NO, DEPARTMENT)
      VALUES (7734, "Marketing");
```

Because the DEPARTMENT table contains additional columns not specified in the INSERT, NULL values are assigned to the missing fields.

The following C code example prompts a user for information to add to the DEPARTMENT table, and inserts those values from host variables:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char department[26], dept_no[16];
      int dept_num;
EXEC SQL
   END DECLARE SECTION;
. . .
printf("Enter name of department: ");
gets(department);
printf("\nEnter department number: ");
dept_num = atoi(gets(dept_no));
EXEC SQL
   INSERT INTO COUNTRIES (DEPT_NO, DEPARTMENT)
      VALUES (:dept_num, :department);
```

When host variables are used in the values list, they must be preceded by colons (:) so that SQL can distinguish them from table column names.

Inserting Columns With SELECT

To insert values from one table into another row in the same table or into a row in another table, use a SELECT statement to specify a list of insertion values. For example, the following INSERT statement copies DEPARTMENT and BUDGET information about the publications department from the OLDDEPT table to the DEPARTMENT table. It also illustrates how values can be hard-coded into a SELECT statement to substitute actual column data.

```
EXEC SQL
   INSERT INTO DEPARTMENTS (DEPT_NO, DEPARTMENT, BUDGET)
      SELECT DEPT_NO, "Publications", BUDGET
         FROM OLDDEPT
         WHERE DEPARTMENT = "Documentation";
```

The assignments in the SELECT can include arithmetic operations. For example, suppose an application keeps track of employees by using an employee number. When a new employee is hired, the following statement inserts a new employee row into the EMPLOYEE table, and assigns a new employee number to the row by using a SELECT statement to find the current maximum employee number and adding one to it. It also reads values for LAST_NAME and FIRST_NAME from the host variables, *lastname*, and *firstname*.

```
EXEC SQL
```

```
INSERT INTO EMPLOYEE (EMP_NO, LAST_NAME, FIRST_NAME)
   SELECT (MAX(EMP_NO) + 1, :lastname, :firstname)
      FROM EMPLOYEE;
```

Inserting Rows With NULL Column Values

Sometimes when a new row is added to a table, values are not necessary or available for all its columns. In these cases, a NULL value should be assigned to those columns when the row is inserted. There are three ways to assign a NULL value to a column on insertion:

- Ignore the column.
- Assign a NULL value to the column. This is standard SQL practice.
- Use indicator variables.

Ignoring a Column

A NULL value is assigned to any column that is not explicitly specified in an INTO clause. When InterBase encounters an unreferenced column during insertion, it sets a flag for the column indicating that its value is unknown. For example, the DEPARTMENT table contains several columns, among them HEAD_DEPT, MNGR_NO, and BUDGET. The following INSERT does not provide values for these columns:

```
EXEC SQL
   INSERT INTO DEPARTMENT (DEPT_NO, DEPARTMENT)
      VALUES (:newdept_no, :newdept_name);
```

Because HEAD_DEPT, MNGR_NO, and BUDGET are not specified, InterBase sets the NULL value flag for each of these columns.

Note If a column is added to an existing table, InterBase sets a NULL value flag for all existing rows in the table.

Assigning a NULL Value to a Column

When a specific value is not provided for a column on insertion, it is standard SQL practice to assign a NULL value to that column. In InterBase a column is set to NULL by specifying NULL for the column in the INSERT statement.

For example, the following statement stores a row into the DEPARTMENT table, assigns the values of host variables to some columns, and assigns a NULL value to other columns:

```
EXEC SQL
   INSERT INTO DEPARTMENT
      (DEPT_NO, DEPARTMENT, HEAD_DEPT, MNGR_NO, BUDGET,
         LOCATION, PHONE_NO)
   VALUES (:dept_no, :dept_name, NULL, NULL, 1500000, NULL, NULL);
```

Using Indicator Variables

Another method for trapping and assigning NULL values—through indicator variables—is necessary in applications that prompt users for data, where users can choose not to enter values. By default, when InterBase stores new data, it stores zeroes for NULL numeric data, and spaces for NULL character data. Because zeroes and spaces may be valid data, it becomes impossible to distinguish missing data in the new row from actual zeroes and spaces.

To trap missing data with indicator variables, and store NULL value flags, follow these steps:

1. Declare a host-language variable to use as an indicator variable.
2. Test a value entered by the user and set the indicator variable to one of the following values:
   - 0. The host-language variable contains data.
   - -1. The host-language variable does not contain data.
3. Associate the indicator variable with the host variable in the INSERT statement using the following syntax:

```
INSERT INTO table (<col> [, <col> ...])
   VALUES (:variable [INDICATOR] :indicator [, :variable [INDICATOR]
   :indicator ...]);
```

Note The INDICATOR keyword is optional.

For example, the following C code fragment prompts the user for the name of a department, the department number, and a budget for the department. It tests that the user has entered a budget. If not, it sets the indicator variable, *bi*, to -1. Otherwise, it sets *bi* to 0. Finally, the program INSERTS the information into the DEPARTMENT table. If the indicator variable is -1, then no actual data is stored in the BUDGET column, but a flag is set for the column indicating that the value is NULL

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      short bi; /* indicator variable declaration */
      char department[26], dept_no_ascii[26], budget_ascii[26];
      long num_val; /* host variable for inserting budget */
```

```
      short dept_no;
EXEC SQL
   END DECLARE SECTION;
. . .
printf("Enter new department name: ");
gets(cidepartment);
printf("\nEnter department number: ");
gets(dept_no_ascii);
printf("\nEnter department's budget: ");
gets(budget_ascii);
if (budget_ascii = "")
{
   bi = -1; num_val = 0;
}
else
{
   bi = 0;
   num_val = atoi(budget_ascii);
}
dept_no = atoi(dept_no_ascii);
EXEC SQL
   INSERT INTO DEPARTMENT (DEPARTMENT, DEPT_NO, BUDGET)
      VALUES (:department, :dept_no, :num_val INDICATOR :bi);
. . .
```

Indicator status can also be determined for data retrieved from a table. For information about trapping NULL values retrieved from a table, see "Retrieving Indicator Status," in this chapter.

Inserting Data Through a View

New rows can be inserted through a view if the following conditions are met:

- The view is updatable. For a complete discussion of updatable views, see the *Data Definition Guide.*
- The view is created using the WITH CHECK OPTION.
- A user or stored procedure has INSERT privilege for the view.

Values can only be inserted through a view for those columns named in the view. InterBase stores NULL values for unreferenced columns. For example, suppose the view, PART_DEPT, is defined as follows:

```
EXEC SQL
   CREATE VIEW PART_DEPT
      (DEPARTMENT, DEPT_NO, BUDGET)
      AS SELECT DEPARTMENT, DEPT_NO, BUDGET
         FROM DEPARTMENT
         WHERE DEPT_NO NOT NULL AND BUDGET > 50000
      WITH CHECK OPTION;
```

Because PART_DEPT references a single table, DEPARTMENT, new data can be inserted for the DEPARTMENT, DEPT_NO, and BUDGET columns. The WITH CHECK OPTION assures that all values entered through the view fall within ranges of values that can be selected by this view. For example, the following statement inserts a new row for the Publications department through the PART_DEPT view:

```
EXEC SQL
   INSERT INTO PART_DEPT (DEPARTMENT, DEPT_NO, BUDGET)
      VALUES ("Publications", "7735", 1500000);
```

InterBase inserts NULL values for all other columns in the DEPARTMENT table that are not available directly through the view.

For information about creating a view, see Chapter 5: “Working With Data Definition Statements.” For the complete syntax of CREATE VIEW, see the *Language Reference*.

Specifying Transaction Names in an INSERT

InterBase enables an SQL application to run simultaneous transactions if:

- Each transaction is first named with a SET TRANSACTION statement. For a complete discussion of transaction handling and naming, see Chapter 4: “Working With Transactions.”
- Each data manipulation statement (SELECT, INSERT, UPDATE, DELETE, DECLARE, OPEN, FETCH, and CLOSE) specifies a TRANSACTION clause that identifies the name of the transaction under which it operates.
- SQL statements are not dynamic (DSQL). DSQL does not support user-specified transaction names.

With INSERT, the TRANSACTION clause intervenes between the INSERT keyword and the list of columns to insert, as in the following syntax fragment:

```
INSERT TRANSACTION name INTO table (col [, col ...])
```

The TRANSACTION clause is optional in single-transaction programs. It must be used in a multi-transaction program unless a statement operates under control of the default transaction, **gds__trans**. For example, the following INSERT is controlled by the transaction, T1:

```
EXEC SQL
   INSERT TRANSACTION T1 INTO DEPARTMENT (DEPARTMENT, DEPT_NO, BUDGET)
      VALUES (:deptname, :deptno, :budget INDICATOR :bi);
```

Updating Data

To change values for existing rows of data in a table, use the UPDATE statement. To update a table, a user or procedure must have UPDATE privilege for it. The syntax of UPDATE is:

```
UPDATE [TRANSACTION name] table
   SET col = <assignment> [, col = <assignment> ...]
   WHERE <search_condition> | WHERE CURRENT OF cursorname;
```

UPDATE changes values for columns specified in the SET clause; columns not listed in the SET clause are not changed. A single UPDATE statement can be used to modify any number of rows in a table. For example, the following statement modifies a single row:

```
EXEC SQL
   UPDATE DEPARTMENT
      SET DEPARTMENT = "Publications"
      WHERE DEPARTMENT = "Documentation";
```

The WHERE clause in this example targets a single row for update. If the same change should be propagated to a number of rows in a table, the WHERE clause can be more general. For example, to change all occurrences of “Documentation” to “Publications” for all departments in the DEPARTMENT table where DEPARTMENT equals “Documentation,” the UPDATE statement would be as follows:

```
EXEC SQL
   UPDATE DEPARTMENT
      SET DEPARTMENT = "Publications"
      WHERE DEPARTMENT = "Documentation";
```

Using UPDATE to make the same modification to a number of rows is sometimes called a *mass update*, or a *searched update*.

The WHERE clause in an UPDATE statement can contain a subquery that references one or more other tables. For a complete discussion of subqueries, see “Using Subqueries,” in this chapter.

Updating Multiple Rows

There are two basic methods for modifying rows:

- The *searched update* method, where the same changes are applied to a number of rows, is most useful for automated updating of rows without a cursor.

- The *positioned update* method, where rows are retrieved through a cursor and updated row by row, is most useful for enabling users to enter different changes for each row retrieved.

A searched update is easier to program than a positioned update, but also more limited in what it can accomplish.

Using a Searched Update

Use a searched update to make the same changes to a number of rows. The UPDATE SET clause specifies the actual changes that are to be made to columns for each row that matches the search condition specified in the WHERE clause. Values to set can be specified as constants or variables.

For example, the following C code fragment prompts for a country name and a percentage change in population, then updates all cities in that country with the new population:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char country[26], asciimult[10];
      int multiplier;
EXEC SQL
   END DECLARE SECTION;
. . .
main ()
{
   printf("Enter country with city populations needing adjustment: ");
   gets(country);
   printf("\nPercent change (100%% to -100%%:");
   gets(asciimult);
   multiplier = atoi(asciimult);
   EXEC SQL
      UPDATE CITIES
         SET POPULATION  = POPULATION * (1 + :multiplier / 100)
         WHERE COUNTRY = :country;

   if (SQLCODE && (SQLCODE != 100))
   {
      isc_print_sqlerr(SQLCODE, isc_$status);
      EXEC SQL
         ROLLBACK RELEASE;
   }
   else
   {
      EXEC SQL
         COMMIT RELEASE;
   }
}
```

Important Searched updates cannot be performed on arrays of data types.

Using a Positioned Update

Use cursors to select rows for update when prompting users for changes on a row-by-row basis, and displaying pre- or post-modification values between row updates. Updating through a cursor is a seven-step process:

1. Declare host-language variables needed for the update operation.
2. Declare a cursor describing the rows to retrieve for update, and include the FOR UPDATE clause in DSQL. For more information about declaring and using cursors, see "Selecting Multiple Rows," in this chapter.
3. Open the cursor.
4. Fetch a row.
5. Display current values and prompt for new values.
6. Update the currently selected row using the WHERE CURRENT OF clause.
7. Repeat steps 3 to 7 until all selected rows are updated.

For example, the following C code fragment updates the POPULATION column by user-specified amounts for cities in the CITIES table that are in a country also specified by the user:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char country[26], asciimult[10];
      int multiplier;
EXEC SQL
   END DECLARE SECTION;
. . .
main ()
{
   EXEC SQL
      DECLARE CHANGEPOP CURSOR FOR
         SELECT CITY, POPULATION
            FROM CITIES
            WHERE COUNTRY = :country;

   printf("Enter country with city populations needing adjustment: ");
   gets(country);
   EXEC SQL
      OPEN CHANGEPOP;
   EXEC SQL
      FETCH CHANGEPOP INTO :country;
```

```
    while(!SQLCODE)
    {
       printf("\nPercent change (100%% to -100%%:");
       gets(asciimult);
       multiplier = atoi(asciimult);
       EXEC SQL
          UPDATE CITIES
             SET POPULATION  = POPULATION * (1 + :multiplier / 100)
             WHERE CURRENT OF CHANGEPOP;
       EXEC SQL
          FETCH CHANGEPOP INTO :country;
       if (SQLCODE && (SQLCODE != 100))
       {
          isc_print_sqlerr(SQLCODE, isc_$status);
          EXEC SQL
             ROLLBACK RELEASE;
          exit(1);
       }
    }
    EXEC SQL
       COMMIT RELEASE;
}
```

Important Using FOR UPDATE with a cursor causes rows to be fetched from the database one at a time. If FOR UPDATE is omitted, rows are fetched in batches.

Setting Column Values to NULL With UPDATE

To set a column's value to NULL during update, specify a NULL value for the column in the SET clause. For example, the following UPDATE sets the budget of all departments without managers to NULL:

```
EXEC SQL
   UPDATE DEPARTMENT
      SET BUDGET = NULL
      WHERE MNGR_NO = NULL;
```

Updating Through a View

Existing rows can be updated through a view if the following conditions are met:

- The view is updatable. For a complete discussion of updatable views, see the *Data Definition Guide.*
- The view is created using the WITH CHECK OPTION.
- A user or stored procedure has UPDATE privilege for the view.

Values can only be updated through a view for those columns named in the view. For example, suppose the view, PART_DEPT, is defined as follows:

```
EXEC SQL
   CREATE VIEW PART_DEPT
      (DEPARTMENT, NUMBER, BUDGET)
      AS SELECT DEPARTMENT, DEPT_NO, BUDGET
         FROM DEPARTMENT
      WITH CHECK OPTION;
```

Because PART_DEPT references a single table, data can be updated for the columns named in the view. The WITH CHECK OPTION assures that all values entered through the view fall within ranges prescribed for each column when the DEPARTMENT table was created. For example, the following statement updates the budget of the Publications department through the PART_DEPT view:

```
EXEC SQL
   UPDATE PART_DEPT
      SET BUDGET = 2505700
      WHERE DEPARTMENT = "Publications";
```

For information about creating a view, see Chapter 5: “Working With Data Definition Statements.” For the complete syntax of CREATE VIEW, see the *Language Reference*.

Specifying Transaction Names in UPDATE

InterBase enables an SQL application to run simultaneous transactions if:

- Each transaction is first named with a SET TRANSACTION statement. For a complete discussion of transaction handling and naming, see Chapter 4: “Working With Transactions.”
- Each data manipulation statement (SELECT, INSERT, UPDATE, DELETE, DECLARE, OPEN, FETCH, and CLOSE) specifies a TRANSACTION clause that identifies the name of the transaction under which it operates.
- SQL statements are not dynamic (DSQL). DSQL does not support multiple simultaneous transactions.

In UPDATE, the TRANSACTION clause intervenes between the UPDATE keyword and the name of the table to update, as in the following syntax:

```
UPDATE [TRANSACTION name] table
   SET col = <assignment> [, col = <assignment> ...]
   WHERE <search_condition> | WHERE CURRENT OF cursorname;
```

The TRANSACTION clause must be used in multi-transaction programs, but is optional in single-transaction programs or in programs where only one transaction is open at a time. For example, the following UPDATE is controlled by the transaction, T1:

```
EXEC SQL
   UPDATE TRANSACTION T1 DEPARTMENT
      SET BUDGET = 2505700
      WHERE DEPARTMENT = "Publications";
```

Deleting Data

To remove rows of data from a table, use the DELETE statement. To delete rows a user or procedure must have DELETE privilege for the table.

The syntax of DELETE is:

```
DELETE [TRANSACTION name] FROM table
   WHERE <search_condition> | WHERE CURRENT OF cursorname;
```

DELETE irretrievably removes entire rows from the table specified in the FROM clause, regardless of each column's data type.

A single DELETE can be used to remove any number of rows in a table. For example, the following statement removes the single row containing "Channel Marketing" from the DEPARTMENT table:

```
EXEC SQL
   DELETE FROM DEPARTMENT
      WHERE DEPARTMENT = "Channel Marketing:;
```

The WHERE clause in this example targets a single row for update. If the same deletion criteria apply to a number of rows in a table, the WHERE clause can be more general. For example, to remove all rows from the DEPARTMENT table with BUDGET values < $1,000,000, the DELETE statement would be as follows:

```
EXEC SQL
   DELETE FROM DEPARTMENT
      WHERE BUDGET < 1000000;
```

Using DELETE to remove a number of rows is sometimes called a *mass delete*.

The WHERE clause in a DELETE statement can contain a subquery that references one or more other tables. For a complete discussion of subqueries, see "Using Subqueries," in this chapter.

Deleting Multiple Rows

There are two methods for modifying rows:

- The *searched delete* method, where the same deletion condition applies to a number of rows, is most useful for automated removal of rows.
- The *positioned delete* method, where rows are retrieved through a cursor and deleted row by row, is most useful for enabling users to choose which rows that meet certain conditions should be removed.

A searched delete is easier to program than a positioned delete, but less flexible.

Using a Searched Delete

Use a searched delete to remove a number of rows that match a condition specified in the WHERE clause. For example, the following C code fragment prompts for a country name, then deletes all rows that have cities in that country:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char country[26];
EXEC SQL
   END DECLARE SECTION;
. . .
main ()
{
   printf("Enter country with cities to delete: ");
   gets(country);
   EXEC SQL
      DELETE FROM CITIES
         WHERE COUNTRY = :country;

   if(SQLCODE && (SQLCODE != 100))
   {
      isc_print_sqlerr(SQLCODE, isc_$status);
      EXEC SQL
         ROLLBACK RELEASE;
   }
   else
   {
      EXEC SQL
         COMMIT RELEASE;
   }
}
```

Using a Positioned Delete

Use cursors to select rows for deletion when users should decide deletion on a row-by-row basis, and displaying pre- or post-modification values between row updates. Updating through a cursor is a seven-step process:

1. Declare host-language variables needed for the delete operation.
2. Declare a cursor describing the rows to retrieve for possible deletion, and include the FOR UPDATE clause. For more information about declaring and using cursors, see “Selecting Multiple Rows,” in this chapter.
3. Open the cursor.
4. Fetch a row.
5. Display current values and prompt for permission to delete.
6. Delete the currently selected row using the WHERE CURRENT OF clause to specify the name of the cursor.
7. Repeat steps 3 to 7 until all selected rows are deleted.

For example, the following C code deletes rows in the CITIES table that are in North America only if a user types **Y** when prompted:

```
. . .
EXEC SQL
   BEGIN DECLARE SECTION;
      char cityname[26];
EXEC SQL
   END DECLARE SECTION;
char response[5];
. . .
main ()
{
   EXEC SQL
      DECLARE DELETECITY CURSOR FOR
         SELECT CITY,
            FROM CITIES
            WHERE CONTINENT = "North America";
   EXEC SQL
      OPEN DELETECITY;
   while (!SQLCODE)
   {
      EXEC SQL
         FETCH DELETECITY INTO :cityname;
      if (SQLCODE)
      {
         if (SQLCODE == 100)
         {
            printf("Deletions complete.");
```

```
            EXEC SQL
               COMMIT;
            EXEC SQL
               CLOSE DELETECITY;
            EXEC SQL
               DISCONNECT ALL:
         }
         isc_print_sqlerr(SQLCODE, isc_$status);
         EXEC SQL
            ROLLBACK;
         EXEC SQL
            DISCONNECT ALL;
         exit(1);
      }
      printf("\nDelete %s (Y/N)?", cityname);
      gets(response);
      if(response[0] == 'Y' || response == 'y')
      {
         EXEC SQL
            DELETE FROM CITIES
            WHERE CURRENT OF DELETECITY;
      if(SQLCODE && (SQLCODE != 100))
      {
         isc_print_sqlerr(SQLCODE, isc_$status);
         EXEC SQL
            ROLLBACK;
         EXEC SQL
            DISCONNECT;
         exit(1);
      }
   }
}
```

Deleting Through a View

Entire rows can be deleted through a view if the following conditions are met:

- The view is updatable. For a complete discussion of updatable views, see the *Data Definition Guide.*
- A user or stored procedure has DELETE privilege for the view.

For example, the following statement deletes all departments with budgets under $1,000,000, from the DEPARTMENT table through the PART_DEPT view:

```
EXEC SQL
   DELETE FROM PART_DEPT
      WHERE BUDGET < 1000000;
```

For information about creating a view, see Chapter 5: “Working With Data Definition Statements.” For CREATE VIEW syntax, see the *Language Reference.*

Specifying Transaction Names in a DELETE

InterBase enables an SQL application to run simultaneous transactions if:

- Each transaction is first named with a SET TRANSACTION statement. For a complete discussion of transaction handling and naming, see Chapter 4: “Working With Transactions.”
- Each data manipulation statement (SELECT, INSERT, UPDATE, DELETE, DECLARE, OPEN, FETCH, and CLOSE) specifies a TRANSACTION clause that identifies the name of the transaction under which it operates.
- SQL statements are not dynamic (DSQL). DSQL does not support multiple simultaneous transactions.

For DELETE, the TRANSACTION clause intervenes between the DELETE keyword and the FROM clause specifying the table from which to delete:

```
DELETE TRANSACTION name FROM table ...
```

The TRANSACTION clause is optional in single-transaction programs or in programs where only one transaction is open at a time. It must be used in a multi-transaction program. For example, the following DELETE is controlled by the transaction, T1:

```
EXEC SQL
   DELETE TRANSACTION T1 FROM PART_DEPT
      WHERE BUDGET < 1000000";
```

CHAPTER 7

Working With Dates

Most host languages do not support the DATE data type. Instead, they treat dates as strings or structures. InterBase supports a DATE data type that is stored in tables as two long integers. An InterBase DATE data type includes information about year, month, day of the month, and time.

This chapter discusses how to SELECT, INSERT, and UPDATE dates from tables in SQL applications using the following isc call interface routines:

- **isc_decode_date()** to convert the InterBase internal date format to the C time structure
- **isc_encode_date()** to convert the C time structure to the internal InterBase date format

The chapter also discusses how to use the CAST() function to translate a DATE data type into a CHARACTER data type and back again, and how to use the DATE literals, NOW and TODAY when selecting and inserting dates.

Note InterBase does not directly support SQL-92 DATE, TIME, and TIMESTAMP data types.

Selecting Dates

To select a date from a table, and convert it to a form usable in a C language program, follow these steps:

1. Create a host variable for a C time structure. Most C and C++ compilers provide a typedef declaration, *tm*, for the C time structure in the *time.h* header file. The following C code includes that header file, and declares a variable of type *tm*:

```
#include <time.h>;
. . .
struct tm hire_time;
. . .
```

Note To create host-language time structures in languages other than C and C++, see the host-language reference manual.

2. Create a host variable of type ISC_QUAD. For example, the host-variable declaration might look like this:

```
ISC_QUAD hire_date;
```

   The ISC_QUAD structure is automatically declared for programs when they are preprocessed with **gpre**, but the programmer must declare actual host-language variables of type ISC_QUAD.

3. Retrieve a date from a table into the ISC_QUAD variable. For example,

```
EXEC SQL
   SELECT LAST_NAME, FIRST_NAME, DATE_OF_HIRE
      INTO :lname, :fname, :hire_date
      FROM EMPLOYEE
      WHERE LAST_NAME = "Smith" AND FIRST_NAME = "Margaret";
```

   Convert the ISC_QUAD variable into a numeric Unix format with the InterBase function, **isc_decode_date()**. This function is automatically declared for programs when they are preprocessed with **gpre**. **isc_decode_date()** requires two parameters, the address of the ISC_QUAD host-language variable, and the address of the *tm* host-language variable. For example, the following code fragment coverts *hire_date* to *hire_time*:

```
isc_decode_date(&hire_date, &hire_time);
```

Inserting Dates

To insert a date in a table, it must be converted from the host-language format into InterBase format, and then stored. To perform the conversion and insertion in a C program, follow these steps:

1. Create a host variable for a C time structure. Most C and C++ compilers provide a typedef declaration, *tm*, for the C time structure in the *time.h* header file. The following C code includes that header file, and declares a *tm* variable, *hire_time*:

```
#include <time.h>;
. . .
struct tm hire_time;
. . .
```

To create host-language time structures in languages other than C and C++, see the host-language reference manual.

2. Create a host variable of type ISC_QUAD, for use by InterBase. For example, the host-variable declaration might look like this:

   ```
   ISC_QUAD mydate;
   ```

   The ISC_QUAD structure is automatically declared for programs when they are preprocessed with **gpre**, but the programmer must declare actual host-language variables of type ISC_QUAD.

3. Put date and time information into *hire_time*.

4. Use the InterBase **isc_encode_date()** function to convert the information in *hire_time* into InterBase internal format and store that formatted information in the ISC_QUAD host variable (*hire_date* in the example). This function is automatically declared for programs when they are preprocessed with **gpre**. **isc_encode_date()** requires two parameters, the address of the Unix time structure, and the address of the ISC_QUAD host-language variable.

   For example, the following code converts *hire_time* to *hire_date*:

   ```
   isc_encode_date(&hire_time, &hire_date);
   ```

5. Insert the date into a table. For example,

   ```
   EXEC SQL
      INSERT INTO EMPLOYEE (EMP_NO, DEPARTMENT, DATE_OF_HIRE)
         VALUES (:emp_no, :deptname, :hire_date);
   ```

Updating Dates

To update a date in a table, it must be converted from the host-language format into InterBase format, and then stored. To convert a host variable into InterBase format, see “Inserting Dates,” in this chapter. The actual update is performed using an UPDATE statement. For example,

```
EXEC SQL
   UPDATE EMPLOYEE
   SET DATE_OF_HIRE = :hire_date
   WHERE DATE_OF_HIRE < "1 JAN 1994"
```

Using CAST( ) to Convert Dates

The built-in CAST() function can be used in SELECT statements to translate a DATE data type into a CHARACTER or NUMERIC data type, or to translate CHARACTER and NUMERIC data types into DATE data types. Typically, CAST() is used in the WHERE clause to compare different data types. The syntax for CAST() is:

```
CAST (<value> AS <datatype>)
```

In the following WHERE clause, CAST() is used to translate a CHAR data type, INTERVIEW_DATE, to a DATE data type to compare against a DATE data type, HIRE_DATE:

```
. . . WHERE HIRE_DATE = CAST(INTERVIEW_DATE AS DATE);
```

In the next example, CAST() is used to translate a DATE data type into a CHAR data type:

```
. . . WHERE CAST(HIRE_DATE AS CHAR) = INTERVIEW_DATE;
```

CAST() also can be used to compare columns with different data types in the same table, or across tables. For more information about CAST(), see Chapter 6: "Working With Data."

Using Date Literals

InterBase supports two date literals, NOW, and TODAY. *Date literals* are string values entered between quotation marks that can be interpreted as date values for SELECT, INSERT, and UPDATE operations. NOW is a date literal that combines today's date and time in InterBase format. TODAY is today's date with time information set to zero.

In SELECT, NOW and TODAY can be used in the search condition of a WHERE clause to restrict the data retrieved:

```
EXEC SQL
   SELECT * FROM CROSS_RATE WHERE UPDATE_DATE = "TODAY";
```

In INSERT and UPDATE, NOW and TODAY can be used to enter date and time values instead of relying on **isc** calls to convert C dates to InterBase dates:

```
EXEC SQL
   INSERT INTO CROSS_RATE VALUES(:from, :to, :rate, "NOW");
```

```
EXEC SQL
   UPDATE CROSS_RATE
      SET CONV_RATE = 1.75,
      SET UPDATE_DATE = "NOW"
      WHERE FROM_CURRENCY = "POUND" AND TO_CURRENCT = "DOLLAR"
         AND UPDATE_DATE < "TODAY";
```

CHAPTER 8

Working With BLOB Data

This chapter describes the binary large object (BLOB) data type and how to work with it using SQL, DSQL, and the InterBase API. Depending on your particular application, you might need to read all or only part of this chapter.

For example, if you plan to convert BLOB data from one data type to another, such as from one bitmapped graphic format to another or from the MIDI sound format to the Wave format, you need to read the entire chapter, including the information about BLOB filters and BLOB control.

Important BLOB filters are not available on NetWare servers.

If you plan to read and write BLOB data without any conversion, you can read the first few sections of the chapter, including:

- What is a BLOB?
- How are BLOB Data Stored?
- BLOB Subtypes
- BLOB Database Storage
- BLOB Segment Length
- Accessing BLOB Data with SQL
  - Selecting BLOB Data
  - Inserting BLOB Data
  - Updating BLOB Data
  - Deleting BLOB Data
- Accessing BLOB Data Using API Calls

You do not need to read the information on BLOB filters, unless you determine that the information is required for your application.

What is a BLOB?

A BLOB is a binary large object that cannot easily be stored in a database as one of the standard data types. You can use a BLOB to store large amounts of data of various types, including:

- Bitmapped images
- Sounds
- Video segments
- Text

InterBase support of BLOB data provides all the advantages of a database management system, including transaction control, maintenance, and access using the standard relational operators and data manipulation statements. BLOB data is stored in the database. Other systems only store pointers in the database to non-database files. InterBase stores the actual BLOB data in the database, and establishes a unique identification handle in the appropriate table to point to the database location of the BLOB. By maintaining the BLOB data within the database, InterBase greatly improves access to and management of the data.

The combination of true database management of BLOB data and support for a variety of data types makes InterBase BLOB support ideal for transaction-intensive multimedia applications. For example, InterBase is an ideal platform for interactive kiosk applications that might provide hundreds or thousands of product descriptions, photographs, and video clips, in addition to point-of-sale and order processing capabilities.

How are BLOB Data Stored?

BLOB is the InterBase data type that represents various objects, such as bitmapped images, sound, video, and text. Before you store these items in the database, you create or manage them as platform- or product-specific files or data structures, such as:

- TIFF, PICT, .BMP, .WMF, .GEM, TARGA or other bitmapped or vector-graphic files.
- MIDI or .WAV sound files.
- Audio Video Interleaved Format (.AVI) or QuickTime video files.
- ASCII, .MIF, .DOC, .WPx or other text files.

- CAD files.

You must programmatically load these files from memory into the database, as you do any other host-language data items or records you intend to store in InterBase.

BLOB Subtypes

Although you manage BLOB data in the same way you manage other data types, InterBase provides more flexible data typing rules for BLOB data. Because there are many native data types that you can define as BLOB data, InterBase treats them somewhat generically and allows you to define your own data type, known as a *subtype*. Also, InterBase provides two standard subtypes with which you can characterize BLOB data:

- Subtype 0, an unstructured subtype, generally applied to binary data or data of an indeterminate type.
- Subtype 1, applied to TEXT.

You can specify user-defined subtypes as negative numbers between -1 and -32,678. Positive integers are reserved for InterBase subtypes.

For example, the following statement defines three BLOB columns: BLOB1 with subtype 0 (the default), BLOB2 with subtype 1 (TEXT), and BLOB3 with user-defined subtype -1:

```
EXEC SQL CREATE TABLE TABLE2
(
   BLOB1 BLOB,
   BLOB2 BLOB SUB_TYPE 1,
   BLOB3 BLOB SUB_TYPE -1
);
```

To specify both a default segment length and a subtype when creating a BLOB column, use the SEGMENT SIZE option after the SUB_TYPE option. For example:

```
EXEC SQL CREATE TABLE TABLE2
(
   BLOB1 BLOB SUB_TYPE 1 SEGMENT SIZE 100;
);
```

The only rule InterBase enforces over these user-defined subtypes is that, when converting a BLOB from one subtype to another, those subtypes must be compatible. InterBase does not otherwise enforce subtype integrity.

BLOB Database Storage

Because BLOB data are typically large, variably-sized objects of binary or text data, InterBase stores them most efficiently using a method of segmentation. It would be an inefficient use of disk space to store each BLOB as one contiguous mass. Instead, InterBase stores each BLOB in segments that are indexed by a handle that InterBase generates when you create the BLOB. This handle is known as the *BLOB ID* and is a quadword (64-bit) containing a unique combination of table identifier and BLOB identifier.

The BLOB ID for each BLOB is stored in its appropriate field in the table record. The BLOB ID points to the first segment of the BLOB, or to a page of pointers, each of which points to a segment of one or more BLOB fields. You can retrieve the BLOB ID by executing a SELECT statement that specifies the BLOB as the target, as in the following example:

```
EXEC SQL
   DECLARE BLOBDESC CURSOR FOR
      SELECT GUIDEBOOK
      FROM TOURISM
      WHERE STATE = "CA";
```

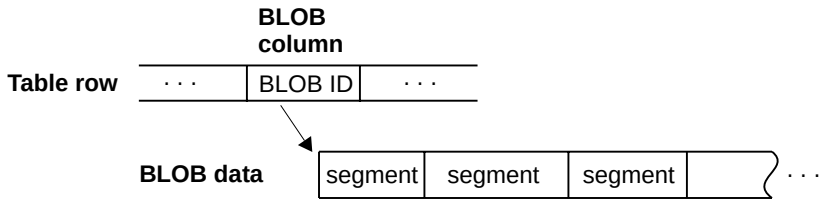
You can define BLOB columns the same way you define non-BLOB columns. For example, the following SQL code creates a table with a BLOB column called PROJ_DESC:

```
CREATE TABLE PROJECT
(
   PROJ_ID PROJNO NOT NULL,
   PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
   PROJ_DESC BLOB SUBTYPE 1 SEGMENT SIZE 80,
   TEAM_LEADER EMPNO,
   PRODUCT PRODTYPE,
   ...
);
```

Note that in the preceding example the subtype parameter is set to 1, which denotes a TEXT BLOB, and the segment size is set to 80 bytes.

The following diagram shows the relationship between a BLOB column containing a BLOB ID and the BLOB data referenced by the BLOB ID:

Figure 8-1: Relationship of a BLOB ID to BLOB Segments in a Database



Rather than store BLOB data directly in the table, InterBase stores a BLOB ID in each row of the table. The BLOB ID, a unique number, points to the first segment of the BLOB data that is stored elsewhere in the database, in a series of segments. When an application creates a BLOB, it must write data to that BLOB a segment at a time. Similarly, when an application reads of BLOB, it reads a segment at a time. Because most BLOB data are large objects, most BLOB management is performed with loops in the application code.

BLOB Segment Length

When you define a BLOB in a table, you can specify, in the BLOB definition statement, the expected size of BLOB segments that are to be written to the column. The segment length you define for a BLOB column specifies the maximum number of bytes that an application is expected to write to or read from any BLOB in the column. The default segment length is 80. For example, the following column declaration creates a BLOB with a segment length of 120:

```
EXEC SQL CREATE TABLE TABLE2
(
   BLOB1 BLOB SEGMENT SIZE 120;
);
```

InterBase uses the segment length setting to determine the size of an internal buffer to which it writes BLOB segment data. Normally, you should not attempt to write segments larger than the segment length you defined in the table; doing so may result in a buffer overflow and possible memory corruption.

Specifying a segment size of *n* guarantees that no more than *n* number of bytes are read or written in a single BLOB operation. With some types of operations, for instance, with SELECT, INSERT, and UPDATE operations, you can read or write BLOB segments of varying length.

In the following example of an INSERT CURSOR statement, specify the segment length in a host language variable, *segment_length*, as follows:

```
EXEC SQL
   INSERT CURSOR BCINS VALUES (:write_segment_buffer INDICATOR
:segment_length);
```

For more information about the syntax of the INSERT CURSOR statement, see the *Language Reference*.

Overriding Segment Length

You can override the segment length setting by including the MAXIMUM_SEGMENT option in a DECLARE CURSOR statement. For example, the following BLOB INSERT cursor declaration overrides the segment length that was defined for the field, BLOB2, increasing it to 1024:

```
EXEC SQL
   DECLARE BCINS CURSOR FOR INSERT BLOB BLOB2 INTO TABLE 2
   MAXIMUM_SEGMENT 1024;
```

Note By overriding the segment length setting, you affect only the segment size for the cursor, not for the column, or for other cursors. Other cursors using the same BLOB column maintain the original segment size that was defined in the column definition, or can specify their own overrides.

The segment length setting does not affect InterBase system performance. Choose the segment length most convenient for the specific application. The largest possible segment length is 65,535 bytes (64K).

Accessing BLOB Data With SQL

InterBase supports SELECT, INSERT, UPDATE, and DELETE operations on BLOB data. The following sections contain brief discussions of example programs. These programs illustrate how to perform standard SQL operations on BLOB data.

Selecting BLOB Data

The following example program, *emblob1.e*, selects BLOB data from the GUIDEBOOK column of the TOURISM table contained in the example database, *atlas.gdb:*

1. Declare host-language variables to store the BLOB ID, the BLOB segment data, and the length of segment data:

```
EXEC SQL
```

```
   BEGIN DECLARE SECTION;
      BASED ON TOURISM.GUIDEBOOK blob_id;
      BASED ON TOURISM.GUIDEBOOK.SEGMENT blob_segment_buf;
      BASED ON TOURISM.STATE state;
      unsigned short blob_seg_len;
EXEC SQL
   END DECLARE SECTION;
```

The BASED ON . . . SEGMENT syntax declares a host-language variable, *blob_segment_buf*, that is large enough to hold a BLOB segment during a FETCH operation. For more information about the BASED ON statement, see the *Language Reference*.

2. Declare a table cursor to select the desired BLOB column, in this case the GUIDEBOOK column:

```
EXEC SQL
   DECLARE TC CURSOR FOR
      SELECT STATE, GUIDEBOOK
      FROM TOURISM
      WHERE STATE = "CA";
```

3. Declare a BLOB read cursor. A *BLOB read cursor* is a special cursor used for reading BLOB segments:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   READ BLOB GUIDEBOOK
   FROM TOURISM;
```

The segment length of the GUIDEBOOK BLOB column is defined as 60, so BLOB cursor, BC, reads a maximum of 60 bytes at a time.

To override the segment length specified in the database schema for GUIDEBOOK, use the MAXIMUM_SEGMENT option. For example, the following code restricts each BLOB read operation to a maximum of 40 bytes, and SQLCODE is set to 101 to indicate when only a portion of a segment has been read:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   READ BLOB GUIDEBOOK
   FROM TOURISM
   MAXIMUM_SEGMENT 40;
```

Note No matter what the segment length setting is, only one segment is read at a time.

4. Open the table cursor and fetch a row of data containing a BLOB:

```
EXEC SQL
   OPEN TC;
```

```
EXEC SQL
   FETCH TC INTO :state, :blob_id;
```

The FETCH statement fetches the STATE and GUIDEBOOK columns into host variables, *state*,and *blob_id*, respectively.

5. Open the BLOB read cursor using the BLOB ID stored in the *blob_id* variable, and fetch the first segment of BLOB data:

```
EXEC SQL
   OPEN BC USING :blob_id;
EXEC SQL
   FETCH BC INTO :blob_segment_buf:blob_seg_len;
```

When the FETCH operation completes, *blob_segment_buf* contains the first segment of the BLOB, and *blob_seg_len* contains the segment's length, which is the number of bytes copied into *blob_segment_buf*.

6. Fetch the remaining segments in a loop. SQLCODE should be checked each time a fetch is performed. An error code of 100 indicates that all of the BLOB data has been fetched. An error code of 101 indicates that the segment contains additional data:

```
while (SQLCODE != 100 || SQLCODE == 101)
{
   printf("%*.*s", blob_seg_len, blob_seg_len, blob_segment_buf);
   EXEC SQL
      FETCH BC INTO :blob_segment_buf:blob_seg_len;
}
```

InterBase produces an error code of 101 when the length of the segment buffer is less than the length of a particular segment.

For example, if the length of the segment buffer is 40 and the length of a particular segment is 60, the first FETCH produces an error code of 101 indicating that data remains in the segment. The second FETCH reads the remaining 20 bytes of data, and produces an SQLCODE of 0, indicating that the next segment is ready to be read, or 100 if this was the last segment in the BLOB.

7. Close the BLOB read cursor:

```
EXEC SQL
   CLOSE BC;
```

8. Close the table cursor:

```
EXEC SQL
   CLOSE TC;
```

Inserting BLOB Data

The following program, *emblob2.e*, inserts BLOB data into the GUIDEBOOK column of the TOURISM table contained in the example database, *atlas.gdb:*

1. Declare host-language variables to store the BLOB ID, BLOB segment data, and the length of segment data:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED ON TOURISM.GUIDEBOOK blob_id;
      BASED ON TOURISM.GUIDEBOOK.SEGMENT blob_segment_buf;
      BASED ON TOURISM.STATE state;
      unsigned short blob_seg_len;
EXEC SQL
   END DECLARE SECTION;
```

   The BASED ON . . . SEGMENT syntax declares a host-language variable, *blob_segment_buf*, that is large enough to hold a BLOB segment during a FETCH operation. For more information about the BASED ON directive, see the *Language Reference.*

2. Declare a BLOB insert cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR INSERT BLOB GUIDEBOOK INTO TOURISM;
```

3. Open the BLOB insert cursor and specify the host variable in which to store the BLOB ID:

```
EXEC SQL
   OPEN BC INTO :blob_id;
```

4. Store the segment data in the segment buffer, *blob_segment_buf*, calculate the length of the segment data, and use an INSERT CURSOR statement to write the segment:

```
sprintf(blob_segment_buf, "My segment number is: %d\n", i);
blob_segment_len = strlen(blob_segment_buf);

EXEC SQL
   INSERT CURSOR BC VALUES (:blob_segment_buf:blob_segment_len);
```

   Repeat these steps in a loop until you have written all BLOB segments.

5. Close the BLOB insert cursor:

```
EXEC SQL
   CLOSE BC;
```

6. Use an INSERT statement to insert a new row containing the BLOB into the TOURISM table:

```
EXEC SQL
   INSERT INTO TOURISM (STATE,GUIDEBOOK) VALUES ("CA",:blob_id);
```

7. Commit the changes to the database:

```
EXEC SQL
   COMMIT;
```

Updating BLOB Data

You cannot update a BLOB directly. You must create a new BLOB, read the old BLOB data into a buffer where you can edit or modify it, then write the modified data to the new BLOB.

Create a new BLOB by following these steps:

1. Declare a BLOB insert cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR INSERT BLOB GUIDEBOOK INTO TOURISM;
```

2. Open the BLOB insert cursor and specify the host variable in which to store the BLOB ID:

```
EXEC SQL
   OPEN BC INTO :blob_id;
```

3. Store the old BLOB segment data in the segment buffer *blob_segment_buf*, calculate the length of the segment data, perform any modifications to the data, and use an INSERT CURSOR statement to write the segment:

```
/* Programmatically read the first/next segment of the old BLOB
* segment data into blob_segment_buf; */

EXEC SQL
   INSERT CURSOR BC VALUES (:blob_segment_buf:blob_segment_len);
```

   Repeat these steps in a loop until you have written all BLOB segments.

4. Close the BLOB insert cursor:

```
EXEC SQL
   CLOSE BC;
```

5. When you have completed creating the new BLOB, issue an UPDATE statement to replace the old BLOB in the table with the new one, as in the following example:

```
EXEC SQL UPDATE TOURISM
   SET
      GUIDEBOOK = :blob_id;
   WHERE CURRENT OF TC;
```

Note The TC table cursor points to a target row established by declaring the cursor and then fetching the row to update.

To modify a text BLOB using this technique, you might read an existing BLOB field into a host-language buffer, modify the data, then write the modified buffer over the existing field data with an UPDATE statement.

Deleting BLOB Data

There are two methods to delete a BLOB. The first method is to delete the row containing the BLOB. The second method is to update the row and set the BLOB column to NULL, or to the BLOB ID of a different BLOB (e.g., the new BLOB created to update the data of an existing BLOB). For example, the following statement deletes current BLOB data in the GUIDEBOOK column of the TOURISM table by setting it to NULL:

```
EXEC SQL UPDATE TOURISM
   SET
      GUIDEBOOK = NULL;
   WHERE CURRENT OF TC;
```

BLOB data is not immediately deleted when DELETE is specified. The actual delete operation occurs when InterBase performs version cleanup. The following code fragment illustrates how to recover space after deleting a BLOB:

```
EXEC SQL
   UPDATE TABLE SET BLOB_COLUMN = NULL WHERE ROW = :myrow;
EXEC SQL
   COMMIT;
/* wait for all active transactions to finish */
/* force a sweep of the database */
```

When InterBase performs garbage collection on old versions of a record, it verifies whether or not recent versions of the record reference the BLOB ID. If the record does not reference the BLOB ID, InterBase cleans up the BLOB.

Accessing BLOB Data With API Calls

In addition to accessing BLOB data using SQL as described in this chapter, the InterBase API provides routines for accessing BLOB data. The following API calls are provided for accessing and managing BLOB data:

Table 8-1: API BLOB Calls

Function	Description
isc_blob_default_desc()	Loads a BLOB descriptor data structure with default information about a BLOB.
isc_blob_gen_bpb()	Generates a BLOB parameter buffer (BPB) from source and target BLOB descriptors to allow dynamic access to BLOB subtype and character set information.
isc_blob_info()	Returns information about an open BLOB.
isc_blob_lookup_desc()	Looks up and stores into a BLOB descriptor the subtype, character set, and segment size of a BLOB.
isc_blob_set_desc()	Sets the fields of a BLOB descriptor to values specified in parameters to **isc_blob_set_desc()**.
isc_cancel_blob()	Discards a BLOB and frees internal storage.
isc_close_blob()	Closes an open BLOB.
isc_create_blob2()	Creates a context for storing a BLOB, opens the BLOB for write access, and optionally specifies a filter to be used to translate the BLOB data from one subtype to another.
isc_get_segment()	Reads a segment from an open BLOB.
isc_open_blob2()	Opens an existing BLOB for retrieval and optional filtering.
isc_put_segment()	Writes a BLOB segment.

For details on using the API calls to access BLOB data, see the *API Guide.*

Filtering BLOB Data

An understanding of BLOB subtypes is particularly important when working with BLOB filters. A *BLOB filter* is a routine that translates BLOB data from one subtype to another. InterBase includes a set of special internal BLOB filters that convert from subtype 0 to subtype 1 (TEXT), and from subtype 1 (TEXT) to sub-

type 0. In addition to using these standard filters, you can write your own external filters to provide special data translation. For example, you might develop a filter to translate bitmapped images from one format to another.

Important BLOB filters are available for databases residing on all InterBase server platforms except NetWare, where BLOB filters cannot be created or used.

Using the Standard InterBase Text Filters

The standard InterBase filters convert BLOB data of subtype 0, or any InterBase system type, to subtype 1 (TEXT).

When a text filter is being used to read data from a BLOB column, it modifies the standard InterBase behavior for supplying segments. Regardless of the actual nature of the segments in the BLOB column, the text filter enforces the rule that segments must end with a newline character (\n).

The text filter returns all the characters up to and including the first newline as the first segment, the next characters up to and including the second newline as the second segment, and so on.

Tip To convert any non-text subtype to TEXT, declare its FROM subtype as subtype 0 and its TO subtype as subtype 1.

Using an External BLOB Filter

Unlike the standard InterBase filters that convert between subtype 0 and subtype 1, an external BLOB filter is generally part of a library of routines you create and link to your application.

To use an external filter, you must first write it, compile and link it, then declare it to the database that contains the BLOB data you want processed.

Declaring an External Filter to the Database

To declare an external filter to a database, use the DECLARE FILTER statement. For example, the following statement declares the filter, SAMPLE:

```
EXEC SQL
   DECLARE FILTER SAMPLE
      INPUT_TYPE -1 OUTPUT_TYPE -2
      ENTRY_POINT "FilterFunction"
      MODULE_NAME "filter.dll";
```

In the example, the filter's input subtype is defined as -1 and its output subtype as -2. In this example, INPUT_TYPE specifies lowercase text and

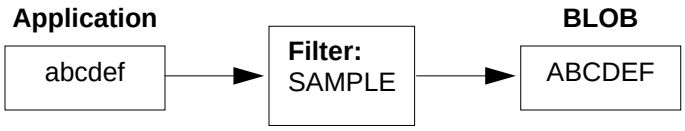
OUTPUT_TYPE specifies uppercase text. The purpose of filter, SAMPLE, therefore, is to translate BLOB data from lowercase text to uppercase text.

The ENTRY_POINT and MODULE_NAME parameters specify the external routine that InterBase calls when the filter is invoked. The MODULE_NAME parameter specifies *filter.dll*, the dynamic link library containing the filter's executable code. The ENTRY_POINT parameter specifies the entry point into the DLL. The example shows only a simple file name. It is good practice to specify a fully-qualified path name, since users of your application need to load the file.

Reading and Writing BLOB Data Using a Filter

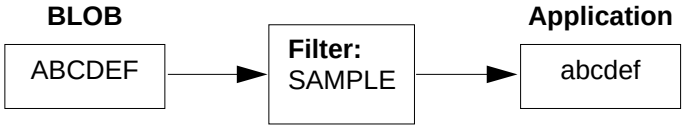
The following illustration shows the default behavior of the SAMPLE filter that translates from lowercase text to uppercase text.

Figure 8-2: Filtering from Lowercase to Uppercase



Similarly, when reading data, the SAMPLE filter can easily read BLOB data of subtype -2, and translate it to data of subtype -1.

Figure 8-3: Filtering from Uppercase to Lowercase



Invoking a Filter in an Application

To invoke a filter in an application, use the FILTER option when declaring a BLOB cursor. Then, when the application performs operations using the cursor, InterBase automatically invokes the filter.

For example, the following INSERT cursor definition specifies that the filter, SAMPLE, is to be used in any operations involving the cursor, BCINS1:

```
EXEC SQL
   DECLARE BCINS1 CURSOR FOR
      INSERT BLOB BLOB1 INTO TABLE1
      FILTER FROM -1 TO -2;
```

When InterBase processes this declaration, it searches a list of filters defined in the current database for a filter with matching FROM and TO subtypes. If such a filter exists, InterBase invokes it during BLOB operations that use the cursor, BCINS1. If InterBase cannot locate a filter with matching FROM and TO subtypes, it returns an error to the application.

Writing an External BLOB Filter

If you choose to write your own filters, you must have a detailed understanding of the data types you plan to translate. As mentioned elsewhere in this chapter, InterBase does not do strict data type checking on BLOB data, but does enforce the rule that BLOB source and target subtypes must be compatible. Maintaining and enforcing this compatibility is your responsibility.

Filter Types

Filters can be divided into two types: filters that convert data one segment at a time, and filters that convert data many segments at a time.

The first type of filter reads a segment of data, converts it, and supplies it to the application a segment at a time.

The second type of filter might read all the data and do all the conversion when the BLOB read cursor is first opened, and then simulate supplying data a segment at a time to the application.

If timing is an issue for your application, you should carefully consider these two types of filters and which might better serve your purpose.

Read-only and Write-only Filters

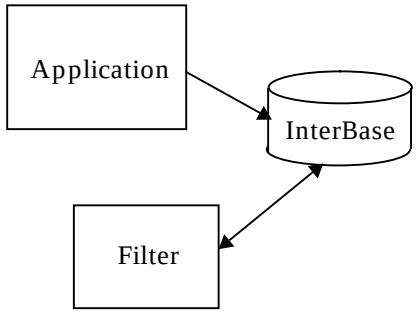
Some filters may only support reading from or writing to a BLOB, but not both operations. If you attempt to use a BLOB filter for an operation that it does not support, InterBase returns an error to the application.

Defining the Filter Function

When writing your filter, you must include an entry point, known as a *filter function* in the declaration section of the program. InterBase calls the filter function when your application performs a BLOB access operation. All communication

between InterBase and the filter is through the filter function. The filter function itself may call other functions that comprise the filter executable.

Figure 8-4: Filter Interaction with an Application and a Database



Declare the name of the filter function and the name of the filter executable with the ENTRY_POINT and MODULE_NAME parameters of the DECLARE FILTER statement.

A filter function must have the following declaration *calling sequence*:

```
filter_function_name(short action, isc_blob_ctl control);
```

The parameter, *action*, is one of eight possible action macro definitions and the parameter, *control*, is an instance of the *isc_blob_ctl* BLOB control structure, defined in the InterBase header file *ibase.h*. These parameters are discussed later in this chapter.

The following listing of a skeleton filter declares the filter function, *jpeg_filter*:

```
#include <ibase.h>

#define SUCCESS 0
#define FAILURE 1

ISC_STATUS jpeg_filter(short action, isc_blob_ctl control)
{
   ISC_STATUS status = SUCCESS;

   switch (action)
   {
   case isc_blob_filter_open:
      . . .
      break;
   case isc_blob_filter_get_segment:
      . . .
      break;
```

```
    case isc_blob_filter_create:
       . . .
       break;
    case isc_blob_filter_put_segment:
       . . .
       break;
    case isc_blob_filter_close:
       . . .
       break;
    case isc_blob_filter_alloc:
       . . .
       break;
    case isc_blob_filter_free:
       . . .
       break;
    case isc_blob_filter_seek:
       . . .
       break;
    default:
       status = isc_uns_ext /* unsupported action value */
       . . .
       break;
    }
return status;
}
```

InterBase passes one of eight possible actions to the filter function, *jpeg_filter*, by way of the *action* parameter, and also passes an instance of the BLOB control structure, *isc_blob_ctl*, by way of the parameter *control*.

The ellipses (. . .) in the previous listing represent code that performs some operations based on each action, or event, that is listed in the case statement. Each action is a particular event invoked by a database operation the application might perform. For more information, see "Programming Filter Function Actions," in this chapter.

The *isc_blob_ctl* BLOB control structure provides the fundamental data exchange between InterBase and the filter. For more information on the BLOB control structure, see "Defining the BLOB Control Structure," in this chapter.

Defining the BLOB Control Structure

The BLOB control structure, *isc_blob_ctl*, provides the fundamental method of data exchange between InterBase and a filter. The declaration for the *isc_blob_ctl* control structure is in the InterBase include file, *ibase.h*.

The *isc_blob_ctl* structure is used in two ways:

1. When the application performs a BLOB access operation, InterBase calls the filter function and passes it an instance of *isc_blob_ctl*.

2. Internal filter functions can pass an instance of *isc_blob_ctl* to internal InterBase access routines.

In either case, the purpose of certain *isc_blob_ctl* fields depends on the action being performed.

For example, when an application attempts a BLOB INSERT, InterBase passes an *isc_blob_filter_put_segment* action to the filter function. The filter function passes an instance of the control structure to InterBase. The *ctl_buffer* of the structure contains the segment data to be written, as specified by the application in its BLOB INSERT statement. Because the buffer contains information to pass into the filter function, it is called an *IN* field. The filter function should include instructions in the case statement under the *isc_blob_filter_put_segment* case for performing the write to the database.

In a different case, for instance when an application attempts a FETCH operation, the case of an *isc_blob_filter_get_segment* action should include instructions for filling *ctl_buffer* with segment data from the database to return to the application. In this case, because the buffer is used for filter function output, it is called an *OUT* field.

The following table describes each of the fields in the *isc_blob_ctl* BLOB control structure, and whether they are used for filter function input (IN), or output (OUT).

Table 8-2: *isc_blob_ctl* Structure Field Descriptions

Field Name	Description
*(*ctl_source)()*	Pointer to the internal InterBase BLOB access routine. (IN)
**ctl_source_handle*	Pointer to an instance of *isc_blob_ctl* to be passed to the internal InterBase BLOB access routine. (IN)
ctl_to_sub_type	Target subtype. Information field. Provided to support multi-purpose filters that can perform more than one kind of translation. This field and the next one enable such a filter to decide which translation to perform. (IN)
ctl_from_sub_type	Source subtype. Information field. Provided to support multi-purpose filters that can perform more than one kind of translation. This field and the previous one enable such a filter to decide which translation to perform. (IN)
ctl_buffer_length	For *isc_blob_filter_put_segment*, field is an IN field that contains the length of the segment data contained in *ctl_buffer*. For *isc_blob_filter_get_segment*, field is an IN field set to the size of the buffer pointed to by *ctl_buffer*, which is used to store the retrieved BLOB data.

Table 8-2: *isc_blob_ctl* Structure Field Descriptions (Continued)

Field Name	Description
ctl_segment_length	Length of the current segment. For *isc_blob_filter_put_segment*, this field is not used. For *isc_blob_filter_get_segment*, the field is an OUT field set to the size of the retrieved segment (or partial segment, in the case when the buffer length *ctl_buffer_length* is less than the actual segment length).
ctl_bpb_length	Length of the BLOB parameter buffer. Reserved for future enhancement.
**ctl_bpb*	Pointer to a BLOB parameter buffer. Reserved for future enhancement.
**ctl_buffer*	Pointer to a segment buffer. For *isc_blob_filter_put_segment*, field is an IN field that contains the segment data. For *isc_blob_filter_get_segment*, the field is an OUT field the filter function fills with segment data for return to the application.
ctl_max_segment	Length of longest segment in the BLOB. Initial value is 0. The filter function sets this field. This field is informational only.
ctl_number_segments	Total number of segments in the BLOB. Initial value is 0. The filter function sets this field. This field is informational only.
ctl_total_length	Total length of the BLOB. Initial value is 0. The filter function sets this field. This field is informational only.
**ctl_status*	Pointer to the InterBase status vector. (OUT)
ctl_data[8]	8-element array of application-specific data. Use this field to store resource pointers, such as memory pointers and file handles created by the *isc_blob_filter_open* handler, for example. Then, the next time the filter function is called, the resource pointers will be available for use. (IN/OUT)

Setting Control Structure Information Field Values

The *isc_blob_ctl* structure contains three fields that store information about the BLOB currently being accessed: *ctl_max_segment*, *ctl_number_segments*, and *ctl_total_length*.

You should attempt to maintain correct values for these fields in the filter function, whenever possible. Depending on the purpose of the filter, maintaining correct values for the fields is not always possible. For example, a filter that compresses data on a segment-by-segment basis cannot determine the size of *ctl_max_segment* until it processes all segments.

These fields are informational only. InterBase does not use the values of these fields in internal processing.

Programming Filter Function Actions

When an application performs a BLOB access operation, InterBase passes a corresponding action message to the filter function by way of the *action* parameter. There are eight possible actions, each of which results from a particular access operation. The following list of action macro definitions are declared in the *ibase.h* file:

```
#define isc_blob_filter_open        0
#define isc_blob_filter_get_segment 1
#define isc_blob_filter_close       2
#define isc_blob_filter_create      3
#define isc_blob_filter_put_segment 4
#define isc_blob_filter_alloc       5
#define isc_blob_filter_free        6
#define isc_blob_filter_seek        7
```

The following table describes the BLOB access operation that corresponds to each action:

Table 8-3: BLOB Access Operations

Action	Invoked when . . .	Use to . . .
isc_blob_ filter_open	Application opens a BLOB READ cursor.	• Set the information fields of the BLOB control structure. • Perform initialization tasks, such as allocating memory or opening temporary files. • Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.
isc_blob_filter_ get_segment	Application executes a BLOB FETCH statement.	• Set the *ctl_buffer* and *ctl_segment_length* fields of the BLOB control structure to contain a segment's worth of translated data on the return of the filter function. • Perform the data translation if the filter processes the BLOB segment-by-segment. • Set the status variable. The value of the status variable becomes the filter function's return value.
isc_blob_filter_ close	Application closes a BLOB cursor.	• Perform exit tasks, such as freeing allocated memory, closing, or removing temporary files.

Table 8-3: BLOB Access Operations (Continued)

Action	Invoked when . . .	Use to . . .
isc_blob_filter_create	Application opens a BLOB INSERT cursor.	• Set the information fields of the BLOB control structure. • Perform initialization tasks, such as allocating memory or opening temporary files. • Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.
isc_blob_filter_put_segment	Application executes a BLOB INSERT statement.	• Perform the data translation on the segment data passed in through the BLOB control structure. • Write the segment data to the database. If the translation process changes the segment length, the new value must be reflected in the values passed to the writing function. • Set the status variable. The value of the status variable becomes the filter function's return value.
isc_blob_filter_alloc	InterBase initializes filter processing. Not a result of a particular application action.	• Set the information fields of the BLOB control structure. • Perform initialization tasks, such as allocating memory or opening temporary files. • Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.
isc_blob_filter_free	InterBase ends filter processing. Not a result of a particular application action.	• Perform exit tasks, such as freeing allocated memory, closing, or removing temporary files.
isc_blob_filter_seek	Reserved for internal filter use; not used by external filters.	

Tip Store resource pointers, such as memory pointers and file handles created by the *isc_blob_filter_open* handler, in the *ctl_data* field of the *isc_blob_ctl* BLOB control structure. Then, the next time the filter function is called, the resource pointers are still available.

Testing the Filter Function Status Return Value

The filter function must return an integer indicating the status of the operation it performed. You can have the function return any InterBase status value returned by an internal InterBase routine.

In certain filter applications, a filter function may have to supply status values directly. The following table lists status values that apply particularly to BLOB processing:

Table 8-4: BLOB Filter Status Values

Macro Constant	Value	Meaning
SUCCESS	0	Indicates the filter action has been handled successfully. On a BLOB read (*isc_blob_filter_get_segment*) operation, indicates that the entire segment has been read.
FAILURE	1	Indicates an unsuccessful operation. In most cases, a status more specific to the error is returned.
isc_uns_ext	See *ibase.h*	Indicates that the attempted action is unsupported by the filter. For example, a read-only filter would return *isc_uns_ext* for an *isc_blob_filter_put_segment* action.
isc_segment	See *ibase.h*	Returned during a BLOB read operation. Indicates that the supplied buffer is not large enough to contain the remaining bytes in the current segment. In this case, only *ctl_buffer_length* bytes are copied, and the remainder of the segment must be obtained through additional *isc_blob_filter_get_segment* calls.
isc_segstr_eof	See *ibase.h*	Returned during a BLOB read operation. Indicates that the end of the BLOB has been reached; there are no additional segments remaining to be read.

For more information about InterBase status values, see the *Language Reference*.

CHAPTER 9

Using Arrays

InterBase supports arrays of most data types. Using an array enables multiple data items to be stored in a single column. InterBase can treat an array as a single unit, or as a series of separate units, called *slices*. Using an array is appropriate when:

- The data items naturally form a set of the same data type.
- The entire set of data items in a single database column must be represented and controlled as a unit, as opposed to storing each item in a separate column.
- Each item must also be identified and accessed individually.

The data items in an array are called *array elements*. An array can contain elements of any InterBase data type except BLOB, and cannot be an array of arrays. All of the elements of a particular array are of the same data type.

Creating Arrays

Arrays are defined with the CREATE DOMAIN or CREATE TABLE statements. Defining an array column is just like defining any other column, except that the array dimensions must also be specified. For example, the following statement defines both a regular character column, and a single-dimension, character array column containing four elements:

```
EXEC SQL
   CREATE TABLE TABLE1
   (
      NAME CHAR(10),
      CHAR_ARR CHAR(10)[4]
   );
```

Array dimensions are always enclosed in square brackets following a column's data type specification.

For a complete discussion of CREATE TABLE and array syntax, see the *Language Reference*.

Multi-dimensional Arrays

InterBase supports *multi-dimensional arrays*, arrays with 1 to 16 dimensions. For example, the following statement defines three integer array columns with two, three, and six dimensions respectively:

```
EXEC SQL
   CREATE TABLE TABLE1
   (
      INT_ARR2 INTEGER[4,5]
      INT_ARR3 INTEGER[4,5,6]
      INT_ARR6 INTEGER[4,5,6,7,8,9]
   );
```

In this example, INT_ARR2 allocates storage for 4 rows, 5 elements in width, for a total of 20 integer elements, INT_ARR3 allocates 120 elements, and INT_ARR6 allocates 60,480 elements.

Important InterBase stores multi-dimensional arrays in row-major order. Some host languages, such as FORTRAN, expect arrays to be in column-major order. In these cases, care must be taken to translate element ordering correctly between InterBase and the host language.

Specifying Subscript Ranges for Array Dimensions

In InterBase, array dimensions have a specific range of upper and lower boundaries, called *subscripts*. In many cases, the subscript range is implicit: the first element of the array is element 1, the second element 2, and the last is element *n*. For example, the following statement creates a table with a column that is an array of four integers:

```
EXEC SQL
   CREATE TABLE TABLE1
   (
      INT_ARR INTEGER[4]
   };
```

The subscripts for this array are 1, 2, 3, and 4.

A different set of upper and lower boundaries for each array dimension can be explicitly defined when an array column is created. For example, C programmers, familiar with arrays that start with a lower subscript boundary of zero, may want to create array columns with a lower boundary of zero as well.

To specify array subscripts for an array dimension, both the lower and upper boundaries of the dimension must be specified using the following syntax:

```
lower:upper
```

For example, the following statement creates a table with a single-dimension array column of four elements where the lower boundary is 0 and the upper boundary is 3:

```
EXEC SQL
   CREATE TABLE TABLE1
   (
      INT_ARR INTEGER[0:3]
   };
```

The subscripts for this array are 0, 1, 2, and 3.

When creating multi-dimensional arrays with explicit array boundaries, separate each dimension's set of subscripts from the next with commas. For example, the following statement creates a table with a two-dimensional array column where each dimension has four elements with boundaries of 0 and 3:

```
EXEC SQL
   CREATE TABLE TABLE1
   (
      INT_ARR INTEGER[0:3, 0:3]
   };
```

Accessing Arrays

InterBase can perform operations on an entire array, effectively treating it as a single element, or it can operate on an *array slice*, a subset of array elements. An array slice can consist of a single element, or a set of many contiguous elements.

InterBase supports the following data manipulation operations on arrays:

- Selecting data from an array
- Inserting data into an array
- Updating data in an array slice
- Selecting data from an array slice
- Evaluating an array element in a search condition

A user-defined function (UDF) can only reference a single array element.

The following array operations are *not* supported:

- Referencing array dimensions dynamically in DSQL
- Inserting data into an array slice
- Setting individual array elements to NULL
- Using the aggregate functions, MIN(), MAX(), SUM(), AVG(), and COUNT() with arrays
- Referencing arrays in the GROUP BY clause of a SELECT
- Creating views that select from array slices

Selecting Data From an Array

To select data from an array, perform the following steps:

1. Declare a host-language array variable of the correct size to hold the array data. For example, the following statements create three such variables:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED ON TABLE1.CHAR_ARR  char_arr;
      BASED ON TABLE1.INT_ARR   int_arr;
      BASED ON TABLE1.FLOAT_ARR float_arr;
EXEC SQL
   END DECLARE SECTION;
```

2. Declare a cursor that specifies the array columns to select. For example,

```
EXEC SQL
   DECLARE TC1 CURSOR FOR
      SELECT NAME, CHAR_ARR[], INT_ARR[]
         FROM TABLE1;
```

   Be sure to include brackets ([]) after the array column name to select the array data. If the brackets are left out, InterBase reads the array ID for the column, instead of the array data.

   The ability to read the array ID, which is actually a BLOB ID, is included only to support applications that access array data using InterBase API calls.

3. Open the cursor, and fetch data:

```
EXEC SQL
   OPEN TC1;
EXEC SQL
   FETCH TC1 INTO :name, :char_arr, :int_arr;
```

Note It is not necessary to use a cursor to select array data. For example, a singleton SELECT might be appropriate, too.

When selecting array data, keep in mind that InterBase stores elements in row-major order. For example, in a 2-dimensional array, with 2 rows and 3 columns, all 3 elements in row 1 are returned, then all three elements in row two.

Inserting Data Into an Array

INSERT can be used to insert data into an array column. The data to insert must exactly fill the entire array, or an error can occur.

To insert data into an array, follow these steps:

1. Declare a host-language variable to hold the array data. Use the BASED ON clause as a handy way of declaring array variables capable of holding data to insert into the entire array. For example, the following statements create three such variables:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED ON TABLE1.CHAR_ARR  char_arr;
      BASED ON TABLE1.INT_ARR   int_arr;
      BASED ON TABLE1.FLOAT_ARR float_arr;
EXEC SQL
   END DECLARE SECTION;
```

2. Load the host-language variables with data.
3. Use INSERT to write the arrays. For example,

```
EXEC SQL
   INSERT INTO TABLE1 (NAME, CHAR_ARR, INT_ARR, FLOAT_ARR)
   VALUES ("Sample", :char_arr, :int_arr, :float_arr);
```

4. Commit the changes:

```
EXEC SQL
   COMMIT;
```

Important When inserting data into an array column, provide data to fill all array elements, or the results will be unpredictable.

Selecting From an Array Slice

The SELECT statement supports syntax for retrieving contiguous ranges of elements from arrays. These ranges are referred to as *array slices.* Array slices to retrieve are specified in square brackets ([]) following a column name containing

an array. The number inside the brackets indicates the elements to retrieve. For a one-dimensional array, this is a single number. For example, the following statement selects the second element in a one-dimensional array:

```
EXEC SQL
   SELECT JOB_TITLE[2]
      INTO :title
      FROM EMPLOYEE
      WHERE LAST_NAME = :lname;
```

To retrieve a subset of several contiguous elements from a one-dimensional array, specify both the first and last elements of the range to retrieve, separating the values with a colon. The syntax is as follows:

```
[lower_bound:upper_bound]
```

For example, the following statement retrieves a subset of three elements from a one-dimensional array:

```
EXEC SQL
   SELECT JOB_TITLE[2:4]
      INTO :title
      FROM EMPLOYEE
      WHERE LAST_NAME = :lname;
```

For multi-dimensional arrays, the lower and upper values for each dimension must be specified, separated from one another by commas, using the following syntax:

```
[lower:upper, lower:upper [, lower:upper ...]]
```

Note In this syntax, the bold brackets must be included.

For example, the following statement retrieves two rows of three elements each:

```
EXEC SQL
   DECLARE TC2 CURSOR FOR
      SELECT INT_ARR[1:2,1:3]
      FROM TABLE1
```

Because InterBase stores array data in row-major order, the first range of values between the brackets specifies the subset of rows to retrieve. The second range of values specifies which elements in each row to retrieve.

To select data from an array slice, perform the following steps:

1. Declare a host-language variable large enough to hold the array slice data retrieved. For example,

```
EXEC SQL
   BEGIN DECLARE SECTION;
      char char_slice[11]; /* 11-byte string for CHAR(10) data type */
```

```
      long int_slice[2][3];
EXEC SQL
   END DECLARE SECTION;
```

The first variable, *char_slice*, is intended to store a single element from the CHAR_ARR column. The second example, *int_slice*, is intended to store a six-element slice from the INT_ARR integer column.

2. Declare a cursor that specifies the array slices to read. For example,

```
EXEC SQL
   DECLARE TC2 CURSOR FOR
      SELECT CHAR_ARR[1], INT_ARR[1:2,1:3]
      FROM TABLE1
```

3. Open the cursor, and the fetch data:

```
EXEC SQL
   OPEN TC2;
EXEC SQL
   FETCH TC2 INTO :char_slice, :int_slice;
```

Updating Data in an Array Slice

A subset of elements in an array can be updated with a cursor. To perform an update, follow these steps:

1. Declare a host-language variable to hold the array slice data. For example,

```
EXEC SQL
   BEGIN DECLARE SECTION;
      char char_slice[11]; /* 11-byte string for CHAR(10) data type */
      long int_slice[2][3];
EXEC SQL
   END DECLARE SECTION;
```

The first variable, *char_slice*, is intended to hold a single element of the CHAR_ARR array column defined in the programming example in the previous section. The second example, *int_slice*, is intended to hold a six-element slice of the INT_ARR integer array column.

2. Select the row that contains the array data to modify. For example, the following cursor declaration selects data from the INT_ARRAY and CHAR_ARRAY columns:

```
EXEC SQL
   DECLARE TC1 CURSOR FOR
      SELECT CHAR_ARRAY[1], INT_ARRAY[1:2,1:3] FROM TABLE1;
EXEC SQL
```

```
   OPEN TC1;
EXEC SQL
   FETCH TC1 INTO :char_slice, :int_slice;
```

This example fetches the data currently stored in the specified slices of CHAR_ARRAY and INT_ARRAY, and stores it into the *char_slice* and *int_slice* host-language variables, respectively.

3. Load the host-language variables with new or updated data.
4. Execute an UPDATE statement to insert data into the array slices. For example, the following statements put data into parts of CHAR_ARRAY and INT_ARRAY, assuming *char_slice* and *int_slice* contain information to insert into the table:

```
EXEC SQL
   UPDATE TABLE1
   SET
      CHAR_ARR[1]  = :char_slice,
      INT_ARR[1:2,1:3] = :int_slice
   WHERE CURRENT OF TC1;
```

5. Commit the changes:

```
EXEC SQL
   COMMIT;
```

The following fragment of the output from this example illustrates the contents of the columns, CHAR_ARR and INT_ARR after this operation.

```
char_arr values:
 [0]:string0 [1]:NewString [2]:string2  [3]:string3

int_arr values:                                          updated values
 [0][0]:0  [0][1]:1   [0][2]:2   [0][3]:3
 [1][0]:10 [1][1]:999 [1][2]:999 [1][3]:999
 [2][0]:20 [2][1]:999 [2][2]:999 [2][3]:999
 [3][0]:30 [3][1]:31  [3][2]:32  [3][3]:33
```

Testing an Array Element Value in a Search Condition

A single array element's value can be evaluated in the search condition of a WHERE clause. For example,

```
EXEC SQL
   DECLARE TC2 CURSOR FOR
      SELECT CHAR_ARR[1], INT_ARR[1:2,1:3]
      FROM TABLE1
      WHERE SMALLINT_ARR[1,1,1] = 111;
```

Important Multi-element array slices cannot be evaluated.

Using Host Variables in Array Subscripts

Integer host variables can be used as array subscripts. For example, the following cursor declaration uses host variables, *getval*, and *testval*, in array subscripts:

```
EXEC SQL
   DECLARE TC2 CURSOR FOR
      SELECT CHAR_ARR[1], INT_ARR[:getval:1,1:3]
      FROM TABLE1
      WHERE FLOAT_ARR[:testval,1,1] = 111.0;
```

Using Arithmetic Expressions With Arrays

Arithmetic expressions involving arrays can be used only in search conditions. For example, the following code fetches a row of array data at a time that meets the search criterion:

```
for (i = 1; i < 100 && SQLCODE == 0; i++)
{
   EXEC SQL
      SELECT ARR[:i] INTO :array_var
      FROM TABLE1
      WHERE ARR1[:j + 1] = 5;
   process_array(array_var);
}
```

CHAPTER 10

Working With Security

This chapter describes SQL security, and how to assign and revoke security privileges for tables, views, and stored procedures, and also how to assign and revoke privileges for individual columns in a table. It discusses using views to restrict data access, and examines other InterBase security measures available to safeguard databases.

Overview of SQL Access Privileges

SQL security is controlled at the table level with *access privileges*, a list of operations that a user is allowed to perform on a given table or view. The GRANT statement assigns access privileges for a table or view to specified users or procedures. The REVOKE statement removes previously granted access privileges.

GRANT can also enable users or stored procedures to use stored procedures through the EXECUTE privilege, while REVOKE is used to remove those privileges.

Default Table Security and Access

In SQL, all tables are automatically secured against unauthorized access when they are created. Initially, only a table's creator, its *owner*, has access to a table, and only they may use GRANT to assign privileges to other users or to procedures.

Note On some platforms InterBase also supports a SYSDBA user who has access to all database objects; furthermore, on platforms that support the concept of a superuser, or user with root or locksmith privileges, such a user also has access to all database objects.

Default Procedure Security and Access

All stored procedures are automatically secured against unauthorized use when they are created. Initially, only a procedure's creator, its owner, can execute or call the procedure, and only its owner may assign EXECUTE privileges to other users or to other procedures.

Note On some platforms InterBase also supports a SYSDBA user who has access to all database objects; furthermore, on platforms that support the concept of a superuser, or user with root or locksmith privileges, such a user also has access to all database objects.

Privileges Available

The following table lists the SQL access privileges that can be granted and revoked:

Table 10-1: SQL Access Privileges

Privilege	Access
ALL	Select, delete, insert, and update data.
SELECT	Read data.
DELETE	Delete data.
INSERT	Write new data.
UPDATE	Modify existing data.
EXECUTE	Execute or call a stored procedure.

ALL grants or revokes SELECT, DELETE, INSERT, and UPDATE privileges using a single keyword, but does not grant EXECUTE privilege. SELECT, DELETE, INSERT, and UPDATE privileges can also be granted or revoked singly or in combination.

Note Statements that grant or revoke EXECUTE privilege cannot grant or revoke other privileges.

Granting Access to a Table

To give a user access to a table, use GRANT to assign privileges. At a minimum, GRANT requires parameters which specify an access privilege to grant, the table name to which access is to be granted, and the name of a user to whom the privilege will apply. User names are derived either from the system, or, for remote or

secured database attachments, from the *isc4.gdb* security database on the server where the database being accessed resides. For more information about *isc4.gdb*, see the *Windows Client User's Guide*.

For example, the following statement grants SELECT privilege for the DEPARTMENTS table to a user, EMIL:

```
EXEC SQL
   GRANT SELECT ON DEPARTMENTS TO EMIL;
```

A stored procedure, too, can be given privileges for a table. When a procedure is the recipient of privileges, the PROCEDURE keyword must precede the name of the procedure. For example, the following statement grants INSERT privilege for the ACCOUNTS table to the procedure, MONEY_TRANSFER:

```
EXEC SQL
   GRANT INSERT ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER
```

Tip As a security measure, privileges to tables can be granted to a procedure instead of to individual users. If a user has EXECUTE privilege on a procedure that accesses a table, then the user does not need privileges to the table.

Granting Multiple Privileges

To give a user a specific subset of privileges to a table, but not all of them, list the privileges, separating them with commas, in the GRANT statement. For example, the following statement assigns INSERT and UPDATE privileges for the DEPARTMENTS table to a user, LI:

```
EXEC SQL
   GRANT INSERT, UPDATE ON DEPARTMENTS TO LI;
```

Procedures, too, can be assigned a specific subset of privileges. When a procedure is given privileges, the PROCEDURE keyword must precede its name. For example, the following statement assigns INSERT and UPDATE privileges for the ACCOUNTS table to a procedure, MONEY_TRANSFER:

```
EXEC SQL
   GRANT INSERT, UPDATE ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER;
```

Any combination of SELECT, DELETE, INSERT, and UPDATE privileges can be assigned with the GRANT statement.

Granting All Privileges

The ALL privilege combines the SELECT, DELETE, INSERT, and UPDATE privileges for a table in a single expression. It is a shorthand way to assign all SQL access privileges to a user or procedure except for EXECUTE. For example, the following statement grants all access privileges for the DEPARTMENTS table to a user, SUSAN:

```
EXEC SQL
   GRANT ALL ON DEPARTMENTS TO SUSAN;
```

SUSAN can now SELECT, DELETE, INSERT, and UPDATE the DEPARTMENTS table.

Procedures, too, can be assigned all privileges. When a procedure is assigned privileges, the PROCEDURE keyword must precede its name. For example, the following statement grants all privileges for the ACCOUNTS table to the procedure, MONEY_TRANSFER:

```
EXEC SQL
   GRANT ALL ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER;
```

Granting Privileges to a List of Users

GRANT can be used to assign the same access privileges to a number of users at the same time. Instead of indicating a single user to whom privileges should be assigned, provide a list of users, each separated from one another by commas. The following statement gives INSERT and UPDATE privileges for the DEPARTMENTS table to users FRANÇOIS, BEATRICE, and HELGA:

```
EXEC SQL
   GRANT INSERT, UPDATE ON DEPARTMENTS TO FRANÇOIS, BEATRICE, HELGA;
```

Granting Privileges to a List of Procedures

GRANT can be used to assign the same access privileges to a number of stored procedures at the same time. Instead of indicating a single procedure to which privileges should be assigned, provide a list of procedures, each separated from one another by commas. The PROCEDURE keyword must precede the first procedure name in the list. The following statement gives INSERT and UPDATE privileges for the ACCOUNTS table to the procedures, ACCT_MAINT, and MONEY_TRANSFER:

```
EXEC SQL
   GRANT INSERT, UPDATE ON DEPARTMENTS TO PROCEDURE ACCT_MAINT,
      MONEY_TRANSFER;
```

Granting Privileges to All Users

To assign the same access privileges for a table to all users, use the PUBLIC keyword rather than listing all users in the GRANT statement. The following statement grants SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table to all users:

```
EXEC SQL
   GRANT SELECT, INSERT, UPDATE ON DEPARTMENTS TO PUBLIC;
```

Important PUBLIC only grants privileges to users, not to stored procedures. Privileges granted to users with PUBLIC can only be revoked from PUBLIC.

Granting Users UPDATE Access to Columns in a Table

Besides assigning access rights for an entire table at a time, GRANT can assign UPDATE rights for columns within a table. To assign access to columns within a table, a list of columns in parentheses, separated from one another with commas, should follow the list of rights to assign in the GRANT statement. For example, the following statement assigns UPDATE access for the CONTACT and PHONE columns in the CUSTOMERS table to all users:

```
EXEC SQL
   GRANT UPDATE (CONTACT, PHONE) ON CUSTOMERS TO PUBLIC;
```

Privileges granted to users for columns within a table may *increase* the rights already assigned to them at the table level, but cannot subtract from them.

To restrict user access to a table, use the REVOKE statement.

Granting Users the Right to Grant Privileges

GRANT can also be used to assign *grant authority*, the right to assign access privileges for a table, to other users, but not to stored procedures.

Initially, only a table's owner has grant authority for that table, so only the owner can grant SELECT, DELETE, INSERT, and UPDATE access to other users. When other users are granted access to the table, they cannot, in turn, grant access to still other users unless they have been given permission to do so.

To assign grant authority to another user, include the WITH GRANT OPTION clause at the end of a GRANT statement. For example, the following statement

assigns SELECT access to user, EMIL, and allows EMIL to grant SELECT access to other users:

```
EXEC SQL
   GRANT SELECT ON DEPARTMENTS TO EMIL WITH GRANT OPTION;
```

WITH GRANT OPTION clauses are cumulative, even if issued by different users. For example, EMIL can be given grant authority for SELECT by one user, and grant authority for INSERT by another user. For more information about cumulative privileges, see "Grant Authority Implications," in this chapter.

Grant Authority Restrictions

Users with grant authority can only assign the privileges assigned to them with the WITH GRANT OPTION clause.

For example, in the GRANT example in the previous section, EMIL is granted SELECT access to the DEPARTMENTS table. EMIL can grant SELECT privilege to other users. Suppose EMIL is now given INSERT access as well, but *without* the WITH GRANT OPTION:

```
EXEC SQL
   GRANT INSERT ON DEPARTMENTS TO EMIL;
```

EMIL can SELECT from and INSERT to the DEPARTMENTS table. EMIL can still grant SELECT privileges to other users, but *cannot* assign INSERT privileges.

To change a user's existing privileges to include grant authority, issue a second GRANT statement which includes the WITH GRANT OPTION clause. For example, to grant EMIL INSERT privilege with grant authority, reissue the GRANT statement and include the WITH GRANT OPTION clause:

```
EXEC SQL
   GRANT INSERT ON DEPARTMENTS TO EMIL WITH GRANT OPTION;
```

Grant Authority Implications

Consider every extension of grant authority with care. Once other users are permitted grant authority on a table, they can, in turn, grant those same privileges, and grant authority for them, to other users.

As the number of users with privileges and grant authority for a table increases, the likelihood that different users can grant the same privileges and grant authority to any single user also increases.

SQL permits duplicate privilege and authority assignment under the assumption that it is intentional. Duplicate privilege and authority assignments to a single user have implications for subsequent revocation of that user's privileges and authority. For more information about revoking privileges, see "Revoking User Access," in this chapter.

Suppose two users to whom the appropriate privileges and grant authority have been extended, GALENA and SUDHANSHU, both issue the following statement:

```
EXEC SQL
   GRANT INSERT ON DEPARTMENTS TO SPINOZA WITH GRANT OPTION;
```

Later, GALENA revokes the privilege and grant authority for SPINOZA:

```
EXEC SQL
   REVOKE INSERT ON DEPARTMENTS FROM SPINOZA;
```

GALENA now believes that SPINOZA no longer has INSERT privilege and grant authority for the DEPARTMENTS table. The immediate net effect of the statement is negligible because SPINOZA retains the INSERT privilege and grant authority assigned by SUDHANSHU.

When full control of access privileges on a table is desired, grant authority should not be assigned indiscriminately. In cases where privileges must be universally revoked for a user who may have received rights from several users, there are two options:

- Each user who assigned rights must issue an appropriate REVOKE statement.
- The table's owner must issue a REVOKE statement for all users of the table, then issue GRANT statements to reestablish access privileges for the users who should not lose their rights.

For more information about the REVOKE statement, see "Revoking User Access," in this chapter.

Granting Privileges to Execute Procedures

To use a stored procedure, users or other stored procedures must have EXECUTE privilege for it, using the following GRANT syntax:

```
EXEC SQL
   GRANT EXECUTE ON PROCEDURE name TO <userlist>;
```

```
<userlist> = username [, username ...] | PROCEDURE name
   [, name ...] | PUBLIC
```

<userlist> can contain both a list of user names and a list of procedure names. If PUBLIC is used, separate lists of users and procedures are prohibited. PUBLIC cannot be used to assign privileges to procedures.

In the following statement, EXECUTE privilege for the FUND_BALANCE procedure is extended to two users, NKOMO, and SUSAN, and to two procedures, ACCT_MAINT, and MONEY_TRANSFER:

```
EXEC SQL
   GRANT EXECUTE ON PROCEDURE FUND_BALANCE TO NKOMO, SUSAN, PROCEDURE
      ACCT_MAINT, MONEY_TRANSFER;
```

How GRANT Affects Views

In most respects, SQL treats a view as it does any other table. GRANT can be used to assign access privileges for it.

For SELECT privileges, this poses no problem. Reading data from a view does not change the view.

INSERT, UPDATE, and DELETE privileges for an updatable view, on the other hand, should be assigned with caution. To a user assigned these privileges, it appears that changes the user makes are applied to an actual table. In reality, for a user's changes to occur they must actually be made to the base table underlying the view.

Tip Updatable views for which INSERT and UPDATE privileges are to be granted should always be created using the SQL integrity constraint, WITH CHECK OPTION so that users can only update rows in the base table that can be accessed through the view.

Before granting INSERT, UPDATE, and DELETE privileges for a view, determine that it is updatable. For more information about updatable views, see the *Data Definition Guide*.

Views That are Subsets of a Table

When a view is based on a single table, data changes are made directly to the view's underlying table.

For UPDATE, changes only affect the columns visible through the view. Existing values in other invisible and inaccessible columns are not changed. Views

created using the SQL integrity constraint, WITH CHECK OPTION, can be updated only if the UPDATE statement fulfills the constraint's requirements.

For DELETE, the removal of a row affects columns invisible through the view. If SQL integrity constraints or triggers exist for any column in the underlying table, and the deletion of the row violates any of those constraints or trigger conditions, the DELETE statement fails.

For INSERT, the addition of a row affects columns both visible and accessible to the view, and columns which are invisible and inaccessible. Insertion succeeds only when:

- Data for insertion into visible columns meets SQL integrity constraint criteria for the column, and also meets any trigger conditions that apply.
- All other columns may contain NULL values. SQL integrity constraints or triggers for these columns might not allow NULL values in some or all of these columns.

For more information about creating and working with views, see the *Data Definition Guide.*

Views With Joins

When a view definition contains a join of any kind, it is no longer a legally updatable view, and InterBase cannot directly update the underlying tables.

Note If extreme care is taken, INSERT, UPDATE, and DELETE operations on views with joins are still possible if appropriate triggers are defined on the view. Even if triggers are defined, integrity constraints in the base tables might still prevent INSERT, UPDATE**,** and DELETE statements from succeeding.

For more information about integrity constraints and triggers, see the *Data Definition Guide.*

Revoking User Access

To take away a user's or stored procedure's privileges for a table or view, use the REVOKE statement. At a minimum, REVOKE requires parameters which specify one access privilege to remove, the table or view to which the privilege revocation applies, and the name of a user for whom or procedure for which the privilege should be revoked. For example, the following statement removes SELECT privilege for user SUSAN on the DEPARTMENTS table:

```
EXEC SQL
   REVOKE SELECT ON DEPARTMENTS FROM SUSAN;
```

The following statement removes UPDATE privilege for the procedure, MONEY_TRANSFER, on the ACCOUNTS table:

```
EXEC SQL
   REVOKE UPDATE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSER;
```

REVOKE also removes a user's or procedure's privilege to execute a stored procedure. For example, the following statement removes EXECUTE privilege for user, EMIL, on the MONEY_TRANSFER procedure:

```
EXEC SQL
   REVOKE EXECUTE ON PROCEDURE MONEY_TRANSFER FROM EMIL;
```

The next statement removes EXECUTE privilege for the procedure, ACCT_MAINT, on the MONEY_TRANSFER procedure:

```
EXEC SQL
   REVOKE EXECUTE ON PROCEDURE MONEY_TRANSER FROM PROCEDURE ACCT_MAINT;
```

For the complete syntax of REVOKE, see the *Language Reference*.

REVOKE Restrictions

The following restrictions and rules of scope apply to the REVOKE statement.

- Privileges can only be revoked by the user who granted them.
- Other privileges assigned by other users are not affected.
- Revoking a privilege for a user, A, to whom grant authority was given, automatically revokes that privilege for all users to whom it was subsequently assigned by user A.
- Privileges granted to PUBLIC can only be revoked for PUBLIC.

Revoking Multiple Privileges

To remove some, but not all, of the access privileges assigned to a user or procedure for a table, list the privileges to remove, separating them with commas. The following statement removes INSERT and UPDATE privileges for the DEPARTMENTS table from a user, LI:

```
EXEC SQL
   REVOKE INSERT, UPDATE ON DEPARTMENTS FROM LI;
```

The next statement removes INSERT and DELETE privileges for the ACCOUNTS table from a stored procedure, MONEY_TRANSFER:

```
EXEC SQL
   REVOKE INSERT, DELETE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSFER;
```

Any combination of previously assigned SELECT, DELETE, INSERT, and UPDATE privileges can be revoked.

Revoking All Privileges

The ALL privilege combines the SELECT, DELETE, INSERT, and UPDATE privileges for a table in a single expression. It is a shorthand way to remove all SQL table access privileges from a user or procedure. For example, the following statement revokes all access privileges for the DEPARTMENTS table for a user, SUSAN:

```
EXEC SQL
   REVOKE ALL ON DEPARTMENTS FROM SUSAN;
```

Even if a user does not have all access privileges for a table, ALL can still be used. Using ALL in this manner may be helpful when a current user's access rights are unknown.

Important ALL does not revoke EXECUTE privilege.

Revoking Privileges for a List of Users

REVOKE can be used to remove the same access privileges for a number of users at the same time. Instead of indicating a single user for whom privileges should be revoked, provide a list of users, each separated from one another by commas. For example, the following statement revokes INSERT and UPDATE privileges for the DEPARTMENTS table for users, FRANÇOIS, BEATRICE, and HELGA:

```
EXEC SQL
   REVOKE INSERT, UPDATE ON DEPARTMENTS FROM FRANÇOIS, BEATRICE, HELGA;
```

Revoking Privileges for a List of Procedures

REVOKE can also be used to remove the same access privileges for a number of procedures at the same time. Instead of indicating a single procedure for which privileges should be revoked, provide a list of procedures, each separated from one another by commas. The PROCEDURE keyword must precede the first procedure listed. For example, the following statement revokes INSERT and

UPDATE privileges for the ACCOUNTS table for the procedures, MONEY_TRANSFER, and ACCT_MAINT:

```
EXEC SQL
   REVOKE INSERT, UPDATE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSFER,
      ACCT_MAINT;
```

Revoking Privileges for All Users

To revoke privileges granted to all users as PUBLIC, use REVOKE with PUBLIC. The following statement revokes SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table for all users:

```
EXEC SQL
   REVOKE SELECT, INSERT, UPDATE ON DEPARTMENTS FROM PUBLIC;
```

Once this statement is executed, only the table's owner retains full access privileges to DEPARTMENTS.

Important PUBLIC does not revoke privileges for stored procedures. PUBLIC cannot be used to strip privileges from users who were granted them as individual users.

Revoking Grant Authority

To revoke a user's grant authority for a given privilege, use the following REVOKE syntax:

```
EXEC SQL
   REVOKE GRANT OPTION FOR privilege [, privilege ...] ON table
      FROM user;
```

The following statement revokes SELECT grant authority on the DEPARTMENTS table from a user, EMIL:

```
EXEC SQL
   REVOKE GRANT OPTION FOR SELECT ON DEPARTMENTS FROM EMIL;
```

Using Views to Restrict Data Access

Besides using GRANT and REVOKE to control access to database tables, views can be used to restrict data access. A view is usually created as a subset of columns and rows from one or more underlying tables. Because it is only a subset of its underlying tables, a view already provides a measure of access security.

For example, suppose an EMPLOYEE table contains columns for LAST_NAME, FIRST_NAME, JOB, SALARY, DEPT, and PHONE. This table contains much information that is useful to all employees. It also contains employee information that should remain confidential to almost everyone: SALARY. Rather than allow all employees access to the EMPLOYEE table, and the confidential salary information it contains, a view can be created which allows access to other columns in the EMPLOYEE table, but which excludes SALARY:

```
EXEC SQL
   CREATE VIEW EMPDATA AS
      SELECT LAST_NAME, FIRST_NAME, DEPARTMENT, JOB, PHONE
         FROM EMPLOYEE;
```

Access to the EMPLOYEE table can now be restricted, while SELECT access to the view, EMPDATA, can be granted to everyone.

Note Be careful when creating a view from base tables which contain sensitive information. Depending on the data included in a view, it may be possible for users to recreate or infer the missing data.

Providing Additional Security

Standard SQL security is controlled at the table level. The database that contains SQL tables is *not* protected in SQL. Because a database is a file, standard SQL leaves file security to the operating system. Properly managed by a system administrator, file security at the operating system level might be adequate for many situations, but not for Windows clients.

Other situations may call for additional database file security. InterBase supports a security database scheme that is optional for some server/client connections, and required for others. For example, all Windows client applications that connect to a server must pass a valid USER and PASSWORD combination to the server as part of the CONNECT statement. Any client application connecting to an NT or NetWare server must also supply a valid USER and PASSWORD combination. A valid combination is one that matches an entry in the security database on the server.

For more information about providing a USER and PASSWORD combination when connecting to a database, see the *Language Reference*. For information about creating and administering a security database on a server, see the *Windows Client User's Guide*.

CHAPTER 11

Working With User-defined Functions

Just as InterBase has built-in SQL functions such as MIN(), MAX(), and CAST(), it also supports libraries of external functions, or user-defined functions (UDFs). A *UDF* is a function written entirely in a host language to perform a data manipulation task not directly supported by InterBase. Possibilities include statistical, string, and date functions.

Important UDFs are not supported on NetWare.

Once a UDF is created, it can be used in a database application anywhere that a built-in SQL function can be used. This chapter describes how to create UDFs and how to use them in an application.

Creating a UDF

Creating a UDF is a three-step process:

1. Writing and compiling a UDF in a programming language such as C.
2. Building a dynamically linked library containing the UDF.
3. Declaring the UDF to the database.

All steps in this process are programming tasks, except for declaring a UDF to the database, which is a data-definition task. Step 2, building a dynamic link library of UDFs is optional, though highly recommended. Building a library enables the database to link the library to an application at run time without requiring user or programmer intervention.

If a library is not built, then each individual UDF object file must be made separately available to the database.

Writing and Compiling Functions

A UDF can be written in C or in any other host language that can be called from C. Throughout this chapter, the sample UDF code comes from a single C source file, *udflib.c*, in the InterBase *examples* directory.

Writing a Function Module

In C, a UDF is written like any standard function. The UDF can require up to ten input parameters, and must return only a single C data value. A single source code module can define one or more UDFs. For example, the sample UDF module, *udflib.c*, contains the following UDFs:

- **fn_abs()** returns the absolute value of a number passed as an input argument.
- **fn_datediff()** takes two InterBase dates as input, and returns the number of days between them.
- **fn_trim()** imitates the SQL-92 TRIM() function. It takes three input arguments, an integer specifying the string trim operation to perform (trim leading characters, trim trailing characters, or trim both leading and trailing characters), the character to trim, and the string from which to trim characters. It returns the trimmed string.

The sample code for these functions is as follows:

```
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <time.h>

/* Defines for fn_trim(). */

#define LEADING 0
#define TRAILING 1
#define BOTH 2

/* Function prototypes. */

static char *strtriml(char *string, int c);
static char *strtrimr(char *string, int c);
char *fn_trim(int operation, int c, char *string);
long fn_datediff(ISC_QUAD d1, ISC_QUAD d2);
double fn_abs(double *x);
```

```
/* Function Definitons */

/* fn_abs() returns the absolute value of its argument. */
double fn_abs( double *x)
{
   return(*x < 0.0) ? -*x : *x;
}
/* fn_datediff() returns the number of days between two dates */
long fn_datediff(ISC_QUAD d1, ISC_QUAD d2)
{
   struct tm tm1, tm2;

   isc_decode_date(d1, &tm1); /* convert IB date to tm */
   isc_decode_date(d2, &tm2);
   return(long) (timelocal(&tm1) - timelocal(&tm2)) / (24 * 3600.0);
}
/* trim leading and/or trailing characters of type c from string */
char *fn_trim(int operation, int c, char *string)
{
   switch (operation) {
      case LEADING:
         strtriml(string, c);
      case TRAILING:
         strtrimr(string, c);
         break;
      case BOTH:
      default:
         strtrimr(string, c);
         strtriml(string, c);
         break;
   }
   return(string);
}

/* trim all chars of type c from left of string and close up */

static char *strtriml(char *string, int c)
{
   int n,i;

   n = 0;
   while (string[n] == c) /* skip leading characters */
      n++;
   for (i = 0;string[n];i++,n++) /* copy backward over itself */
      string[i] = string[n];
   string[i] = NULL; /* don't forget string terminator */
}
/* trim all chars of type c from right of string and truncate length */

static char *strtrimr(char *string, int c)
{
   int n;
```

```
    n = strlen(string) - 1;
    while (string[n] == c)
       n--;
    string[n + 1] = NULL;
    return(string);
}
```

As this sample code illustrates, a UDF source code module can use typedefs defined in the InterBase *ibase.h* header file. To compile such a module successfully, include *ibase.h* in the source code by adding the following include directive:

```
#include "ibase.h"
```

Note The sample code also includes calls to two InterBase library functions, **isc_decode_date()**, and **isc_encode_date()**. The *ibase.h* header file includes function prototypes for all InterBase library function calls.

Specifying Parameters

A UDF can accept up to ten parameters corresponding to any InterBase data type. Array elements cannot be passed as parameters. If a UDF returns a BLOB, the number of input parameters is restricted to nine. All parameters are passed to the UDF by reference.

Programming language data types specified as parameters must be capable of handling corresponding InterBase data types. For example, the C function declaration for **fn_abs()** accepts one parameter of type double. The expectation is that when **fn_abs()** is called, it will be passed a data type of DOUBLE PRECISION by InterBase.

UDFs that accept BLOB parameters require special data structure for processing. A BLOB is passed by reference to a BLOB UDF structure. For more information about the BLOB UDF structure, see “Writing a BLOB UDF,” in this chapter.

Specifying a Return Value

A UDF can return values that can be translated into any InterBase data type, including a BLOB, but it cannot return arrays of data types. For example, the C function declaration for **fn_abs()** returns a value of type double, which corresponds to the InterBase DOUBLE PRECISION data type.

By default, return values are passed by reference. Numeric values can be returned by reference or by value. To return a numeric parameter by value, include the optional BY VALUE keyword after the return value when declaring a UDF to a database.

A UDF that returns a BLOB does not actually define a return value. Instead, a pointer to a structure describing the BLOB to return must be passed as the last input parameter to the UDF.

For more information about declaring UDFs, see “Declaring a UDF to a Database,” in this chapter. For more information about declaring a BLOB UDF, see “Declaring a BLOB UDF,” in this chapter.

Writing a BLOB UDF

A BLOB UDF differs from other UDFs, because pointers to BLOB control structures are passed to the UDF instead of references to actual data. A BLOB UDF cannot open or close a BLOB, but instead invokes functions to perform BLOB access.

Creating a BLOB Control Structure

A BLOB control structure is a C struct, declared within a UDF module as a typedef. Programmers must provide the control structure definition, which should be defined as follows:

```
typedef struct blob {
   void (*blob_get_segment) ();
   int *blob_handle;
   long number_segments;
   long max_seglen;
   long total_size;
   void (*blob_put_segment) ();
} *BLOB;
```

blob_get_segment

The first field in the BLOB struct, *blob_get_segment*, is a pointer to a function that is called to read a segment from a BLOB if one is passed to the UDF. The function takes four arguments: a BLOB handle, the address of a buffer into which to place BLOB a segment of data that is read, the size of that buffer, and the address of variable into to store the size of the segment that is read.

If BLOB data is not read by the UDF, set *blob_get_segment* to NULL.

blob_handle

The second field in the BLOB struct, *blob_handle*, is required. It is a BLOB handle that uniquely identifies a BLOB passed to a UDF or returned by it.

number_segments

For BLOB data passed to a UDF, *number_segments* specifies the total number of segments in the BLOB. Set this value to NULL if BLOB data is not passed to a UDF.

max_seglen

For BLOB data passed to a UDF, *max_seglen* specifies the size, in bytes, of the largest single segment passed. Set this value to NULL if BLOB data is not passed to a UDF.

total_size

For BLOB data passed to a UDF, *total_size* specifies the actual size, in bytes, of the BLOB as a single unit. Set this value to NULL if BLOB data is not passed to a UDF.

blob_put_segment

The last field in the BLOB struct, *blob_put_segment*, is a pointer to a function that is called to write a segment to a BLOB if one is being returned by the UDF. The function takes three arguments: a BLOB handle, the address of a buffer containing the data to write into the BLOB, and the size, in bytes, of the data to write.

If BLOB data is not read by the UDF, set *blob_put_segment* to NULL.

A BLOB UDF Example

The following code creates a UDF, **blob_concatenate()**, that appends data from one BLOB to the end of another BLOB to return a third BLOB consisting of concatenated BLOB data.

```
/* BLOB control structure */
typedef struct blob {
   void (*blob_get_segment) ();
   int *blob_handle;
   long number_segments;
   long max_seglen;
   long total_size;
   void (*blob_put_segment) ();
} *BLOB;

extern char *isc_$alloc();
#define MAX(a, b) (a > b) ? a : b
#define DELIMITER "----------------------------"

blob_concatenate(BLOB from1, BLOB from2, BLOB to)
/* Note BLOB to, as final input parameter, is actually for output! */
```

```
{
   char *buffer;
   long length, b_length;

   b_length = MAX(from1->max_seglen, from2->max_seglen);
   buffer = isc_alloc(b_length);

   /* write the from1 BLOB into the return BLOB, to */
   while ((*from1->blob_get_segment) (from1->blob_handle, buffer,
      b_length, &length))
         (*to->blob_put_segment) (to->blob_handle, buffer, length);

   /* now write a delimiter as a dividing line in the blob */
   (*to->blob_put_segment) (to->blob_handle, DELIMITER,
      sizeof(DELIMITER) - 1);

   /* finally write the from2 BLOB into the return BLOB, to */
   while ((*from2->blob_get_segment) (from2->blob_handle, buffer,
      b_length, &length))
         (*to->blob_put_segment) (to->blob_handle, buffer, length);

   /* free the memory allocated to the buffer */
   isc_$free(buffer);
}
```

Compiling a Function Module

After a UDF module is written, it can be compiled in a normal fashion into object or library format. The UDFs in the resulting object or library module can then be declared to the database. Once declared to the database, the library containing all the UDFs is automatically loaded at run time from a shared library or dynamic link library.

Note UDFs are not supported on NetWare. Not all other platforms support dynamically linked or shared libraries. On these platforms, UDF libraries must be linked explicitly into applications, or the InterBase server. For more information about platform-specific linking requirements, see the *Installing and Running on . . . Guide* for that platform.

Creating a UDF Library

UDF libraries are standard C object libraries that are dynamically loaded by the database at run time. UDF libraries can be created on any platform supported by InterBase. To use the same set of UDFs with databases running on different platforms, create separate libraries on each platform where the databases reside. UDFs run on the server where the database resides.

The InterBase *examples* directory contains a sample makefile, *make.lib*, that builds a UDF function library from *udflib.c*. *make.lib* is specific to each platform where InterBase runs.

Modifying a UDF Library

To add a UDF to an existing UDF library on a platform:

- Compile the UDF according to the instructions for the platform.
- Include all object files previously included in the library and the newly-created object file in the command line when creating the function library.

Note On some platforms, object files can be added directly to existing libraries. For more information, consult the platform-specific compiler and linker documentation.

To delete a UDF from a library, follow the linker's instructions for removing an object from a library. Deleting a UDF from a library does not eliminate references to it in the database.

Declaring a UDF to a Database

After a UDF is created, it must also be declared to any databases where it will be used. Declaring a UDF to a database informs the database about the function:

- Its name as it will be used in embedded SQL statements.
- The number and data types of its arguments.
- The return data type.
- The name of the function as it exists in the UDF module or library.
- The name of the module where the UDF exists.

To declare a UDF to a database, follow these steps:

1. Start **isql** and connect to the desired database.
2. Use the DECLARE EXTERNAL FUNCTION statement to inform the database about the UDF.

The syntax for DECLARE EXTERNAL FUNCTION is as follows:

```
DECLARE EXTERNAL FUNCTION sql_name
   [<datatype> | CSTRING (int) [, <datatype>] | CSTRING (int) ...]
   RETURNS {<datatype> [BY VALUE] | CSTRING (int)}
   ENTRY_POINT "<entryname>"
   MODULE_NAME "<modulename>";
```

The following table describes the parameters for DECLARE EXTERNAL FUNCTION:

Table 11-1: DECLARE EXTERNAL FUNCTION Parameters

Argument	Description
sql_name	Name of the UDF as it will appear in SQL statements.
<datatype>	InterBase data type of an input parameter. All input parameters are passed to a UDF by reference. A UDF can have zero or more input parameters.
RETURNS	Specifies the return value of a function.
PARAMETER *pnum*	Specifies that the return value for the UDF is stored in the input parameter identified by *<pnum>*.
<datatype> [BY VALUE]	Specifies the InterBase SQL data type returned by the UDF. Return values are passed by reference unless the optional BY VALUE clause is used. Only numeric data types can be returned by value.
CSTRING (*int*)	Specifies a string value *int* bytes in length.
ENTRY_POINT "*<entryname>*"	Quoted string specifying the name of the UDF as stored in the linked library.
MODULE_NAME "*<modulename>*"	Quoted file specification identifying the library or module in which the UDF resides.

sql_name is the name of the UDF that is used to call the function from SQL statements. It can be different from the name of the actual function in the library or module source code. That name must be specified as *"<entryname>"*. For example, the following **isql** script declares three UDFs, ABS(), DATEDIFF(), and TRIM(), to the *employee.gdb* database:

```
CONNECT "employee.gdb";
DECLARE EXTERNAL FUNCTION ABS
   DOUBLE PRECISION
   RETURNS DOUBLE BY VALUE
   ENTRY_POINT "fn_abs" MODULE_NAME "udflib.lib";
COMMIT;
DECLARE EXTERNAL FUNCTION DATEDIFF
   DATE, DATE
   RETURNS INTEGER
   ENTRY_POINT "fn_datediff" MODULE_NAME "udflib.lib";
COMMIT;
DECLARE EXTERNAL FUNCTION TRIM
```

```
   SMALLINT, CSTRING(256), SMALLINT
   RETURNS CSTRING(256)
   ENTRY_POINT "fn_trim" MODULE_NAME "udflib.lib";
COMMIT;
```

Although UDFs are written in a host language and therefore take host-language data types for both its parameters and its return value, when a UDF is declared, it must translate them to SQL data types or to a CSTRING type of a specified maximum byte length. CSTRING is used to translate parameters of CHAR and VARCHAR data types into a null-terminated C string for processing, and to return a variable-length, null-terminated C string to InterBase for automatic conversion to CHAR or VARCHAR.

For a complete discussion of declaring UDFs, see the *Data Definition Guide*. For examples of calling UDFs from an application, including the use of CSTRING as a parameter and return value, see "Calling a UDF," in this chapter.

Declaring a BLOB UDF

A BLOB UDF is declared to the database using DECLARE EXTERNAL FUNCTION like any non-BLOB UDF. A UDF that returns a BLOB does not actually define a return value. Instead, a pointer to a structure describing the BLOB to return must be passed as the last input parameter to the UDF. For example, the following statement declares the BLOB UDF, BLOB_PLUS_BLOB, to a database:

```
DECLARE EXTERNAL FUNCTION BLOB_PLUS_BLOB
   BLOB,
   BLOB,
   BLOB
   ENTRY_POINT "blob_concatenate" MODULE_NAME "udflib.lib";
COMMIT;
```

Calling a UDF

After a UDF is created and declared to a database, it can be used in SQL statements wherever a built-in function is permitted. To use a UDF, insert its name in an SQL statement at an appropriate location, and enclose its input arguments in parentheses.

For example, the following DELETE statement calls the ABS() UDF as part of a search condition that restricts which rows are deleted:

```
EXEC SQL
   DELETE FROM CITIES
      WHERE ABS (POPULATION - 100000) > 50000;
```

UDFs can also be called in stored procedures and triggers. For more information, see the *Data Definition Guide*.

Using a UDF With SELECT

In SELECT statements, a UDF can be used in a select list to specify data retrieval, or in the WHERE clause search condition.

The following statement uses ABS() to guarantee that a returned column value is positive:

```
EXEC SQL
   SELECT ABS (JOB_GRADE) FROM PROJECTS;
```

The next statement uses DATEDIFF() in a search condition to restrict rows retrieved:

```
EXEC SQL
   SELECT START_DATE FROM PROJECTS
      WHERE DATEDIFF (:today, START_DATE) > 10;
```

Using a UDF With INSERT

In INSERT statements, a UDF can be used in the comma-delimited list in the VALUES clause.

The following statement uses TRIM() to remove leading blanks from *firstname* and trailing blanks from *lastname* before inserting the values of these host variables into the EMPLOYEE table:

```
EXEC SQL
   INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, EMP_NO, DEPT_NO, SALARY)
      VALUES (TRIM (0, ' ',:firstname), TRIM (1, ' ', :lastname),
         :empno, :deptno, greater(30000, :est_salary));
```

Using a UDF With UPDATE

In UPDATE statements, a UDF can be used in the SET clause as part of the expression assigning column values.

For example, the following statement uses TRIM() to ensure that update values do not contain leading or trailing blanks:

```
EXEC SQL
   UPDATE COUNTRIES
      SET COUNTRY = TRIM (2, ' ', COUNTRY);
```

Using a UDF With DELETE

In DELETE statements, a UDF can be used in a WHERE clause search condition:

```
EXEC SQL
   DELETE FROM COUNTRIES
      WHERE ABS (POPULATION - 100000) < 50000;
```

CHAPTER 12

Working With Stored Procedures

A *stored procedure* is a self-contained set of extended SQL statements stored in a database as part of its metadata. Stored procedures can pass parameters to and receive return values from applications. In applications, stored procedures can be invoked directly to perform a task, or can be substituted for a table or view in a SELECT statement.

The advantages of using stored procedures are:

- Applications can share code. A common piece of SQL code written once and stored in the database can be used in any application that accesses the database, including the new InterBase interactive SQL tool, **isql**.
- Modular design. Stored procedures can be shared among applications, eliminating duplicate code, and reducing the size of applications.
- Streamlined maintenance. When a procedure is updated, the changes are automatically reflected in all applications that use it without the need to recompile and relink them.
- Improved performance, especially for remote client access. Stored procedures are executed by the server, not the client.

This chapter describes how to call and execute stored procedures in applications once they are written. For information on how to create a stored procedure, see the *Data Definition Guide*.

Using Stored Procedures

There are two types of procedures that can be called from an application:

- *Select procedures* that an application can use in place of a table or view in a SELECT statement. A select procedure must return one or more values, or an error results.

- *Executable procedures* that an application can call directly, with the EXECUTE PROCEDURE statement. An executable procedure may or may not return values to the calling program.

Both kinds of procedures are defined with CREATE PROCEDURE and have the same syntax. The difference is in how the procedure is written and how it is intended to be used. Select procedures always return zero or more rows, so that to the calling program they appear as a table or view. Executable procedures are simply routines invoked by the calling program that can return only a single set of values.

In fact, a single procedure conceivably can be used as a select procedure or an executable procedure, but this is not recommended. In general a procedure is written specifically to be used in a SELECT statement (a select procedure) or to be used in an EXECUTE PROCEDURE statement (an executable procedure). For more information on creating stored procedures, see the *Data Definition Guide*.

Procedures and Transactions

Procedures operate within the context of a transaction in the program that uses them. If procedures are used in a transaction, and the transaction is rolled back, then any actions performed by the procedures are also rolled back. Similarly, a procedure's actions are not final until its controlling transaction is committed.

Security for Procedures

Users must be given EXECUTE privilege to use a stored procedure in an application. An extension to the GRANT statement enables assignment of EXECUTE privilege, and an extension to the REVOKE statement enables removal of the privilege. For more information about granting privileges to users, see the *Data Definition Guide*.

Stored procedures themselves sometimes need access to tables or views for which a user does not—or should not—have privileges. The GRANT statement assigns privileges to procedures, and REVOKE eliminates privileges.

Using Select Procedures

A select procedure is used in place of a table or view in a SELECT statement and can return zero or more rows. A select procedure must return one or more output parameters, or an error results. If returned values are not specified, NULL values are returned by default.

The advantages of select procedures over tables or views are:

- They can take input parameters that can affect the output produced.
- They can contain control statements, local variables, and data manipulation statements, offering great flexibility to the user.

Input parameters are passed to a select procedure in a comma-delimited list in parentheses following the procedure name. For example, the following **isql** script file defines the procedure, GET_EMP_PROJ, which returns the project numbers to which an employee is assigned (PROJ_ID) when passed the employee number (*emp_no*) as an input parameter:

```
SET TERM !! ;
CREATE PROCEDURE GET_EMP_PROJ (emp_no SMALLINT)
RETURNS (PROJ_ID CHAR(5)) AS
BEGIN
   FOR SELECT PROJ_ID
      FROM EMPLOYEE_PROJECT
      WHERE EMP_NO = :emp_no
      INTO :PROJ_ID
   DO
      SUSPEND;
END !!
```

The following statement retrieves PROJ_ID from the above procedure, passing the host variable, *number*, as input:

```
SELECT PROJ_ID FROM GET_EMP_PROJ (:number);
```

Calling a Select Procedure

To use a select procedure in place of a table or view name in an application, use the procedure name anywhere a table or view name is appropriate. Supply any input parameters required in a comma-delimited list in parentheses following the procedure name.

```
EXEC SQL
   SELECT PROJ_ID FROM GET_EMP_PROJ (:emp_no)
   ORDER BY PROJ_ID;
```

Using a Select Procedure With Cursors

A select procedure can also be used in a cursor declaration. For example, the following code declares a cursor named PROJECTS, using the GET_EMP_PROJ procedure in place of a table:

```
EXEC SQL
```

```
DECLARE PROJECTS CURSOR FOR
SELECT PROJ_ID FROM GET_EMP_PROJ (:emp_no)
   ORDER BY PROJ_ID;
```

The following application C code with embedded SQL then uses the PROJECTS cursor to print project numbers to standard output:

```
EXEC SQL
OPEN PROJECTS;

/* Print employee projects. */
while (SQLCODE == 0)
{
   EXEC SQL
      FETCH PROJECTS INTO :proj_id :nullind;

   if (SQLCODE == 100)
      break;
   if (nullind == 0)
      printf("\t%s\n", proj_id);
}
```

Using Executable Procedures

An executable procedure is called directly by an application, and often performs a task common to applications using the same database. Executable procedures can receive input parameters from the calling program, and can optionally return a single row to the calling program.

Input parameters pass to an executable procedure in a comma-delimited list following the procedure name.

Note Executable procedures cannot return multiple rows.

Executing a Procedure

To execute a procedure in an application, use the following syntax:

```
EXEC SQL
   EXECUTE PROCEDURE name [:param [[INDICATOR]:indicator]]
      [, :param [[INDICATOR]:indicator] ...]
      [RETURNING_VALUES :param [[INDICATOR]:indicator]
      [, :param [[INDICATOR]:indicator]...]];
```

When an executable procedure uses input parameters, the parameters can be literal values (such as 7 or “Fred”), or host variables. If a procedure returns output

parameters, host variables must be supplied in the RETURNING_VALUES clause to hold the values returned.

For example, the following statement demonstrates how the executable procedure, DEPT_BUDGET, is called with literal parameters:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET 100 RETURNING_VALUES :sumb;
```

The following statement also calls the same procedure using a host variable instead of a literal as the input parameter:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNING_VALUES :sumb;
```

Indicator Variables

Both input parameters and return values can have associated indicator variables for tracking NULL values. You must use indicator variables to indicate unknown or NULL values of return parameters. The INDICATOR keyword is optional. An indicator variable that is less than zero indicates that the parameter is unknown or NULL. An indicator variable that is 0 indicates that the associated parameter contains a non-NULL value. For more information about indicator variables, see Chapter 6: “Working With Data.”

Executing a Procedure in a DSQL Application

To execute a stored procedure in a dynamic SQL (DSQL) application follow these steps:

1. Use a PREPARE statement to parse and prepare the procedure call for execution using the following syntax:

   ```
   EXEC SQL
      PREPARE sql_statement_name FROM :var | "<statement>";
   ```

2. Set up an input XSQLDA using the following syntax:

   ```
   EXEC SQL
      DESCRIBE INPUT sql_statement_name INTO SQL DESCRIPTOR input_xsqlda;
   ```

3. Use DESCRIBE OUTPUT to set up an output XSQLDA using the following syntax:

   ```
   EXEC SQL
      DESCRIBE OUTPUT sql_statement_name INTO SQL DESCRIPTOR
         output_xsqlda;
   ```

Setting up an output XSQLDA is only necessary for procedures that return values.

4. Execute the statement using the following syntax:

```
EXEC SQL
   EXECUTE statement USING SQL DESCRIPTOR input_xsqlda
   INTO DESCRIPTOR output_xsqlda;
```

Input parameters to stored procedures can be passed as run-time values by substituting a question mark (?) for each value. For example, the following DSQL statements prepare and execute the ADD_EMP_PROJ procedure:

```
. . .
strcpy(uquery, "EXECUTE PROCEDURE ADD_EMP_PROJ ?, ?");
. . .
EXEC SQL
   PREPARE QUERY FROM :uquery;
EXEC SQL
   DESCRIBE INPUT QUERY INTO SQL DESCRIPTOR input_xsqlda;
EXEC SQL
   DESCRIBE OUTPUT QUERY INTO SQL DESCRIPTOR output_xsqlda;
EXEC SQL
   EXECUTE QUERY USING SQL DESCRIPTOR input_xsqlda INTO SQL DESCRIPTOR
      output_xsqlda;
. . .
```

CHAPTER 13

Working With Events

This chapter describes the InterBase event mechanism and how to write applications that register interest in and respond to events. The event mechanism enables applications to respond to actions and database changes made by other, concurrently running applications without the need for those applications to communicate directly with one another, and without incurring the expense of CPU time required for periodic polling to determine if an event has occurred.

Understanding the Event Mechanism

In InterBase, an *event* is a message passed by a trigger or a stored procedure to the InterBase event manager to announce the occurrence of a specified condition or action, usually a database change such as an INSERT, UPDATE, or DELETE. Events are passed by triggers or stored procedures only when the transaction under which they occur is committed.

The *event manager* maintains a list of events posted to it by triggers and stored procedures. It also maintains a list of applications that have registered an interest in events. Each time a new event is posted to it, the event manager notifies interested applications that the event has occurred.

Applications can respond to specific events that might be posted by a trigger or stored procedure by:

1. Indicating an interest in the events to the event manager.
2. Waiting for event notification.
3. Determining which event occurred (if an application is waiting for more than one event to occur).

The InterBase event mechanism, then, consists of three parts:

- A trigger or stored procedure that posts an event to the event manager.

- The event manager that maintains an event queue and notifies applications when an event occurs.
- An application that registers interest in the event and waits for it to occur.

A second application that uses the event-posting stored procedure (or that fires the trigger) causes the event manager to notify the waiting application so that it can resume processing.

Signaling Event Occurrence With POST_EVENT

A trigger or stored procedure must signal the occurrence of an event, usually a database change such as an INSERT, UPDATE, or DELETE, by using the POST_EVENT statement. POST_EVENT alerts the event manager to the occurrence of an event after a transaction is committed. At that time, the event manager passes the information to registered applications.

A trigger or stored procedure that posts an event is sometimes called an *event alerter*. For example, the following **isql** script creates a trigger that posts an event to the event manager whenever any application inserts data in a table:

```
SET TERM !! ;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
   ACTIVE
   AFTER INSERT
   POSITION 0
   AS
      BEGIN
         POST_EVENT "new_order";
      END
!!
SET TERM ; !!
```

Event names are restricted to 15 characters in size.

Note POST_EVENT is a stored procedure and trigger language extension, available only within stored procedures and triggers.

For a complete discussion of writing a trigger or stored procedure as an event alerter, see the *Data Definition Guide*.

Registering Interest in Events With EVENT INIT

An application must register a request to be notified about a particular event with the InterBase event manager before waiting for the event to occur. To register interest in an event, use the EVENT INIT statement. EVENT INIT requires two arguments:

- An application-specific request handle to pass to the event manager.
- A list of events to be notified about, enclosed in parentheses.

The application-specific request handle is used by the application in a subsequent EVENT WAIT statement to indicate a readiness to receive event notification. The request handle is used by the event manager to determine where to send notification about particular events to wake up a sleeping application so that it can respond to them.

The list of event names in parentheses must match event names posted by triggers or stored procedures, or notification cannot occur.

To register interest in a single event, use the following EVENT INIT syntax:

```
EXEC SQL
   EVENT INIT request_name (event_name);
```

event_name can be up to 15 characters in size, and can be passed as a constant string in quotes, or as a host-language variable.

For example, the following application code creates a request named RESPOND_NEW that registers interest in the “new_order” event:

```
EXEC SQL
   EVENT INIT RESPOND_NEW ("new_order");
```

The next example illustrates how RESPOND_NEW might be initialized using a host-language variable, *myevent*, to specify the name of an event:

```
EXEC SQL
   EVENT INIT RESPOND_NEW (:myevent);
```

After an application registers interest in an event, it is not notified about an event until it first pauses execution with EVENT WAIT. For more information about waiting for events, see “Waiting for Events With EVENT WAIT,” in this chapter.

Note As an alternative to registering interest in an event and waiting for the event to occur, applications can use an InterBase API call to register interest in an event, and identify an asynchronous trap (AST) function to receive event notification. This method enables an application to continue other

processing instead of waiting for an event to occur. For more information about programming events with the InterBase API, see the *API Guide*.

Registering Interest in Multiple Events

Often, an application may be interested in several different events even though it can only wait on a single request handle at a time. EVENT INIT enables an application to specify a list of event names in parentheses, using the following syntax:

```
EXEC SQL
   EVENT INIT request_name (event_name [event_name ...]);
```

Each *event_name* can be up to 15 characters in size, and should correspond to event names posted by triggers or stored procedures, or notification may never occur. For example, the following application code creates a request named RESPOND_MANY that registers interest in three events, “new_order,” “change_order,” and “cancel_order”:

```
EXEC SQL
   EVENT INIT RESPOND_MANY ("new_order", "change_order",
      "cancel_order");
```

Note An application can also register interest in multiple events by using a separate EVENT INIT statement with a unique request handle for a single event or groups of events, but it can only wait on one request handle at a time.

Waiting for Events With EVENT WAIT

Even after an application registers interest in an event, it does not receive notification about that event. Before it can receive notification, it must use the EVENT WAIT statement to indicate its readiness to the event manager, and to suspend its processing until notification occurs.

To signal the event manager and suspend an application’s processing, use the following EVENT WAIT statement syntax:

```
EXEC SQL
   EVENT WAIT request_name;
```

request_name must be the name of a request handle declared in a previous EVENT INIT statement.

The following statements register interest in an event, and wait for event notification:

```
EXEC SQL
   EVENT INIT RESPOND_NEW ("new_order");
EXEC SQL
   EVENT WAIT RESPOND_NEW;
```

Once EVENT WAIT is executed, application processing stops until the event manager sends a notification message to the application.

Note An application can contain more than one EVENT WAIT statement, but all processing stops when the first statement is encountered. When processing is restarted, it stops when the next EVENT WAIT is encountered, and so forth.

If one event occurs while an application is processing another, the event manager sends notification the next time the application returns to a wait state.

Responding to Events

When event notification occurs, a suspended application resumes normal processing at the next statement following EVENT WAIT.

If an application has registered interest in more than one event with a single EVENT INIT call, then the application must determine which event occurred by examining the event array, **isc_event[].** The event array is automatically created for an application during preprocessing. Each element in the array corresponds to an event name passed as an argument to EVENT INIT. The value of each element is the number of times that event occurred since execution of the last EVENT WAIT statement with the same request handle.

In the following code, an application registers interest in three events, then suspends operation pending event notification:

```
EXEC SQL
   EVENT INIT RESPOND_MANY ("new_order", "change_order",
      "cancel_order");
EXEC SQL
   EVENT WAIT RESPOND_MANY;
```

When any of the “new_order,” “change_order,” or “cancel_order” events are posted and their controlling transactions commit, the event manager notifies the application and processing resumes. The following code illustrates how an application might test which event occurred:

```
for (i = 0; i < 3; i++)
{
   if (isc_$event[i] > 0)
   {
      /* this event occurred, so process it */
      . . .
   }
}
```

CHAPTER 14

Error Handling and Recovery

All SQL applications should include mechanisms for trapping, responding to, and recovering from *run-time errors*, the errors that can occur when someone uses an application. This chapter describes both standard, portable SQL methods for handling errors, and additional error handling specific to InterBase.

Standard Error Handling

Every time an SQL statement is executed, it returns a status indicator in the SQLCODE variable, which is declared automatically for SQL programs during preprocessing with **gpre**. The following table summarizes possible SQLCODE values and their meanings:

Table 14-1: Possible SQLCODE Values

Value	Meaning
0	Success.
1-99	Warning or informational message.
100	End of file (no more data).
< 0	Error. Statement failed to complete.

To trap and respond to run-time errors, SQLCODE should be checked after each SQL operation. There are three ways to examine SQLCODE and respond to errors:

- Use WHENEVER statements to automate checking SQLCODE and handle errors when they occur.
- Test SQLCODE directly after individual SQL statements.
- Use a judicious combination of WHENEVER statements and direct testing.

Each method has advantages and disadvantages, described fully in the remainder of this chapter.

Handling Errors With WHENEVER Statements

The WHENEVER statement enables all SQL errors to be handled with a minimum of coding. WHENEVER statements specify error-handling code that a program should execute when SQLCODE indicates errors, warnings, or end-of-file. The syntax of WHENEVER is:

```
EXEC SQL
   WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}
      {GOTO label | CONTINUE};
```

After WHENEVER appears in a program, all subsequent SQL statements automatically jump to the specified code location identified by *label* when the appropriate error or warning occurs.

Because they affect all subsequent statements, WHENEVER statements are usually embedded near the start of a program. For example, the first statement in the following C code's **main()** function is a WHENEVER that traps SQL errors:

```
main()
{
   EXEC SQL
      WHENEVER SQLERROR GOTO ErrorExit;
. . .
   Error Exit:
      if (SQLCODE)
      {
         print_error();
         EXEC SQL
            ROLLBACK;
         EXEC SQL
            DISCONNECT;
         exit(1);
      }
}
. . .
print_error()
{
   printf("Database error, SQLCODE = %d\n", SQLCODE);
}
```

Up to three WHENEVER statements can be active at any time:

- WHENEVER SQLERROR is activated when SQLCODE is less than zero, indicating that a statement failed.

- WHENEVER SQLWARNING is activated when SQLCODE contains a value from 1 to 99, inclusive, indicating that while a statement executed, there is some question about the way it succeeded.
- WHENEVER NOT FOUND is activated when SQLCODE is 100, indicating that end-of-file was reached during a FETCH or SELECT.

Omitting a statement for a particular condition means it is not trapped, even if it occurs. For example, if WHENEVER NOT FOUND is left out of a program, then when a FETCH or SELECT encounters the end-of-file, SQLCODE is set to 100, but program execution continues as if no error condition has occurred.

Error conditions also can be overlooked by using the CONTINUE statement inside a WHENEVER statement:

```
. . .
EXEC SQL
   WHENEVER SQLWARNING
      CONTINUE;
. . .
```

This code traps SQLCODE warning values, but ignores them. Ordinarily, warnings should be investigated, not ignored.

Important Use WHENEVER SQLERROR CONTINUE at the start of error-handling routines to disable error handling temporarily. Otherwise, there is a possibility of an infinite loop; should another error occur in the handler itself, the routine will call itself again.

Scope of WHENEVER Statements

WHENEVER only affects all subsequent SQL statements in the *module*, or source code file, where it is defined. In programs with multiple source code files, each module must define its own WHENEVER statements.

Changing Error-handling Routines

To switch to another error-handling routine for a particular error condition, embed another WHENEVER statement in the program at the point where error handling should be changed. The new assignment overrides any previous assignment, and remains in effect until overridden itself. For example, the following program fragment sets an initial jump point for SQLERROR conditions to *ErrorExit1*, then resets it to *ErrorExit2*:

```
EXEC SQL
   WHENEVER SQLERROR
      GOTO ErrorExit1;
```

```
. . .
EXEC SQL
   WHENEVER SQLERROR
      GOTO ErrorExit2;
. . .
```

Limitations of WHENEVER Statements

There are two limitations to WHENEVER. It:

- Traps errors indiscriminately. For example, WHENEVER SQLERROR traps both missing databases and data entry that violates a CHECK constraint, and jumps to a single error-handling routine. While a missing database is a severe error that may require action outside the context of the current program, invalid data entry may be the result of a typing mistake that could be fixed by reentering the data.
- Does not easily enable a program to resume processing at the point where the error occurred. For example, a single WHENEVER SQLERROR can trap data entry that violates a CHECK constraint at several points in a program, but jumps to a single error-handling routine. It might be helpful to allow the user to reenter data in these cases, but the error routine cannot determine where to jump to resume program processing.

Error-handling routines can be very sophisticated. For example, in C or C++, a routine might use a large **case** statement to examine SQLCODE directly and respond differently to different values. Even so, creating a sophisticated routine that can resume processing at the point where an error occurred is difficult. To resume processing after error recovery, consider testing SQLCODE directly after each SQL statement, or consider using a combination of error-handling methods.

Testing SQLCODE Directly

A program can test SQLCODE directly after each SQL statement instead of relying on WHENEVER to trap and handle all errors. The main advantage to testing SQLCODE directly is that custom error-handling routines can be designed for particular situations.

For example, the following C code fragment checks if SQLCODE is not zero after a SELECT statement completes. If so, an error has occurred, so the statements inside the **if** clause are executed. These statements test SQLCODE for two specific values, -1, and 100, handling each differently. If SQLCODE is set to any

other error value, a generic error message is displayed and the program is ended gracefully.

```
EXEC SQL
   SELECT CITY INTO :city FROM STATES
      WHERE STATE = :stat:statind;

if (SQLCODE)
{
   if (SQLCODE == -1)
      printf("too many records found\n");
   else if (SQLCODE == 100)
      printf("no records found\n");
   else
   {
      printf("Database error, SQLCODE = %d\n", SQLCODE);
      EXEC SQL
         ROLLBACK;
      EXEC SQL
         DISCONNECT;
      exit(1);
   }
}
printf("found city named %s\n", city);
EXEC SQL
   COMMIT;
EXEC SQL
   DISCONNECT;
```

The disadvantage to checking SQLCODE directly is that it requires many lines of extra code just to see if an error occurred. On the other hand, it enables errors to be handled with function calls, as the following C code illustrates:

```
EXEC SQL
   SELECT CITY INTO :city FROM STATES
      WHERE STATE = :stat:statind;

switch (SQLCODE)
{
   case 0:
      break; /* NO ERROR */
   case -1
      ErrorTooMany();
      break;
   case 100:
      ErrorNotFound();
      break;
   default:
      ErrorExit(); /* Handle all other errors */
      break;
}
. . .
```

Using function calls for error handling enables programs to resume execution if errors can be corrected.

Combining Error-handling Techniques

Error handling in many programs can benefit from combining WHENEVER with direct checking of SQLCODE. A program might include generic WHENEVER statements for handling most SQL error conditions, but for critical operations, WHENEVER statements might be temporarily overridden to enable direct checking of SQLCODE.

For example, the following C code:

- Sets up generic error handling with three WHENEVER statements.
- Overrides the WHENEVER SQLERROR statement to force program continuation using the CONTINUE clause.
- Checks SQLCODE directly.
- Overrides WHENEVER SQLERROR again to reset it.

```
main()
{
   EXEC SQL
      WHENEVER SQLERROR GOTO ErrorExit; /* trap all errors */
   EXEC SQL
      WHENEVER SQLWARNING GOTO WarningExit; /* trap warnings */
   EXEC SQL
      WHENEVER NOT FOUND GOTO AllDone; /* trap end of file */
. . .
   EXEC SQL
      WHENEVER SQLERROR CONTINUE; /* prevent trapping of errors */
   EXEC SQL
      SELECT CITY INTO :city FROM STATES
         WHERE STATE = :stat:statind;
   switch (SQLCODE)
   {
      case 0:
         break; /* NO ERROR */
      case -1
         ErrorTooMany();
         break;
      case 100:
         ErrorNotFound();
         break;
      default:
         ErrorExitFunction(); /* Handle all other errors */
         break;
   }
   EXEC SQL
```

```
        WHENEVER SQLERROR GOTO ErrorExit; /* reset to trap all errors */
. . .
}
```

Guidelines for Error Handling

The following guidelines apply to all error-handling routines in a program.

Using SQL and Host-language Statements

All SQL statements and InterBase functions can be used in error-handling routines, except for CONNECT.

Any host-language statements and functions can appear in an error-handling routine without restriction.

Important Use WHENEVER SQLERROR CONTINUE at the start of error-handling routines to disable error-handling temporarily. Otherwise, there is a possibility of an infinite loop; should another error occur in the handler itself, the routine will call itself again.

Nesting Error-handling Routines

Although error-handling routines can be nested or called recursively, this practice is not recommended unless the program preserves the original contents of SQLCODE and the InterBase error status array.

Handling Unexpected and Unrecoverable Errors

Even if an error-handling routine catches and handles recoverable errors, it should always contain statements to handle unexpected or unrecoverable errors.

The following code handles unrecoverable errors:

```
. . .
isc_print_sqlerr(SQLCODE, isc_status);
EXEC SQL
   ROLLBACK;
EXEC SQL
   DISCONNECT;
exit(1);
```

Portability

For portability among different SQL implementations, SQL programs should limit error handling to WHENEVER statements or direct examination of SQLCODE values.

InterBase internal error recognition occurs at a finer level of granularity than SQLCODE representation permits. A single SQLCODE value can represent many different internal InterBase errors. Where portability is not an issue, it may be desirable to perform additional InterBase error handling. The remainder of this chapter explains how to use these additional features.

Additional InterBase Error Handling

The same SQLCODE value can be returned by multiple InterBase errors. For example, the SQLCODE value, -901, is generated in response to many different InterBase errors. When portability to other vendors' SQL implementations is not required, SQL programs can sometimes examine the InterBase error status array, **isc_status**, for more specific error information.

isc_status is an array of twenty elements of type ISC_STATUS. It is declared automatically for programs when they are preprocessed with **gpre**. Two kinds of InterBase error information are reported in the status array:

- InterBase error message components.
- InterBase error numbers.

As long as the current SQLCODE value is not -1, 0, or 100, error-handling routines that examine the error status array can do any of the following:

- Display SQL and InterBase error messages.
- Capture SQL and InterBase error messages to a storage buffer for further manipulation.
- Trap for and respond to particular InterBase error codes.

The InterBase error status array is usually examined only *after* trapping errors with WHENEVER or testing SQLCODE directly.

Displaying Error Messages

If SQLCODE is less than -1, additional InterBase error information can be displayed using the InterBase **isc_print_sqlerror()** function inside an error-handling routine. During preprocessing with **gpre**, this function is automatically declared for InterBase applications.

isc_print_sqlerror() displays the SQLCODE value, a related SQL error message, and any InterBase error messages in the status array. It requires two parameters: SQLCODE, and a pointer to the error status array, **isc_status**.

For example, when an error occurs, the following code displays the value of SQLCODE, displays a corresponding SQL error message, then displays additional InterBase error message information if possible:

```
. . .
EXEC SQL
   SELECT CITY INTO :city FROM STATES
      WHERE STATE = :stat:statind;
if(SQLCODE)
{
   isc_print_sqlerror(SQLCODE, isc_status);
   EXEC SQL
      ROLLBACK;
   EXEC SQL
      DISCONNECT ALL;
   exit(1);
}
. . .
```

Important Some windowing systems do not encourage or permit direct screen writes. Do not use **isc_print_sqlerror()** when developing applications for these environments. Instead, use **isc_sql_interprete()** and **isc_interprete()** to capture messages to a buffer for display.

Capturing SQL Error Messages With isc_sql_interprete()

Instead of displaying SQL error messages, an application can capture the text of those messages in a buffer by using **isc_sql_interprete()**. Capture messages in a buffer when applications:

- Run under windowing systems that do not permit direct writing to the screen.
- Store a record of all error messages in a log file.
- Manipulate or format error messages for display.

Given SQLCODE, a pointer to a storage buffer, and the maximum size of the buffer in bytes, **isc_sql_interprete()** builds an SQL error message string, and puts the formatted string in the buffer where it can be manipulated. A buffer size of 256 bytes is large enough to hold any SQL error message.

For example, the following code stores an SQL error message in *err_buf*, then writes *err_buf* to a log file. The code assumes that iff an error occurs during the processing of an SQL statement, the log file is properly declared and opened inside **main()** *before* SQL statements are executed:

```
. . .
char err_buf[256]; /* error message buffer for isc_sql_interprete() */
FILE *efile; /* code fragment assumes pointer to an open file */
. . .
EXEC SQL
   SELECT CITY INTO :city FROM STATES
      WHERE STATE = :stat:statind;
if (SQLCODE)
{
   isc_sql_interprete(SQLCODE, err_buf, sizeof(err_buf));
   fprintf(efile, "%s\n", err_buf); /* write buffer to log file */
   EXEC SQL
      ROLLBACK; /* undo database changes */
   EXEC SQL
      DISCONNECT ALL; /* close open databases */
   exit(1); /* exit with error flag set */
}
. . .
```

isc_sql_interprete() retrieves and formats a single message corresponding to a given SQLCODE. When SQLCODE is less than -1, more specific InterBase error information is available. It, too, can be retrieved, formatted, and stored in a buffer by using the **isc_interprete()** function.

Capturing InterBase Error Messages With isc_interprete()

The text of InterBase error messages can be captured in a buffer by using **isc_interprete()**. Capture messages in a buffer when applications:

- Run under windowing systems that do not permit direct writing to the screen.
- Store a record of all error messages in a log file.
- Manipulate or format error messages for display.

Important **isc_interprete()** should not be used unless SQLCODE is less than -1 because the contents of **isc_status** may not contain reliable error information in these cases.

Given both the location of a storage buffer previously allocated by the program, and a pointer to the start of the status array, **isc_interprete()** builds an error message string from the information in the status array, and puts the formatted string in the buffer where it can be manipulated. It also advances the status array pointer to the start of the next cluster of available error information.

isc_interprete() retrieves and formats a single error message each time it is called. When an error occurs in an InterBase program, however, the status array may contain more than one error message. To retrieve all relevant error messages, error-handling routines should repeatedly call **isc_interprete()** until it returns no more messages.

Because **isc_interprete()** modifies the pointer to the status array that it receives, do *not* pass **isc_status** directly to it. Instead, declare a pointer to **isc_status**, then pass the pointer to **isc_interprete()**.

The following C code fragment illustrates how InterBase error messages can be captured to a log file, and demonstrates the proper declaration of a string buffer and pointer to **isc_status**. It assumes the log file is properly declared and opened *before* control is passed to the error-handling routine. It also demonstrates how to set the pointer to the start of the status array in the error-handling routine *before* **isc_interprete()** is first called.

```
. . .
#include "ibase.h";
. . .
main()
{
char msg[512];
ISC_STATUS *vector;
FILE *efile; /* code fragment assumes pointer to an open file */
. . .
if (SQLCODE < -1)
   ErrorExit();
}
. . .

ErrorExit()
{
   vector = isc_status; /* (re)set to start of status vector */
   isc_interprete(msg, &vector); /* retrieve first mesage */
   fprintf(efile, "%s\n", msg); /* write buffer to log file */
   msg[0] = '-'; /* append leading hyphen to secondary messages */
   while (isc_interprete(msg + 1, &vector)) /* more?*/
      fprintf(efile, "%s\n", msg); /* if so, write it to log */
   fclose(efile); /* close log prior to quitting program */
   EXEC SQL
      ROLLBACK;
   EXEC SQL
      DISCONNECT ALL;
```

```
        exit(1); /* quit program with an 'abnormal termination' code */
    }
    . . .
```

In this example, the error-handling routine performs the following tasks:

- Sets the error array pointer to the starting address of the status vector, **isc_status**.
- Calls **isc_interprete()** a single time to retrieve the first error message from the status vector.
- Writes the first message to a log file.
- Makes repeated calls to **isc_interprete()** within a **while** loop to retrieve any additional messages. If additional messages are retrieved, they are also written to the log file.
- Rolls back the transaction.
- Closes the database and releases system resources.

Trapping and Handling InterBase Error Codes

Whenever SQLCODE is less than -1, the error status array, **isc_status**, may contain detailed error information specific to InterBase, including *error codes*, numbers that uniquely identify each error. With care, error-handling routines can trap for and respond to specific codes.

To trap and handle InterBase error codes in an error-handling routine, follow these steps:

1. Check SQLCODE to be sure it is less than -1.
2. Check that the first element of the status array is set to **isc_arg_gds**, indicating that an InterBase error code is available. In C programs, the first element of the status array is **isc_status[0]**.

   Do *not* attempt to handle errors reported in the status array if the first status array element contains a value other than 1.
3. If SQLCODE is less than -1 and the first element in **isc_status** is set to **isc_arg_gds**, use the actual InterBase error code in the second element of **isc_status** to branch to an appropriate routine for that error.

Tip InterBase error codes are mapped to mnemonic definitions (for example, **isc_arg_gds**) that can be used in code to make it easier to read, understand, and maintain. Definitions for all InterBase error codes can be found in the *ibase.h* file.

The following C code fragment illustrates an error-handling routine that:

- Displays error messages with **isc_print_sqlerror()**.
- Illustrates how to parse for and handle six specific InterBase errors which might be corrected upon roll back, data entry, and retry.
- Uses mnemonic definitions for InterBase error numbers.

```
. . .
int c, jval, retry_flag = 0;
jmp_buf jumper;
. . .
main()
{
   . . .
   jval = setjmp(jumper);
   if (retry_flag)
      ROLLBACK;
   . . .
}
int ErrorHandler(void)
{
   retry_flag = 0; /* reset to 0, no retry */
   isc_print_sqlerror(SQLCODE, isc_status); /* display errors */
   if (SQLCODE < -1)
   {
      if (isc_status[0] == isc_arg_gds)
      {
         switch (isc_status[1])
         {
            case isc_convert_error:
            case isc_deadlock:
            case isc_integ_fail:
            case isc_lock_conflict:
            case isc_no_dup:
            case isc_not_valid:
               printf("\n Do you want to try again? (Y/N)");
               c = getchar();
               if (c == 'Y' || c == 'y')
               {
                           retry_flag = 1; /* set flag to retry */
                           longjmp(jumper, 1);
               }
               break;
            case isc_end_arg: /* there really isn't an error */
               retry_flag = 1; /* set flag to retry */
               longjump(jumper, 1);
               break;
            default: /* we can't handle everything, so abort */
               break;
         }
      }
```

```
    }
        EXEC SQL
            ROLLBACK;
        EXEC SQL
            DISCONNECT ALL;
        exit(1);
}
```

CHAPTER 15

Using Dynamic SQL

This chapter describes how to write dynamic SQL applications, applications that elicit or build SQL statements for execution at run time.

In many database applications, a programmer specifies exactly which SQL statements to execute against a particular database. When the application is compiled, these statements become fixed. In some database application it is useful to build and execute statements from text string fragments or from strings elicited from the user at run time. These applications require the capability to create and execute SQL statements dynamically at run time. Dynamic SQL (DSQL) provides this capability. For example, the InterBase **isql** utility is a DSQL application.

Overview of the DSQL Programming Process

Building and executing DSQL statements involves the following general steps:

- Embedding SQL statements that support DSQL processing in an application.
- Using host-language facilities, such as data types and macros, to provide input and output areas for passing statements and parameters at run time.
- Programming methods that use these statements and facilities to process SQL statements at run time.

These steps are described in detail throughout this chapter.

DSQL Limitations

Although DSQL offers many advantages, it also has the following limitations:

- Access to one database at a time.
- Dynamic transaction processing is not permitted; all named transactions must be declared at compile time.
- Dynamic access to BLOB and array data is not supported; BLOB and array data can be accessed, but only through standard, statically processed SQL statements, or through low-level API calls.
- Database creation is restricted to CREATE DATABASE statements executed within the context of EXECUTE IMMEDIATE.

For more information about database access in DSQL, see "Accessing Databases," in this chapter. For more information about handling transactions in DSQL applications, see "Handling Transactions," in this chapter. For more information about working with BLOB data in DSQL, see "Processing BLOB Data," in this chapter. For more information about handling array data in DSQL, see "Processing Array Data," in this chapter. For more information about dynamic creation of databases, see "Creating a Database," in this chapter.

Accessing Databases

Using standard SQL syntax, a DSQL application can only use one database handle per source file module, and can, therefore, only be connected to a single database at a time. Database handles must be declared and initialized when an application is preprocessed with **gpre**. For example, the following code creates a single handle, *db1*, and initializes it to zero:

```
#include "ibase.h"
isc_db_handle db1;
. . .
db1 = 0L;
```

After a database handle is declared and initialized, it can be assigned dynamically to a database at run time as follows:

```
char dbname[129];
. . .
prompt_user("Name of database to open: ");
gets(dbname);
EXEC SQL
   SET DATABASE db1 = :dbname;
EXEC SQL
   CONNECT db1;
. . .
```

The database accessed by DSQL statements is always the last database handle mentioned in a SET DATABASE command. A database handle can be used to

connect to different databases as long as a previously connected database is first disconnected with DISCONNECT. DISCONNECT automatically sets database handles to NULL. The following statements disconnect from a database, zero the database handle, and connect to a new database:

```
EXEC SQL
   DISCONNECT db1;
EXEC SQL
   SET DATABASE db1 = "employee.gdb";
EXEC SQL
   CONNECT db1;
```

To access more than one database using DSQL, create a separate source file module for each database, and use low-level API calls to attach to the databases and access data. For more information about accessing databases with API calls, see the *API Guide*. For more information about SQL database statements, see Chapter 3: "Working With Databases."

Handling Transactions

InterBase requires that all transaction names be declared when an application is preprocessed with **gpre**. Once fixed at precompile time, transaction handles cannot be changed at run time, nor can new handles be declared dynamically at run time.

SQL statements such as PREPARE, DESCRIBE, EXECUTE, and EXECUTE IMMEDIATE, can be coded at precompile time to include an optional TRANSACTION clause specifying which transaction controls statement execution. The following code declares, initializes, and uses a transaction handle in a statement that processes a run-time DSQL statement:

```
#include "ibase.h"
isc_tr_handle t1;
. . .
t1 = 0L;
EXEC SQL
   SET TRANSACTION NAME t1;
EXEC SQL
   PREPARE TRANSACTION t1 Q FROM :sql_buf;
```

DSQL statements that are processed with PREPARE, DESCRIBE, EXECUTE, and EXECUTE IMMEDIATE cannot use a TRANSACTION clause, even if it is permitted in standard, embedded SQL.

The SET TRANSACTION statement cannot be prepared, but it can be processed with EXECUTE IMMEDIATE if:

1. Previous transactions are first committed or rolled back.

2. The transaction handle is set to NULL.

For example, the following statements commit the previous default transaction, then start a new one with EXECUTE IMMEDIATE:

```
EXEC SQL
   COMMIT;
/* set default transaction name to NULL */
gds__trans = NULL;
EXEC SQL
   EXECUTE IMMEDIATE "SET TRANSACTION READ ONLY";
```

Creating a Database

To create a new database in a DSQL application:

1. Disconnect from any currently attached databases. Disconnecting from a database automatically sets its database handle to NULL.
2. Build the CREATE DATABASE statement to process.
3. Execute the statement with EXECUTE IMMEDIATE.

For example, the following statements disconnect from any currently connected databases, and create a new database. Any existing database handles are set to NULL, so that they can be used to connect to the new database in future DSQL statements.

```
char *str = "CREATE DATABASE \"new_emp.gdb\"";
. . .
EXEC SQL
   DISCONNECT ALL;
EXEC SQL
   EXECUTE IMMEDIATE :str;
```

Processing BLOB Data

DSQL does not directly support BLOB processing. BLOB cursors are not supported in DSQL. DSQL applications can use API calls to process BLOB data. For more information about BLOB API calls, see the *API Guide.*

Processing Array Data

DSQL does not directly support array processing. DSQL applications can use API calls to process array data. For more information about array API calls, see the *API Guide.*

Writing a DSQL Application

Write a DSQL application when any of the following are not known until run time:

- The text of the SQL statement
- The number of host variables
- The data types of host variables
- References to database objects

Writing a DSQL application is usually more complex than programming with regular SQL because for most DSQL operations, the application needs explicitly to allocate and process an extended SQL descriptor area (XSQLDA) data structure to pass data to and from the database.

To use DSQL to process an SQL statement, follow these basic steps:

1. Determine if DSQL can process the SQL statement.
2. Represent the SQL statement as a character string in the application.
3. If necessary, allocate one or more XSQLDAs for input parameters and return values.
4. Use an appropriate DSQL programming method to process the SQL statement.

Determining if DSQL Can Process an SQL Statement

DSQL can process most SQL statements. For example, DSQL can process data manipulation statements such as DELETE and INSERT, data definition statements such as ALTER TABLE and CREATE INDEX, and SELECT statements.

The following table lists SQL statements that cannot be processed by DSQL:

Table 15-1: SQL Statements That Cannot Be Processed By DSQL

Statement	Statement
CLOSE	DECLARE CURSOR
DESCRIBE	EXECUTE
EXECUTE IMMEDIATE	FETCH
OPEN	PREPARE

These SQL statements are used to process DSQL requests or to handle SQL cursors, which must always be specified when an application is written. Attempting to use them with DSQL results in run-time errors.

For more information about a statement's availability in DSQL, see the *Language Reference*.

Representing an SQL Statement as a Character String

Within a DSQL application, an SQL statement can come from different sources. It can come directly from a user who enters a statement at a prompt, as does **isql**. Or it can be generated by the application in response to user interaction. Whatever the source of the SQL statement it must be represented as an *SQL statement string*, a character string that is passed to DSQL for processing.

Because SQL statement strings are C character strings that are processed directly by DSQL, they cannot begin with the EXEC SQL prefix or end with a semicolon (;). The semicolon is, of course, the appropriate terminator for the C string declaration itself. For example, the following host-language variable declaration is a valid SQL statement string:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = 256";
```

Specifying Parameters in SQL Statement Strings

SQL statement strings often include *value parameters*, expressions that evaluate to a single numeric or character value. Parameters can be used anywhere in statement strings where SQL expects a value that is not the name of a database object.

A value parameter in a statement string can be passed as a constant, or passed as a placeholder at run time. For example, the following statement string passes 256 as a constant:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = 256";
```

It is also possible to build strings at run time from a combination of constants. This method is useful for statements where the variable is not a true constant, or it is a table or column name, and where the statement is executed only once in the application.

To pass a parameter as a placeholder, the value is passed as a question mark (?) embedded within the statement string:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = ?";
```

When DSQL processes a statement containing a placeholder, it replaces the question mark with a value supplied in the XSQLDA. Use placeholders in statements that are prepared once, but executed many times with different parameter values.

Replaceable value parameters are often used to supply values in WHERE clause comparisons and in the UPDATE statement SET clause.

Understanding the XSQLDA

All DSQL applications must declare one or more extended SQL descriptor areas (XSQLDAs). The XSQLDA structure definition can be found in the *ibase.h* header file in the InterBase *include* directory. Applications declare instances of the XSQLDA for use.

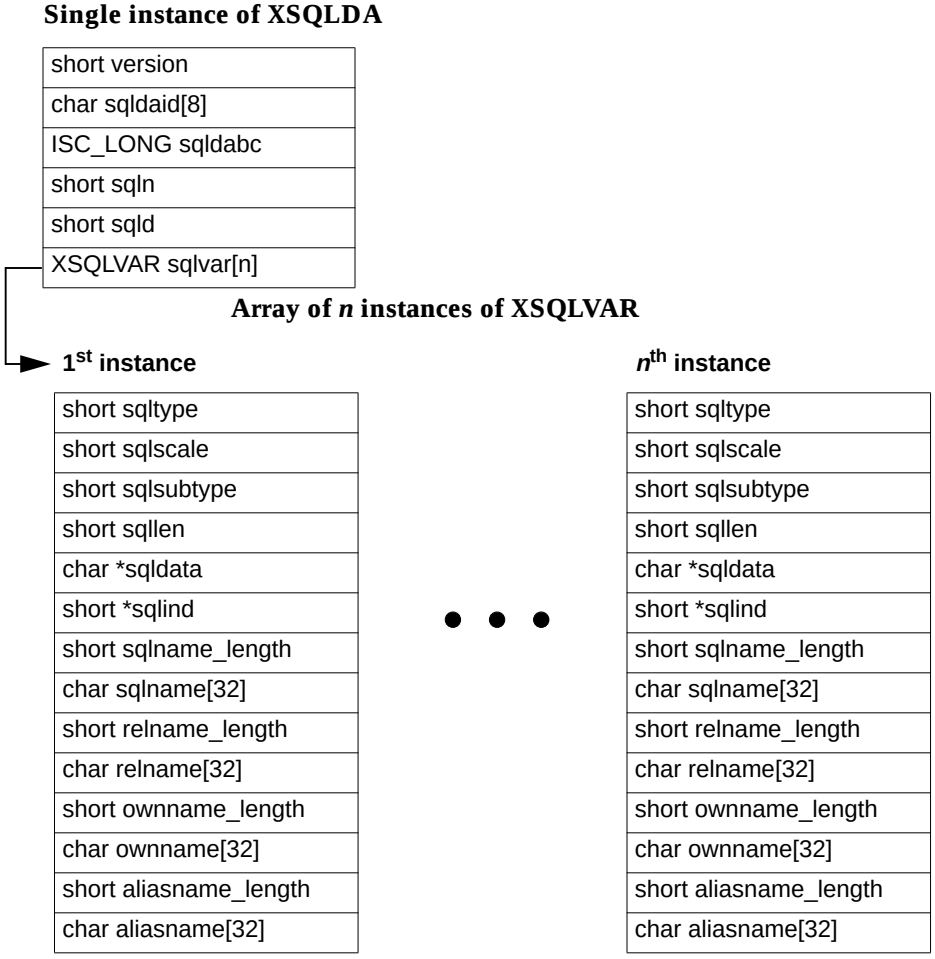
The XSQLDA is a host-language data structure that DSQL uses to transport data to or from a database when processing an SQL statement string. There are two types of XSQLDAs: *input* descriptors and *output* descriptors. Both input and output descriptors are implemented using the XSQLDA structure.

One field in the XSQLDA, the XSQLVAR, is especially important, because one XSQLVAR must be defined for each input parameter or column returned. Like the XSQLDA, the XSQLVAR is a structure defined in *ibase.h* in the InterBase *include* directory.

Applications do not declare instances of the XSQLVAR ahead of time, but must, instead, dynamically allocate storage for the proper number of XSQLVAR structures required for each DSQL statement before it is executed, then deallocate it, as appropriate, after statement execution.

The following figure illustrates the relationship between the XSQLDA and the XSQLVAR:

Figure 15-1: XSQLDA and XSQLVAR Relationship



An input XSQLDA consists of a single XSQLDA structure, and one XSQLVAR structure for each input parameter. An output XSQLDA also consists of one XSQLDA structure and one XSQLVAR structure for each data item returned by the statement. An XSQLDA and its associated XSQLVAR structures are allocated as a single block of contiguous memory.

The PREPARE and DESCRIBE statements can be used to determine the proper number of XSQLVAR structures to allocate, and the XSQLDA_LENGTH macro can be used to allocate the proper amount of space. For more information about

the XSQLDA_LENGTH macro, see “Using the XSQLDA_LENGTH Macro,” in this chapter.

XSQLDA Field Descriptions

The following table describes the fields that comprise the XSQLDA structure:

Table 15-2: XSQLDA Field Descriptions

Field Definition	Description
short version	Indicates the version of the XSQLDA structure. Set by an application. The current version is defined in *ibase.h* as SQLDA_VERSION1.
char sqldaid[8]	Reserved for future use.
ISC_LONG sqldabc	Reserved for future use.
short sqln	Indicates the number of elements in the *sqlvar* array. Set by the application. Whenever the application allocates storage for a descriptor, it should set this field.
short sqld	Indicates the number of parameters (for an input XSQLDA), or the number of select-list items (for an output XSQLDA). Set by InterBase during a DESCRIBE or PREPARE. For an input descriptor, an *sqld* of 0 indicates that the SQL statement has no parameters. For an output descriptor, an *sqld* of 0 indicates that the SQL statement is not a SELECT statement.
XSQLVAR sqlvar	The array of XSQLVAR structures. The number of elements in the array is specified in the *sqln* field.

XSQLVAR Field Descriptions

The following table describes the fields that comprise the XSQLVAR structure:

Table 15-3: XSQLVAR Field Descriptions

Field Definition	Description
short sqltype	Indicates the SQL data type of parameters or select-list items. Set by InterBase during PREPARE or DESCRIBE.
short sqlscale	Provides scale, specified as a negative number, for exact numeric data types (DECIMAL, NUMERIC). Set by InterBase during PREPARE or DESCRIBE.
short sqlsubtype	Specifies the subtype for BLOB data. Set by InterBase during PREPARE or DESCRIBE.

Table 15-3: XSQLVAR Field Descriptions (Continued)

Field Definition	Description
short sqllen	Indicates the maximum size, in bytes, of data in the *sqldata* field. Set by InterBase during PREPARE or DESCRIBE.
char *sqldata	For input descriptors, specifies either the address of a select-list item or a parameter. Set by the application. For output descriptors, contains a value for a select-list item. Set by InterBase.
short *sqlind	On input, specifies the address of an indicator variable. Set by an application. On output, specifies the address of column indicator value for a select-list item following a FETCH. A value of 0 indicates that the column is not NULL; a value of -1 indicates the column is NULL. Set by InterBase.
short sqlname_length	Specifies the length, in bytes, of the data in field, *sqlname*. Set by InterBase during DESCRIBE OUTPUT.
char sqlname[32]	Contains the name of the column. Not null (\0) terminated. Set by InterBase during DESCRIBE OUTPUT.
short relname_length	Specifies the length, in bytes, of the data in field, *relname*. Set by InterBase during DESCRIBE OUTPUT.
char relname[32]	Contains the name of the table. Not null (\0) terminated. Set by InterBase during DESCRIBE OUTPUT.
short ownname_length	Specifies the length, in bytes, of the data in field, *ownname*. Set by InterBase during DESCRIBE OUTPUT.
char ownname[32]	Contains the owner name of the table. Not null (\0) terminated. Set by InterBase during DESCRIBE OUTPUT.
short aliasname_length	Specifies the length, in bytes, of the data in field, *aliasname*. Set by InterBase during DESCRIBE OUTPUT.
char aliasname[32]	Contains the alias name of the column. If no alias exists, contains the column name. Not null (\0) terminated. Set by InterBase during DESCRIBE OUTPUT.

Input Descriptors

Input descriptors process SQL statement strings that contain parameters. Before an application can execute a statement with parameters, it must supply values for them. The application indicates the number of parameters passed in the XSQLDA *sqld* field, then describes each parameter in a separate XSQLVAR structure. For example, the following statement string contains two parameters, so an application must set *sqld* to 2, and describe each parameter:

```
char *str = "UPDATE DEPARTMENT SET BUDGET = ? WHERE LOCATION = ?";
```

When the statement is executed, the first XSQLVAR supplies information about the BUDGET value, and the second XSQLVAR supplies the LOCATION value.

For more information about using input descriptors, see “DSQL Programming Methods,” in this chapter.

Output Descriptors

Output descriptors return values from an executed query to an application. The *sqld* field of the XSQLDA indicates how many values were returned. Each value is stored in a separate XSQLVAR structure. The XSQLDA *sqlvar* field points to the first of these XSQLVAR structures. The following statement string requires an output descriptor:

```
char *str = "SELECT * FROM CUSTOMER WHERE CUST_NO > 100";
```

For information about retrieving information from an output descriptor, see “DSQL Programming Methods,” in this chapter.

Using the XSQLDA_LENGTH Macro

The *ibase.h* header file defines a macro, XSQLDA_LENGTH, to calculate the number of bytes that must be allocated for an input or output XSQLDA. XSQLDA_LENGTH is defined as follows:

```
#define XSQLDA_LENGTH (n) (sizeof (XSQLDA) + (n- 1) * sizeof(XSQLVAR))
```

n is the number of parameters in a statement string, or the number of select-list items returned from a query. For example, the following C statement uses the XSQLDA_LENGTH macro to specify how much memory to allocate for an XSQLDA with 5 parameters or return items:

```
XSQLDA *my_xsqlda;
. . .
my_xsqlda = (XSQLDA *) malloc(XSQLDA_LENGTH(5));
. . .
```

For more information about using the XSQLDA_LENGTH macro, see “DSQL Programming Methods,” in this chapter.

SQL Data Type Macro Constants

InterBase defines a set of macro constants to represent SQL data types and NULL status information in an XSQLVAR. An application should use these macro constants to specify the data type of parameters and to determine the data

types of select-list items in an SQL statement. The following table lists each SQL data type, its corresponding macro constant expression, C data type or InterBase typedef, and whether or not the *sqlind* field is used to indicate a parameter or variable that contains NULL or unknown data:

Table 15-4: SQL Data Types, Macro Expressions, and C Data Types

SQL Data Type	Macro Expression	C Data Type or typedef	*sqlind* Used?
Array	SQL_ARRAY	ISC_QUAD	No
Array	SQL_ARRAY + 1	ISC_QUAD	Yes
BLOB	SQL_BLOB	ISC_QUAD	No
BLOB	SQL_BLOB + 1	ISC_QUAD	Yes
CHAR	SQL_TEXT	char[ ]	No
CHAR	SQL_TEXT + 1	char[ ]	Yes
DATE	SQL_DATE	ISC_QUAD	No
DATE	SQL_DATE + 1	ISC_QUAD	Yes
DECIMAL	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
DECIMAL	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
DOUBLE PRECISON	SQL_DOUBLE	double	No
DOUBLE PRECISION	SQL_DOUBLE + 1	double	Yes
INTEGER	SQL_LONG	long	No
INTEGER	SQL_LONG + 1	long	Yes
FLOAT	SQL_FLOAT	float	No
FLOAT	SQL_FLOAT + 1	float	Yes
NUMERIC	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
NUMERIC	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
SMALLINT	SQL_SHORT	short	No
SMALLINT	SQL_SHORT + 1	short	Yes
VARCHAR	SQL_VARYING	First 2 bytes: short containing the length of the character string. Remaining bytes: char[ ]	No
VARCHAR	SQL_VARYING + 1	First 2 bytes: short containing the length of the character string. Remaining bytes: char[ ]	Yes

Note DECIMAL and NUMERIC data types are stored internally as SMALLINT, INTEGER, or DOUBLE PRECISION data types. To specify the correct macro expression to provide for a DECIMAL or NUMERIC column, use **isql** to examine the column definition in the table to see how InterBase is storing column data, then choose a corresponding macro expression.

The data type information for a parameter or select-list item is contained in the *sqltype* field of the XSQLVAR structure. The value contained in the *sqltype* field provides two pieces of information:

- The data type of the parameter or select-list item.
- Whether *sqlind* is used to indicate NULL values. If *sqlind* is used, its value specifies whether the parameter or select-list item is NULL (-1), or not NULL (0).

For example, if the *sqltype* field equals SQL_TEXT, the parameter or select-list item is a CHAR that does not use *sqlind* to check for a NULL value (because, in theory, NULL values are not allowed for it). If *sqltype* equals SQL_TEXT + 1, then *sqlind* can be checked to see if the parameter or select-list item is NULL.

Tip The C language expression, *sqltype & 1*, provides a useful test of whether a parameter or select-list item can contain a NULL. The expression evaluates to 0 if the parameter or select-list item cannot contain a NULL, and 1 if the parameter or select-list item can contain a NULL. The following code fragment demonstrates how to use the expression:

```
if (sqltype & 1 == 0)
{
 /* parameter or select-list item that CANNOT contain a NULL */
}
else
{
 /* parameter or select-list item CAN contain a NULL */
}
```

By default, both PREPARE INTO and DESCRIBE return a macro expression of type + 1, so the *sqlind* should always be examined for NULL values with these statements.

Handling Varying String Data Types

VARCHAR, CHARACTER VARYING, and NCHAR VARYING data types require careful handling in DSQL. The first two bytes of these data types contain string length information, while the remainder of the data contains the actual bytes of string data to process.

To avoid having to write code to extract and process variable-length strings in an application, it is possible to force these data types to fixed length using SQL macro expressions. For more information about forcing variable-length data to fixed length for processing, see “Coercing Data Types,” in this chapter.

Applications can, instead, detect and process variable-length data directly. To do so, they must extract the first two bytes from the string to determine the byte-length of the string itself, then read the string, byte-by-byte, into a null-terminated buffer.

Handling NUMERIC and DECIMAL Data Types

DECIMAL and NUMERIC data types are stored internally as SMALLINT, INTEGER, or DOUBLE PRECISION data types, depending on the precision and scale defined for a column definition that uses these types. To determine how a DECIMAL or NUMERIC value is actually stored in the database, use **isql** to examine the column definition in the table. If NUMERIC is reported, then data is actually being stored as DOUBLE PRECISION.

When a DECIMAL or NUMERIC value is stored as a SMALLINT or INTEGER, the value is stored as a whole number. During retrieval in DSQL, the *sqlscale* field of the XSQLVAR is set to a negative number that indicates the factor of ten by which the whole number (returned in *sqldata*), must be divided in order to produce the correct NUMERIC or DECIMAL value with its fractional part. If *sqlcale* is -1, then the number must be divided by 10, if it is -2, then the number must be divided by 100, -3 by 1,000, and so forth.

Coercing Data Types

Sometimes when processing DSQL input parameters and select-list items, it is desirable or necessary to translate one data type to another. This process is referred to as *data type coercion*. For example, data type coercion is often used when parameters or select-list items are of type VARCHAR. The first two bytes of VARCHAR data contain string length information, while the remainder of the data is the string to process. By coercing the data from SQL_VARYING to SQL_TEXT, data processing can be simplified.

Coercion can only be from one compatible data type to another. For example, SQL_VARYING to SQL_TEXT, or SQL_SHORT to SQL_LONG.

Coercing Character Data Types

To coerce SQL_VARYING data types to SQL_TEXT data types, change the *sqltype* field in the parameter's or select-list item's XSQLVAR structure to the desired SQL macro data type constant. For example, the following statement assumes that *var* is a pointer to an XSQLVAR structure, and that it contains an SQL_VARYING data type to convert to SQL_TEXT:

```
var->sqltype = SQL_TEXT;
```

After coercing a character data type, provide proper storage space for it. The XSQLVAR field, *sqllen*, contains information about the size of the uncoerced data. Set the XSQLVAR *sqldata* field to the address of the data.

Coercing Numeric Data Types

To coerce one numeric data type to another, change the *sqltype* field in the parameter's or select-list item's XSQLVAR structure to the desired SQL macro data type constant. For example, the following statement assumes that *var* is a pointer to an XSQLVAR structure, and that it contains an SQL_SHORT data type to convert to SQL_LONG:

```
var->sqltype = SQL_LONG;
```

Important Do not coerce a larger data type to a smaller one. Data can be lost in such a translation.

Setting a NULL Indicator

If a parameter or select-list item can contain a NULL value, the *sqlind* field is used to indicate its NULL status. Appropriate storage space must be allocated for *sqlind* before values can be stored there.

On insertion, set *sqlind* to -1 to indicate that NULL values are legal. Otherwise set *sqlind* to 0.

On selection, an *sqlind* of -1 indicates a field contains a NULL value. Other values indicate a field contains non-NULL data.

Aligning Numerical Data

Ordinarily, when a variable with a numeric data type is created, the compiler will ensure that the variable is stored at a properly aligned address, but when numeric data is stored in a dynamically allocated buffer space, such as can be

pointed to by the XSQLDA and XSQLVAR structures, the programmer must take precautions to ensure that the storage space is properly aligned.

Certain platforms, in particular those with RISC processors, require that numeric data in dynamically allocated storage structures be aligned properly in memory. Alignment is dependent both on data type and platform.

For example, a short integer on a Sun SPARCstation must be located at an address divisible by 2, while a long on the same platform must be located at an address divisible by 4. In most cases, a data item is properly aligned if the address of its starting byte is divisible by the correct alignment number. Consult specific system and compiler documentation for alignment requirements.

A useful rule of thumb is that the size of a data type is always a valid alignment number for the data type. For a given type T, if size of (T) equals *n*, then addresses divisible by *n* are correctly aligned for T. The following macro expression can be used to align data:

```
#define ALIGN(ptr, n) ((ptr + n - 1) & ~(n - 1))
```

where *ptr* is a pointer to char.

The following code illustrates how the ALIGN macro might be used:

```
char *buffer_pointer, *next_aligned;

next_aligned = ALIGN(buffer_pointer, sizeof(T));
```

DSQL Programming Methods

There are four possible DSQL programming methods for handling an SQL statement string. The best method for processing a string depends on the type of SQL statement in the string, and whether or not it contains placeholders for parameters. The following decision table explains how to determine the appropriate processing method for a given string.

Table 15-5: SQL Statement Strings and Recommended Processing Methods

Is it a query?	Does it have placeholders?	Processing method to use:
No	No	Method 1
No	Yes	Method 2
Yes	No	Method 3
Yes	Yes	Method 4

Method 1: Non-query Statements Without Parameters

There are two ways to process an SQL statement string containing a non-query statement without placeholder parameters:

- Use EXECUTE IMMEDIATE to prepare and execute the string a single time.
- Use PREPARE to parse the statement for execution and assign it a name, then use EXECUTE to carry out the statement's actions as many times as required in an application.

Using EXECUTE IMMEDIATE

To execute a statement string a single time, use EXECUTE IMMEDIATE:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string:

```
char *str = "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
```

2. Parse and execute the statement string using EXECUTE IMMEDIATE:

```
EXEC SQL
   EXECUTE IMMEDIATE :str;
```

Note EXECUTE IMMEDIATE also accepts string literals. For example,

```
EXEC SQL
   EXECUTE IMMEDIATE
   "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
```

Using PREPARE and EXECUTE

To execute a statement string several times, use PREPARE and EXECUTE:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string:

```
char *str = "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
```

2. Parse and name the statement string with PREPARE. The name is used in subsequent calls to EXECUTE:

```
EXEC SQL
   PREPARE SQL_STMT FROM :str;
```

SQL_STMT is the name assigned to the parsed statement string.

Note PREPARE also accepts string literals. For example,

```
EXEC SQL
   PREPARE SQL_STMT FROM
   "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
```

3. Execute the named statement string using EXECUTE. For example, the following statement executes a statement string named SQL_STMT:

   ```
   EXEC SQL
      EXECUTE SQL_STMT;
   ```

   Once a statement string is prepared, it can be executed as many times as required in an application.

Method 2: Non-query Statements With Parameters

There are two steps to processing an SQL statement string containing a non-query statement with placeholder parameters:

1. Creating an input XSQLDA to process a statement string's parameters.
2. Preparing and executing the statement string with its parameters.

Creating the Input XSQLDA

Placeholder parameters are replaced with actual data before a prepared SQL statement string is executed. Because those parameters are unknown when the statement string is created, an input XSQLDA must be created to supply parameter values at execute time. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *in_sqlda*:

   ```
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   In this statement space for 10 XSQLVAR structures is allocated, allowing the XSQLDA to accommodate up to 10 parameters.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field to indicate the number of XSQLVAR structures allocated:

   ```
   in_sqlda_version = SQLDA_VERSION1;
   in_sqlda->sqln = 10;
   ```

Preparing and Executing a Statement String With Parameters

After an XSQLDA is created for holding a statement string's parameters, the statement string can be created and prepared. Local variables corresponding to the placeholder parameters in the string must be assigned to their corresponding *sqldata* fields in the XSQLVAR structures.

To prepare and execute a non-query statement string with parameters, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string with placeholder parameters:

   ```
   char *str = "UPDATE DEPARTMENT SET BUDGET = ?, LOCATION = ?";
   ```

   This statement string contains two parameters: a value to be assigned to the BUDGET field and a value to be assigned to the LOCATION field.

2. Parse and name the statement string with PREPARE. The name is used in subsequent calls to DESCRIBE and EXECUTE:

   ```
   EXEC SQL
      PREPARE SQL_STMT FROM :str;
   ```

   SQL_STMT is the name assigned to the prepared statement string.

3. Use DESCRIBE INPUT to fill the input XSQLDA with information about the parameters contained in the SQL statement:

   ```
   EXEC SQL
      DESCRIBE INPUT SQL_STMT USING SQL DESCRIPTOR in_sqlda;
   ```

4. Compare the value of the *sqln* field of the XSQLDA to the value of the *sqld* field to make sure enough XSQLVARs are allocated to hold information about each parameter. *sqln* should be at least as large as *sqln*. If not,

free the storage previously allocated to the input descriptor, reallocate storage to reflect the number of parameters specified by *sqld*, reset *sqln* and *version*, then execute DESCRIBE INPUT again:

```
if (in_sqlda->sqld > in_sqlda->sqln)
{
   n = in_sqlda->sqld;
   free(in_sqlda);
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
   in_sqlda->sqln = n;
   in_sqlda->version = SQLDA_VERSION1;
   EXEC SQL
      DESCRIBE INPUT SQL_STMT USING SQL DESCRIPTOR in_sqlda;
}
```

5. Process each XSQLVAR parameter structure in the XSQLDA. Processing a parameter structure involves up to four steps:

   - Coercing a parameter's data type (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until run time. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a value for the parameter consistent with its data type (required).
   - Providing a NULL value indicator for the parameter.

   The following code example illustrates these steps, looping through each XSQLVAR structure in the *in_sqlda* XSQLDA:

```
for (i=0, var = in_sqlda->sqlvar; i < in_sqlda->sqld; i++, var++)
{
   /* Process each XSQLVAR parameter structure here.
   The parameter structure is pointed to by var.*/
   dtype = (var->sqltype & ~1) /* drop NULL flag for now */
   switch(dtype)
   {
      case SQL_VARYING: /* coerce to SQL_TEXT */
         var->sqltype = SQL_TEXT;
         /* Allocate local variable storage. */
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         . . .
         break;
      case SQL_TEXT:
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         /* Provide a value for the parameter. */
         . . .
         break;
      case SQL_LONG:
```

```
            var->sqldata = (char *)malloc(sizeof(long));
            /* Provide a value for the parameter. */
            *(long *)(var->sqldata) = 17;
            break;
         . . .
      }  /* End of switch statement. */
      if (sqltype & 1)
      {
         /* Allocate variable to hold NULL status. */
         var->sqlind = (short *)malloc(sizeof(short));
      }
   }     /* End of for loop. */
```

For more information about data type coercion and NULL indicators, see "Coercing Data Types," in this chapter.

6. Execute the named statement string with EXECUTE. Reference the parameters in the input XSQLDA with the USING SQL DESCRIPTOR clause. For example, the following statement executes a statement string named SQL_STMT:

```
EXEC SQL
   EXECUTE SQL_STMT USING SQL DESCRIPTOR in_sqlda;
```

Re-executing the Statement String

Once a non-query statement string with parameters is prepared, it can be executed as often as required in an application. Before each subsequent execution, the input XSQLDA can be supplied with new parameter and NULL indicator data.

To supply new parameter and NULL indicator data for a prepared statement, repeat steps 3-5 of "Preparing and Executing a Statement String with Parameters," in this chapter.

Method 3: Query Statements Without Parameters

There are three steps to processing an SQL query statement string without parameters:

1. Preparing an output XSQLDA to process the select-list items returned when the query is executed.
2. Preparing the statement string.
3. Using a cursor to execute the statement and retrieve select-list items from the output XSQLDA.

Preparing the Output XSQLDA

Most queries return one or more rows of data, referred to as a *select-list*. Because the number and kind of items returned are unknown when a statement string is created, an output XSQLDA must be created to store select-list items that are returned at run time. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to store the column data for each row that will be fetched. For example, the following declaration creates an XSQLDA called *out_sqlda*:

   ```
   XSQLDA *out_sqlda;
   ```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *out_sqlda*:

   ```
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   Space for 10 XSQLVAR structures is allocated in this statement, enabling the XSQLDA to accommodate up to 10 select-list items.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

   ```
   out_sqlda->version = SQLDA_VERSION1;
   out_sqlda->sqln = 10;
   ```

Preparing a Query Statement String

After an XSQLDA is created for holding the items returned by a query statement string, the statement string can be created, prepared, and described. When a statement string is executed, InterBase creates the select-list of selected rows.

To prepare a query statement string, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string that performs a query:

```
char *str = "SELECT * FROM CUSTOMER";
```

The statement appears to have only one select-list item (*). The asterisk is a wildcard symbol that stands for all of the columns in the table, so the actual number of items returned equals the number of columns in the table.

2. Parse and name the statement string with PREPARE. The name is used in subsequent calls to statements such as DESCRIBE and EXECUTE:

```
EXEC SQL
   PREPARE SQL_STMT FROM :str;
```

SQL_STMT is the name assigned to the prepared statement string.

3. Use DESCRIBE OUTPUT to fill the output XSQLDA with information about the select-list items returned by the statement:

```
EXEC SQL
   DESCRIBE OUTPUT SQL_STMT INTO SQL DESCRIPTOR out_sqlda;
```

4. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the output descriptor can accommodate the number of select-list items specified in the statement. If not, free the storage previously allocated to the output descriptor, reallocate storage to reflect the number of select-list items specified by *sqld*, reset *sqln* and *version*, then execute DESCRIBE OUTPUT again:

```
if (out_sqlda->sqld > out_sqlda->sqln)
{
   n = out_sqlda->sqld;
   free(out_sqlda);
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
   out_sqlda->sqln = n;
   out_sqlda->version = SQLDA_VERSION1;
   EXEC SQL
      DESCRIBE OUTPUT SQL_STMT INTO SQL DESCRIPTOR out_sqlda;
}
```

5. Set up an XSQLVAR structure for each item returned. Setting up an item structure involves the following steps:

   - Coercing an item's data type (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until run time. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a NULL value indicator for the parameter.

The following code example illustrates these steps, looping through each XSQLVAR structure in the *out_sqlda* XSQLDA:

```
for (i=0, var = out_sqlda->sqlvar; i < out_sqlda->sqld; i++, var++)
{
   dtype = (var->sqltype & ~1) /* drop flag bit for now */
   switch (dtype)
   {
      case SQL_VARYING:
         var->sqltype = SQL_TEXT;
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen + 2);
         break;
      case SQL_TEXT:
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         break;
   case SQL_LONG:
         var->sqldata = (char *)malloc(sizeof(long));
         break;
         . . .
         /* process remaining types */
   }  /* end of switch statements */
   if (sqltype & 1)
   {
      /* allocate variable to hold NULL status */
      var->sqlind = (short *)malloc(sizeof(short));
   }
}     /* end of for loop */
```

For more information about data type coercion and NULL indicators, see "Coercing Data Types," in this chapter.

Executing a Statement String Within the Context of a Cursor

To retrieve select-list items from a prepared statement string, the string must be executed within the context of a cursor. All cursor declarations in InterBase are fixed, embedded statements inserted into the application before it is compiled. DSQL application developers must anticipate the need for cursors when writing the application and declare them ahead of time.

A looping construct is used to fetch a single row at a time from the cursor and to process each select-list item (column) in that row before the next row is fetched.

To execute a statement string within the context of a cursor and retrieve rows of select-list items, follow these steps:

1. Declare a cursor for the statement string. For example, the following statement declares a cursor, DYN_CURSOR, for the SQL statement string, SQL_STMT:

   ```
   EXEC SQL
   ```

```
DECLARE DYN_CURSOR CURSOR FOR SQL_STMT;
```

2. Open the cursor:

```
EXEC SQL
   OPEN DYN_CURSOR;
```

   Opening the cursor causes the statement string to be executed, and an active set of rows to be retrieved. For more information about cursors and active sets, see Chapter 6: “Working With Data.”

3. Fetch one row at a time and process the select-list items (columns) it contains. For example, the following loops retrieve one row at a time from DYN_CURSOR and process each item in the retrieved row with an application-specific function (here called **process_column()**):

```
while (SQLCODE == 0)
{
   EXEC SQL
      FETCH DYN_CURSOR USING SQL DESCRIPTOR out_sqlda;

   if (SQLCODE == 100)
      break;

   for (i = 0; i < out_sqlda->sqld; i++)
   {
      process_column(out_sqlda->sqlvar[i]);
   }
}
```

   The **process_column()** function mentioned in this example processes each returned select-list item. The following skeleton code illustrates how such a function can be set up:

```
void process_column(XSQLVAR *var)
{
   /* test for NULL value */
   if ((var->sqltype & 1) && (*(var->sqlind) = -1))
   {
      /* process the NULL value here */
   }
   else
   {
      /* process the data instead */
   }
. . .
}
```

4. When all the rows are fetched, close the cursor:

```
EXEC SQL
   CLOSE DYN_CURSOR;
```

Re-executing a Query Statement String Without Parameters

Once a query statement string without parameters is prepared, it can be executed as often as required in an application by closing and reopening its cursor.

To reopen a cursor and process select-list items, repeat steps 2-4 of "Executing a Statement String Within the Context of a Cursor," in this chapter.

Method 4: Query Statements With Parameters

There are four steps to processing an SQL query statement string with placeholder parameters:

1. Preparing an input XSQLDA to process a statement string's parameters.
2. Preparing an output XSQLDA to process the select-list items returned when the query is executed.
3. Preparing the statement string and its parameters.
4. Using a cursor to execute the statement using input parameter values from an input XSQLDA, and to retrieve select-list items from the output XSQLDA.

Preparing the Input XSQLDA

Placeholder parameters are replaced with actual data before a prepared SQL statement string is executed. Because those parameters are unknown when the statement string is created, an input XSQLDA must be created to supply parameter values at run time. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *in_slqda*:

```
in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
```

In this statement, space for 10 XSQLVAR structures is allocated, allowing the XSQLDA to accommodate up to 10 input parameters. Once structures are allocated, assign values to the *sqldata* field in each XSQLVAR.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

```
in_sqlda->version = SQLDA_VERSION1;
in_sqlda->sqln = 10;
```

Preparing the Output XSQLDA

Because the number and kind of items returned are unknown when a statement string is executed, an output XSQLDA must be created to store select-list items that are returned at run time. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *out_sqlda*:

```
XSQLDA *out_sqlda;
```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

```
XSQLVAR *var;
```

Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *out_sqlda*:

```
out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
```

Space for 10 XSQLVAR structures is allocated in this statement, enabling the XSQLDA to accommodate up to 10 select-list items.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

```
out_sqlda->version = SQLDA_VERSION1;
out_sqlda->sqln = 10;
```

Preparing a Query Statement String With Parameters

After an input and an output XSQLDA are created for holding a statement string's parameters, and the select-list items returned when the statement is executed, the statement string can be created and prepared. When a statement string is prepared, InterBase replaces the placeholder parameters in the string with information about the actual parameters used. The information about the parameters must be assigned to the input XSQLDA (and perhaps adjusted) before the statement can be executed. When the statement string is executed, InterBase stores select-list items in the output XSQLDA.

To prepare a query statement string with parameters, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string with placeholder parameters:

   ```
   char *str = "SELECT * FROM DEPARTMENT WHERE BUDGET = ?, LOCATION = ?";
   ```

   This statement string contains two parameters: a value to be assigned to the BUDGET field and a value to be assigned to the LOCATION field.

2. Prepare and name the statement string with PREPARE. The name is used in subsequent calls to DESCRIBE and EXECUTE:

   ```
   EXEC SQL
      PREPARE SQL_STMT FROM :str;
   ```

   SQL_STMT is the name assigned to the prepared statement string.

3. Use DESCRIBE INPUT to fill the input XSQLDA with information about the parameters contained in the SQL statement:

   ```
   EXEC SQL
      DESCRIBE INPUT SQL_STMT USING SQL DESCRIPTOR in_sqlda;
   ```

4. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the input descriptor can accommodate the number of parameters contained in the statement. If not, free the storage previously allocated to the input descriptor, reallocate storage to reflect the number of parameters specified by *sqld*, reset *sqln* and *version*, then execute DESCRIBE INPUT again:

   ```
   if (in_sqlda->sqld > in_sqlda->sqln)
   {
      n = in_sqlda->sqld;
      free(in_sqlda);
      in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
      in_sqlda->sqln = n;
      in_sqlda->version = SQLDA_VERSION1;
      EXEC SQL
   ```

```
        DESCRIBE INPUT SQL_STMT USING SQL DESCRIPTOR in_sqlda;
}
```

5. Process each XSQLVAR parameter structure in the input XSQLDA. Processing a parameter structure involves up to four steps:

   - Coercing a parameter's data type (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until run time. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a value for the parameter consistent with its data type (required).
   - Providing a NULL value indicator for the parameter.

   These steps must be followed in the order presented. The following code example illustrates these steps, looping through each XSQLVAR structure in the *in_sqlda* XSQLDA:

```
for (i=0, var = in_sqlda->sqlvar; i < in_sqlda->sqld; i++, var++)
{
   /* Process each XSQLVAR parameter structure here.
   The parameter structure is pointed to by var.*/
   dtype = (var->sqltype & ~1) /* drop flag bit for now */
   switch (dtype)
   {
      case SQL_VARYING: /* coerce to SQL_TEXT */
         var->sqltype = SQL_TEXT;
         /* allocate proper storage */
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         /* provide a value for the parameter. See case SQL_LONG */
         . . .
         break;
      case SQL_TEXT:
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         /* provide a value for the parameter. See case SQL_LONG */
         . . .
         break;
      case SQL_LONG:
         var->sqldata = (char *)malloc(sizeof(long));
         /* provide a value for the parameter */
         *(long *)(var->sqldata) = 17;
         break;
      . . .
   }  /* end of switch statement */
   if (sqltype & 1)
   {
      /* allocate variable to hold NULL status */
      var->sqlind = (short *)malloc(sizeof(short));
```

```
   }
}     /* end of for loop */
```

For more information about data type coercion and NULL indicators, see "Coercing Data Types," in this chapter.

6. Use DESCRIBE OUTPUT to fill the output XSQLDA with information about the select-list items returned by the statement:

```
EXEC SQL
   DESCRIBE OUTPUT SQL_STMT INTO SQL DESCRIPTOR out_sqlda;
```

7. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the output descriptor can accommodate the number of select-list items specified in the statement. If not, free the storage previously allocated to the output descriptor, reallocate storage to reflect the number of select-list items specified by *sqld*, reset *sqln* and *version*, and execute DESCRIBE OUTPUT again:

```
if (out_sqlda->sqld > out_sqlda->sqln)
{
   n = out_sqlda->sqld;
   free(out_sqlda);
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
   out_sqlda->sqln = n;
   out_sqlda->version = SQLDA_VERSION1;
   EXEC SQL
      DESCRIBE OUTPUT SQL_STMT INTO SQL DESCRIPTOR out_sqlda;
}
```

8. Set up an XSQLVAR structure for each item returned. Setting up an item structure involves the following steps:

   - Coercing an item's data type (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until run time. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a NULL value indicator for the parameter (optional).

   The following code example illustrates these steps, looping through each XSQLVAR structure in the *out_sqlda* XSQLDA:

```
for (i=0, var = out_sqlda->sqlvar; i < out_sqlda->sqld; i++, var++)
{
   dtype = (var->sqltype & ~1) /* drop flag bit for now */
   switch (dtype)
   {
```

```
      case SQL_VARYING:
         var->sqltype = SQL_TEXT;
      break;
      case SQL_TEXT:
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         break;
   case SQL_LONG:
         var->sqldata = (char *)malloc(sizeof(long));
         break;
         /* process remaining types */
   }  /* end of switch statements */
   if (sqltype & 1)
   {
      /* allocate variable to hold NULL status */
      var->sqlind = (short *)malloc(sizeof(short));
   }
}     /* end of for loop */
```

For more information about data type coercion and NULL indicators, see "Coercing Data Types," in this chapter.

Executing a Query Statement String Within the Context of a Cursor

To retrieve select-list items from a statement string, the string must be executed within the context of a cursor. All cursor declarations in InterBase are fixed, embedded statements inserted into the application before it is compiled. DSQL application developers must anticipate the need for cursors when writing the application and declare them ahead of time.

A looping construct is used to fetch a single row at a time from the cursor and to process each select-list item (column) in that row before the next row is fetched.

To execute a statement string within the context of a cursor and retrieve rows of select-list items, follow these steps:

1. Declare a cursor for the statement string. For example, the following statement declares a cursor, DYN_CURSOR, for the prepared SQL statement string, SQL_STMT:

   ```
   EXEC SQL
      DECLARE DYN_CURSOR CURSOR FOR SQL_STMT;
   ```

2. Open the cursor, specifying the input descriptor:

   ```
   EXEC SQL
      OPEN DYN_CURSOR USING SQL DESCRIPTOR in_sqlda;
   ```

   Opening the cursor causes the statement string to be executed, and an active set of rows to be retrieved. For more information about cursors and active sets, see Chapter 6: "Working With Data."

3. Fetch one row at a time and process the select-list items (columns) it contains. For example, the following loops retrieve one row at a time from DYN_CURSOR and process each item in the retrieved row with an application-specific function (here called **process_column()**):

```
while (SQLCODE == 0)
{
   EXEC SQL
      FETCH DYN_CURSOR USING SQL DESCRIPTOR out_sqlda;

   if (SQLCODE == 100)
      break;

   for (i = 0; i < out_sqlda->sqld; i++)
   {
      process_column(out_sqlda->sqlvar[i]);
   }
}
```

4. When all the rows are fetched, close the cursor:

```
EXEC SQL
   CLOSE DYN_CURSOR;
```

Re-executing a Query Statement String With Parameters

Once a query statement string with parameters is prepared, it can be used as often as required in an application. Before each subsequent use, the input XSQLDA can be supplied with new parameter and NULL indicator data. The cursor must be closed and reopened before processing can occur.

To provide new parameters to the input XSQLDA, follow steps 3-5 of “Preparing a Query Statement String with Parameters,” in this chapter.

To provide new information to the output XSQLDA, follow steps 6-8 of “Preparing a Query Statement String with Parameters,” in this chapter.

To reopen a cursor and process select-list items, repeat steps 2-4 of “Executing a Query Statement String Within the Context of a Cursor,” in this chapter.

CHAPTER 16

Preprocessing, Compiling, and Linking

This chapter describes how to preprocess a program by using **gpre**, and how to compile and link it for execution.

Preprocessing

After coding an SQL or dynamic SQL (DSQL) program, the program must be preprocessed with **gpre** before it can be compiled. **gpre** translates SQL and DSQL commands into statements the host-language compiler accepts by generating InterBase library function calls. **gpre** translates SQL and DSQL database variables into ones the host-language compiler accepts and declares these variables in host-language format. **gpre** also declares certain variables and data structures required by SQL, such as the SQLCODE variable and the extended SQL descriptor area (XSQLDA) used by DSQL.

Using gpre

The syntax for **gpre** is:

```
gpre [-language] [-options] infile [outfile]
```

The *infile* argument specifies the name of the input file.

The optional *outfile* argument specifies the name of the output file. If no file is specified, **gpre** sends its output to a file with the same name as the input file, with an extension depending on the language of the input file.

Switches include specification for **language** and **options**, described in the following sections. The switches can come either before or after the input and output file specification. Each switch must include at least a hyphen (preceded by a space) and a unique character specifying the switch.

Language Switches

The language switch specifies the language of the source program. C and C++ are languages available on all platforms. The switches are shown in the following table:

Table 16-1: **gpre** Language Switches

Switch	Language
-c	C
-cxx	C++

In addition, some platforms support other languages if an additional InterBase license for the language is purchased. The following table lists the available languages and the corresponding switches:

Table 16-2: Additional **gpre** Language Switches

Swsitch	Language
-al[**sys**]	Ada (Alsys)
-a[**da**]	Ada (VERDIX, VMS, Telesoft)
-ansi	ANSI-85 COBOL
-co[**bol**]	COBOL
-f[**ortran**]	FORTRAN
-pa[**scal**]	Pascal

For example, to preprocess a C program called *census.e*, type:

```
gpre -c census.e
```

Option Switches

The option switches specify preprocessing options. The following table describes the available switches:

Table 16-3: **gpre** Option Switches

Switch	Description
-charset *name*	Determines the active character set at compile time, where *name* is the character set name.

Table 16-3: **gpre** Option Switches (Continued)

Switch	Description
-d[**atabase**] *filename*	Declares a database for programs. *filename* is the file name of the database to access. Use this option if a program contains SQL statements and does not attach to the database itself. Do not use this option if the program includes a databasedeclaration.
-d_float	VAX/VMS only. Specifies that double-precision data will be passed from the application in D_FLOAT format and stored in the database in G_FLOAT format. Data comparisons within the database will be performed in G_FLOAT format. Data returned to the application from the database will be in D_FLOAT format.
-e[**ither_case**]	Enables **gpre** to recognize both uppercase and lowercase. Use the **-either_case** switch whenever SQL keywords appear in code in lowercase letters. If case is mixed, and this switch is not used, **gpre** cannot process the input file. This switch is not necessary with languages other than C, since they are case-insensitive.
-m[**anual**]	Suppresses the automatic generation of transactions. Use the **-m** switch for SQL programs that perform their own transaction handling, and for all DSQL programs that must, by definition, explicitly control their own transactions.
-n[**o_lines**]	Suppresses line numbers for C programs.
-o[**utput**]	Directs **gpre**'s output to standard output, rather than to a file.
-password *password*	Specifies *password*, the database password, if the program connects to a database that requires one.
-r[**aw**]	Prints BLR as raw numbers, rather than as their mnemonic equivalents. This option cam be useful for making the **gpre** output file smaller; however, it will be unreadable.
-sqlda [**old** | **new**]	Argument **old** specifies SQLDA, **new** specifies XSQLDA. If this switch is not used, the default is XSQLDA.
-user *username*	Specifies *username*, the database user name, if the program connects to a database that requires one.
-x *handle*	Gives the database handle identified with the **-database** option an external declaration. This option directs the program to pick up a global declaration from another linked module. Use only if the **-d** switch is also used.
-z	Print the version number of **gpre** and the version number of all declared databases. These databases can be declared either in the program or with the **-database** switch.

For sites with the appropriate license and are using a language other than C,

additional **gpre** options can be specified, as described in the following table:

Table 16-4: Language-specific **gpre** Option Switches

Switch	Description
-h[**andles**] *pkg*	Specifies, *pkg*, an Ada handles package.

Examples

The following command preprocesses a C program in a file named *appl1.e*. The output file will be *appl1.c*. Since no database is specified, the source code must connect to the the database.

```
gpre -c appl1
```

The following command is the same as the previous, except that it does not assume the source code opens a database, instead, explicitly declaring the database, *mydb.gdb*:

```
gpre -c appl1 -d mydb.gdb
```

Using a File Extension to Specify Language

In addition to using a language switch to specify the host language, it is also possible to indicate the host language with the file-name extension of the source file. The following table lists the file-name extensions for each language that **gpre** supports and the default extension of the output file:

Table 16-5: File Extensions for Language Specification

Language	Input File Extension	Default Output File Extension
Ada (VERDIX)	ea	a
Ada (Alsys, Telesoft)	eada	ada
C	e	c
C++	exx	cxx
COBOL	ecob	cob
FORTRAN	ef	f
Pascal	epas	pas

For example, to preprocess a COBOL program called *census.ecob*, type:

```
gpre census_report.ecob
```

This generates an output file called *census.cob*.

When specifying a file-name extension, it is possible to specify a language switch as well:

```
gpre -cob census.ecob
```

Specifying the Source File

Because both the language switch and the file-name extension are optional, **gpre** can encounter three different situations:

- A language switch and input file with no extension
- No language switch, but an input file with extension
- Neither a language switch, nor a file extension

Using a Language Switch and No Input File Extension

If **gpre** encounters a language switch, but the specified input file has no extension**,** it does the following:

1. It looks for the input file without an extension. If **gpre** finds the file, it processes it and generates an output file with the appropriate extension.

   If **gpre** does not find the input file, it looks for the file with the extension that corresponds to the indicated language. If it finds such a file, it generates an output file with the appropriate extension.

2. If **gpre** cannot find either the named file or the named file with the appropriate extension, it returns the following error:

   ```
   gpre: can't open filename or filename.extension
   ```

   filename is the file specified in the **gpre** command. *extension* is the language-specific file extension for the specified program.

For example, suppose the following command is processed:

```
gpre -c census
```

Then, the following occurs:

- **gpre** looks for a file called *census* without an extension. If found, it processes the file and generates *census.c*.
- If **gpre** cannot find this file, it looks for a file called *census.e*. If found, it processes the file and generates *census.c*.

- If **gpre** cannot find *census* or *census.e*, it returns this error:

```
gpre: can't open census_report or census.e
```

Using No Language Switch and an Input File With Extension

If a language switch is not specified, but the input file includes a file-name extension, **gpre** looks for the specified file and assumes the language is indicated by the extension.

For example, suppose the following command is processed:

```
gpre census.e
```

gpre looks for a file called *census.e*. If **gpre** finds this file, it processes it as a C program and generates an output file called *census.c*. If **gpre** does not find this file, it returns the following error:

```
gpre: can't open census.e
```

Using Neither a Language Switch Nor a File Extension

If neither a language extension nor a file-name extension is specified, **gpre** looks for a file in the following order:

1. *filename*.e (C)
2. *filename*.epas (Pascal)
3. *filename*.ef (FORTRAN)
4. *filename*.ecob (COBOL)
5. *filename*.ea (VERDIX Ada)
6. *filename*.eada (Alsys, and Telesoft Ada)

If **gpre** finds such a file, it generates an output file with the appropriate extension. If **gpre** does not find the file, it returns the following error:

```
gpre: can't find filename with any known extension. Giving up.
```

Compiling and Linking

After preprocessing a program, it must be compiled and linked. Compiling creates an object module from the preprocessed source file. Use a host-language compiler to compile the program.

The linking process resolves external references and creates an executable object. Use the tools available on a given platform to link a program's object module to other object modules and libraries, based on the platform, operating system and host language used.

Compiling an Ada Program

Before compiling an Ada program, be sure the Ada library contains the package *interbase.ada* (or *interbase.a* for VERDIX Ada). This package is in the InterBase *include* directory.

To use the programs in the InterBase *examples* directory, use the package *basic_io.ada* (or *basic_io.a* for VERDIX Ada), also located in the *examples* directory.

Linking

On Unix platforms, programs can be linked to the following libraries:

- A library that uses pipes, obtained with the **-lgds** option. This library yields faster links and smaller images. It also lets your application work with new versions of InterBase automatically when they are installed.
- A library that does not use pipes, obtained with the **-lgds_b** option. This library has faster execution, but binds an application to a specific version of InterBase. When installing a new version of InterBase, programs must be relinked to use the new features or databases created with that version.

Under SunOS-4, programs can be linked to a shareable library by using the **-lgdslib** option. This creates a dynamic link at run time and yields smaller images with the execution speed of the full library. This option also provides the ability to upgrade InterBase versions automatically.

For specific information about linking options for InterBase on a particular platform, consult the online *readme* in the *interbase* directory.

Index

Symbols

A

B

C

D

E

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X