Firebird

Embedded SQL Guide for RM/Cobol

Table of Contents

1. Program Structure

1.1. General

An embedded SQL application is an application in which the SQL statements needed to access data from the database manager are coded directly into the source code of the application. These SQL statements are compiled into code, executable by the database manager, at compile time rather than at run time as is the case for dynamic SQL (DSQL) applications. This results in a modest improvement in the application's performance. It also makes the application's source code easier to understand since the SQL statements are located in the code at the place where they will be executed rather than being buried in WORKING-STORAGE.

Embedded SQL applications include declarations and statements which tell the preprocessor, **gpre**, how the application is to interact with the Firebird database manager. The application must:

- Declare host variables to use for data transfer between the application and the database manager.
- Declare all databases to be accessed by the application.
- Open, all databases before they are accessed by the application.
- Set the options for and start all non default transactions used by the application.
- Include the SQL statements used to access the application's data.
- Close all transactions and databases before the application terminates.
- Provide error handling.

This chapter touches, very briefly, on each of these points. More detail is given in the following chapters.

1.2. Reading this Guide

There are certain conventions used throughout this guide when describing the syntax of SQL statements.

- Words given in upper case are reserved words and must be typed as shown.
- Words given in lower case are user supplied.
- Items enclosed in braces, {}, represent a list of choices, one of which must be chosen.
- Items enclosed in square brackets, [], represent optional items.
- Punctuation, like semi-colons and quotes, must be typed as shown.

1.3. Definition of Terms

There are certain terms used throughout this guide that may be new to some users. Below is a short explanation of some of these terms.

Tables are similar to COBOL files.

Rows are similar to COBOL records.

Columns are similar to COBOL fields.

Cursors are pointers into the rows of the result set of a SQL search.

Result sets contain rows returned by a query.

A **view** is a subset of one or more tables containing some or all of the columns and some or all of the rows from the table(s).

The keyword **NULL** is used to refer to columns that do not have a value. This is sometimes a difficult concept for Cobol programmers to grasp since “no value” is not equivalent to spaces or zeros. A column value has a value of spaces or zeros is **NOT NULL**. A column the is **NULL** will never be considered as equal to a column with a value, whatever that value might happen to be. In fact, a **NULL** column is not even considered to be equal to another **NULL** column. Any expression which includes **NULL** evaluates to unknown rather than a specific value.

1.4. Declaring Host Variables

Host variables are standard Cobol data items that are used to transfer data between the application and the database manager. Host variables are used in the following situations:

- During data retrieval the values of database fields are moved into host variables.
- During insert or update operations the values of host variables are moved into database fields.
- Host variables can be used to hold the values used in search conditions.

Some embedded SQL preprocessors require that all host variables be declared between BEGIN DECLARE SECTION and END DECLARE SECTION declarations. For portability reasons Firebird supports these declarations but it is not a requirement of the Firebird preprocessor that host variables be declared between them. Host variables may be declared anywhere a standard Cobol data item may be declared, either in the WORKING-STORAGE or LINKAGE sections.

In addition to being able to declare host variables in the usual, Cobol way it is also possible to declare a host variable BASED ON the definition of a database column. Using BASED ON ensures that the host variable is of the proper size and type to hold the data from the database column. For example, the following statements both declare a shipper address field:

```
01  WS-SHIP-ADDR                PIC X(35).
01  WS-SHIP-ADDR                BASED ON LOADS.LD_SHIP_ADDR1.
```

Assuming that the column LD_SHIP_ADDR1 from the table LOADS is defined as a 35 character text field these two lines of code are equivalent.

1.5. Declaring and Accessing Databases

Embedded SQL applications can access multiple Firebird databases simultaneously. Each database must be declared and opened before it can be accessed by the application. Programs that access only a single database need not declare the database or assign a database handle. Instead the database can be specified on the **gpre** command line.

The embedded SQL application must declare and open the database before attempting to start a transaction or access any of the database tables. Opening a database performs the following actions:

- Connects to the database manager
- Open the database files
- Allocates the system resources required to manage the database connection

To declare a database the application uses the SET DATABASE statement. To open the database it uses the CONNECT statement. For example, the following code snippet declares a database and opens it:

```
EXEC SQL
    DECLARE DATABASE DB1 = 'sample.fdb'; .

    ...

EXEC SQL
    CONNECT DB1; .
```

Before terminating, an embedded SQL application must close the database. To close a database use the DISCONNECT statement or the RELEASE clause of the COMMIT and ROLLBACK statements.

1.6. Transactions

An understanding of transaction management is central to any SQL application. All data access must be carried out within the context of a transaction. The transaction is responsible for making certain that all data changes within its scope either succeed or fail as a unit. Efficient use of transactions will result in a fast, responsive system. Inefficient use of transactions can lead to a slow, sluggish system or, even worse, a system plagued by hangs and deadlocks.

Even though **gpre** provides automatic transaction management I wouldn't recommend using it. Explicit transaction management always gives better control and efficiency.

To start a transaction the application uses the SET TRANSACTION statement. This statement specifies the type of transaction to be started, READ ONLY or READ WRITE, the isolation level, READ COMMITTED or SNAPSHOT, the lock resolution mode, WAIT or NO WAIT, and other options. It also, optionally, names the transaction. The transaction name becomes important if you want to use multiple transactions within a single application.

To save the changes made during a transaction, the application uses the COMMIT statement. To discard changes it uses the ROLLBACK statement.

1.7. SQL Statements

SQL statements are always preceded by EXEC SQL and are terminated with a semi-colon. It is also possible to add an optional period following the semi-colon if the Cobol syntax requires it. For example:

```
EXEC SQL
    SELECT LD_SHIP_ADDR1
        INTO :WS-SHIP-ADDR1
        FROM LOADS
        WHERE LD_LOAD_NUMBER = 123456; .
```

1.8. Statement Handles

All SQL statements that retrieve data from or update data in the database cause a request handle to be created in WORKING-STORAGE. For the most part, these handles are invisible to and need not concern the Cobol programmer. The only exception to this occurs in Cobol subprograms that are CALLed and subsequently CANCELed by a main program. The CANCEL verb deallocates the WORKING-STORAGE used by the subprogram which, in turn, deallocates the statement handles created by the subprogram. This can cause problems such as constantly increasing memory size and aborts when the main program tries to detach from the database.

It is necessary to use the RELEASE_REQUESTS statement before terminating any subprograms using the EXIT PROGRAM verb. See the Miscellaneous Commands chapter for details on this statement.

1.9. Dynamic SQL (DSQL)

Dynamic SQL (DSQL) applications must be able to manipulate an XSQLDA structure. Since this structure contains pointers to the data fields and since RM/Cobol is not able to dereference these pointers to obtain the value of the field that they point to, it is not possible to use DSQL in an RM/Cobol application. Therefore, the DSQL capabilities of the preprocessor are not discussed in this manual.

1.10. Error Handling

Every executable SQL statement returns a value in SQLCODE which indicates the success or failure of that statement. The application must check this value after every statement in order to determine whether or not the statement succeeded. This can be done by either manually checking SQLCODE after every statement or by using a WHENEVER SQLERROR statement. WHENEVER SQLERROR provides a means of specifying a default action that should take place whenever an SQL error occurs. The default action will be taken for every SQL statement appearing in the application after the WHENEVER SQLERROR statement. For example:

```
EXEC SQL
    SELECT LD_SHIP_ADDR1
        INTO :WS_SHIP_ADDR
        FROM LOADS
        WHERE LD_LOAD_NUMBER = 123456; .
    IF SQLCODE < ZERO THEN
        GO TO ISC-ERROR.
    MOVE "NEW ADDRESS" TO WS-SHIP-ADDR.
EXEC SQL
    UPDATE LOADS
        SET LD_SHIP_ADDR1 = :WS-SHIP-ADDR
        WHERE LD_LOAD_NUMBER = 123456; .
    IF SQLCODE < ZERO THEN
        GO TO ISC-ERROR.
```

is equivalent to:

```
EXEC SQL
    WHENEVER SQLERROR GO TO ISC-ERROR; .
EXEC SQL
    SELECT LD_SHIP_ADDR1
        INTO :WS_SHIP_ADDR
```

```
           FROM LOADS
           WHERE LD_LOAD_NUMBER = 123456; .
       MOVE "NEW ADDRESS" TO WS-SHIP-ADDR.
   EXEC SQL
       UPDATE LOADS
           SET LD_SHIP_ADDR1 = :WS-SHIP-ADDR
           WHERE LD_LOAD_NUMBER = 123456; .
```

2. Using Databases

This chapter describes how embedded SQL applications use SQL statements to declare and access databases. There are three SQL statements which set up a database for access:

- SET DATABASE declares a database handle and optionally configures certain operational parameters for the database.
- SET NAMES optionally specifies the character set in use by the application for CHAR, VARCHAR and TEXT Blobs. The server uses this information to translate from the database's default character set to that used by the application.
- CONNECT opens a database and optionally assigns values to certain operational parameters for the database.

All databases must be closed before the application terminates.

2.1. Declaring a Database

Before a database can be used in an application it must be declared using the SET DATABASE statement to establish a database handle. A database handle is a 32 bit integer value that provides a unique identifier for the database connection. Database handles are used in subsequent SQL statements to identify the database that they should act upon. Database handles can also be used in applications that access multiple databases simultaneously to differentiate tables when two or more databases have identically named tables.

Each database handle must have a name that is unique among all other variables in the application and cannot be a Cobol reserved word.

It is possible for an embedded SQL application to access data from more than one database concurrently. To do this you must declare each database separately using the SET DATABASE statement. Each declaration must use a unique name for the database handle.

2.1.1. Connection Strings

Every database is referenced by using a connection string (path). A connection string consists of two parts; an optional server name and a platform specific path to the database file itself. There are several different formats for connection strings depending on the protocol to be used to communicate with the database manager but since we will be using only the TCP/IP protocol I will only describe that format in this guide.

The connection string has the following format:

```
server:path
```

Connection String for a Windows Server

```
w2k:c:\data\tl.fdb
```

Connection String for a Unix Server

```
unix:/data/tl.fdb
```

Notice that the database file path conforms to the standard in use by the server where the file is located. Windows servers use drive specifiers and backslashes and Unix servers don't use a drive specifier and use forward slashes. Firebird databases cannot be accessed by way of a shared drive or directory. The drive specifier for Windows systems must always point to a locally connected hard drive. Likewise, the full path for Unix systems must specify directories located on a locally connected hard drive.

2.1.2. SET DATABASE

```
SET {DATABASE | SCHEMA}
    dbhandle = [GLOBAL | STATIC | EXTERN]
               [COMPILETIME] [FILENAME] 'dbname'
               [USER 'name' PASSWORD 'pwd']
               [RUNTIME [FILENAME] {'dbname' | :var}
               [USER {'name' | :var} PASSWORD {'pwd' | :var}]];
```

SET DATABASE declares a host variable named **dbhandle** to hold the database handle associated with **dbname**. Applications which access multiple databases simultaneously must use SET DATABASE to create separate database handles for each database.

dbname is the platform specific path to the database. It must follow the file naming conventions for the server where the database resides.

The **GLOBAL** parameter declares the handle to be global to the run unit. All subroutines executed as part of the current run unit can access this handle. The **STATIC** parameter restricts accessibility of the handle to the current source module. The **EXTERN** parameter causes the handle to reference a global handle from another source module. Global is the default.

The optional **COMPILETIME** and **RUNTIME** parameters allow a single database handle to refer to one database at compile time and a different database at run time. If these parameters are omitted or if only **COMPILETIME** is given the handle will reference the same database at compile time and run time. The run time database can be overridden by the CONNECT statement.

The **USER** and **PASSWORD** parameters supply the user id and password used to connect to the database. The compile time user id and password can be overridden on the **gpre** command line. The run time user id and password can be overridden by the CONNECT statement or by supplying their values in the host variables denoted by **var**.

This statement causes a number of host variables to be declared at the point in the program where it is encountered. Hence, this statement must be placed in WORKING-STORAGE.

Example

```
WORKING-STORAGE SECTION.
EXEC SQL
    SET DATABASE TL = COMPILETIME 'w2k:c:\databases\tl.fdb'
        USER 'sysdba' PASSWORD 'masterkey';
```

2.1.3. SET NAMES

```
SET NAMES [charset | :var];
```

SET NAMES specifies the character set to use for subsequent database attachments. This statement must appear before the SET DATABASE and CONNECT statements that it is to affect.

charset is the name of the character set used by the application. Data will translated between this character set and the default character set for the database as data is transferred between the application and the database manager. The default character set is NONE.

Using character set NONE means that no translation will take place. The application will receive the data just as it was entered into the database.

2.2. Opening a Database

Before data can be retrieved from or stored into a database it must first be opened. This is also known as connecting to the database. Opening, or connecting to, a database is accomplished using the CONNECT statement.

The CONNECT statement:

- Allocates system resource for the database.
- Opens the database and examines it to be sure that it is valid.
- Checks the user name and password against the security database to ascertain whether or not the user is allowed to access the database.
- Optionally provides an SQL role that the user adopts on connection to the database, provided that the user has previously been granted membership in the role. A role is a security group that controls access to various parts of the database.
- Sets the size of the database cache buffer to allocate to the application when the default cache in inappropriate.

2.2.1. CONNECT

```
CONNECT [TO] {{ALL | DEFAULT} [<config_opts>] | <db_specs>};

<db_specs> = {{'connection_string' | :var} AS dbhandle | dbhandle}
             [<config_opts>] [, <dp_specs>]
```

```
<config_opts> = USER {'user' | :var} |
                PASSWORD {'password' | :var} |
                CACHE integer [BUFFERS] [<config_opts>]
```

If the all seems a little daunting, well it is. This can be restated in a form more familiar to Cobol programmers as:

Format 1

```
CONNECT [TO] {ALL | DEFAULT}
    [USER {'user' | :var}]
    [PASSWORD {'password' | :var}]
    [CACHE integer [BUFFERS]];
```

Format 2

```
CONNECT [TO] {{'connection_string' | :var} AS dbhandle | dbhandle}, ...
    [USER {'user' | :var}]
    [PASSWORD {'password' | :var}]
    [CACHE integer [BUFFERS]];
```

Use of **ALL** or **DEFAULT**, as in Format 1, opens all databases specified by a SET DATABASE statement. Options specified with **ALL** or **DEFAULT** apply to all databases. Use of Format 2 opens only the database(s) given in the CONNECT statement.

'connection_string' is the platform specific path to the database. If this parameter is omitted the connection string given in the SET DATABASE statement which declared **dbhandle** is used.

dbhabdle is a database handle declared in a previous SET DATABASE statement.

The **USER** and **PASSWORD** parameters provide the user name and password to be used to connect to the database manager. If these parameters are omitted, the ones given in the SET DATABASE statement will be used.

The **CACHE** parameter sets the number of database pages that can be held in memory (cached). Adjusting this number upward will generally improve performance while adjusting it downward will generally degrade performance. The minimum value for this parameter is 45 while the default and maximum values are dependent on the database manager's configuration and the availability of system resources.

Examples

Connect to all databases declared with SET DATABASE statements, using the user name and password from the SET DATABASE statement:

```
EXEC SQL
    SET DATABASE TL = 'w2k:c:\data\tl.fdb' USER 'sysdba' PASSWORD 'masterkey';
EXEC SQL
    SET DATABASE IMG = 'w2k:c:\data\img.fdb' USER 'sysdba' PASSWORD 'masterkey';
EXEC SQL
    CONNECT TO ALL;
```

Connect to a single database using the connection string given in the SET DATABASE statement for handle TL but providing a hard coded user name and password:

```
EXEC SQL
    SET DATABASE TL = 'w2k:c:\data\tl.fdb' USER 'sysdba' PASSWORD 'masterkey';
EXEC SQL
    CONNECT TL USER 'sysdba' PASSWORD 'masterkey';
```

Connect to a single database using a connection string, user name and password held in host variables:

```
EXEC SQL
  SET DATABASE TL = COMPILETIME 'w2k:c:\data\tl.fdb'
    USER 'sysdba' PASSWORD 'masterkey';
EXEC SQL
    CONNECT :WS-DB-PATH AS TL USER :WS-USER-ID PASSWORD :WS-PASSWORD;
```

2.3. Closing a Database

When the application is finished with a database, the database should be closed. A database can be closed using the DISCONNECT statement or the RELEASE clause of the COMMIT and ROLLBACK statements. Before closing a database you must COMMIT or ROLLBACK all transactions affecting the databases(s) to be closed. Closing a database does the following:

- Closes open database files.
- Releases system resources used by the database connection.
- Disconnects from the database manager.

2.3.1. DISCONNECT

```
DISCONNECT {{ALL | DEFAULT} | dbhandle [[, dbhandle] ...]};
```

Use of **ALL** or **DEFAULT** closes all open databases.

dbhandle is the handle of a previously opened database. You can specify multiple handles to close multiple databases.

Examples

Close all open databases.

```
EXEC SQL
    DISCONNECT ALL;
```

Close specific databases.

```
EXEC SQL
    DISCONNECT TL, IMG;
```

3. Using Transactions

All SQL data definition and data manipulation statements take place within the context of a transaction. A transaction defines a unit of work that either succeeds or fails as a unit. Transactions are managed through the use of the SET TRANSACTION, COMMIT and ROLLBACK statements.

Transaction management statements define the beginning and end of a transaction. They also control its behaviour and interaction with other simultaneously running transactions that share access to the same data. There are two type of transactions in Firebird; the default transaction (ISC-TRANS) and named transactions.

The default transaction is used when a statement which requires a transaction is encountered without a preceding SET TRANSACTION statement. Default behaviour is defined for the default transaction but it can be overridden by starting the default transaction using SET TRANSACTION and specifying alternative behaviour as parameters. In the absence of the **gpre -manual** command line parameter a default transaction will be started automatically when needed and committed when the database is closed.

Named transactions are always started using SET TRANSACTION statements. These statements provide unique names for each transaction and usually include parameters that specify the transaction's behaviour.

3.1. Transaction Behaviour

Transactions control access to data by simultaneously executing applications. There are several parameters which define exactly how the transaction behaves when multiple applications attempt to access the same data at the same time. These behaviours fall into three groups; access mode, isolation level and lock resolution. Access mode determines whether the transaction can read or read and write data. Isolation level determines how the transactions views records committed by other simultaneous transactions. Lock resolution determines whether or not the transaction waits for resources locked by other simultaneous transactions.

If a transaction is started without specifying any behaviour parameters, it is started with the following default behaviour:

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

3.1.1. Access Mode

Access mode is controlled by the **READ ONLY** and **READ WRITE** parameters in the SET TRANSACTION statement. **READ ONLY** allows the transaction to select data from a table but it cannot insert, update or delete table data.

If your application is only going to read data it is good practice the start your transaction using the **READ ONLY** property. This will ensure that your application has the least possible impact on other, concurrently executing transactions.

3.1.2. Lock Resolution

A transaction can choose to wait for resources locked by another transaction to be released or it can choose to fail immediately if a lock is detected. This behaviour is controlled by the lock resolution parameters **WAIT** and **NO WAIT**.

If you choose **WAIT,** the transaction will wait until locked resources are released and will then try to reacquire the lock. If you choose **NO WAIT,** the transaction will fail with a lock conflict error immediately when a lock is detected. If a **NO WAIT** transaction fails, it is your responsibility to rollback the transaction and try again or give up and report the error.

3.1.3. Isolation Level

Isolation level determines how your application views changes made by other, concurrently executing transactions. There are three main isolation levels; **SNAPSHOT**, **SNAPSHOT TABLE STABILITY** and **READ COMMITTED**.

SNAPSHOT isolation provides a view of the database at the moment the transaction starts. Your transaction will be prevented from seeing changes made by other transactions. This isolation level is also known as repeatable read because you can read the same record multiple times and be assured that you will always see the same version of the record.

SNAPSHOT TABLE STABILITY isolation provides the same view of the database as **SNAPSHOT** while, at the same time, preventing other transactions from making changes to tables that your transactions is reading and updating. Other transactions are allowed to read data from tables that your transaction is reading and updating.

READ COMMITTED isolation allows your transaction to read the most recent version of a row committed by other active transactions. There are two further options controlling the behaviour of **READ COMMITTED** transactions; **RECORD_VERSION** and **NO RECORD_VERSION**. When **RECORD_VERSION** is specified your transaction reads the most recently committed version of a row even if there is a newer, uncommitted version present in the database. When

NO RECORD_VERSION is specified and a newer, uncommitted version of a row is present your transaction will wait until that version is committed if the **WAIT** parameter is specified. It will immediately report a deadlock error if a newer, uncommitted version is present and the **NO WAIT** parameter is specified.

3.2. Starting a Transaction

Before any data can be accessed from an open database a transaction must be started. In the absence of the **gpre -manual** command line argument, transactions will be started automatically whenever an SQL statement which requires a transaction is executed. If the **gpre -manual** command line argument was specified or if you want to explicitly start a transaction you use the SET TRANSACTION statement.

3.2.1. SET TRANSACTION

```
SET TRANSACTION [NAME transaction_name]
    [READ WRITE | READ ONLY]
    [WAIT | NO WAIT]
    [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY] |
    READ COMMIT [[NO] RECORD_VERSION]}]
    [RESERVING <reserving_clause>]
    [USING dbhandle [, dbhandle] ...];

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

The **NAME** clause specifies a name for the transaction. **transaction_name** is a host variable that must be defined as PIC S9(10) USAGE BINARY(4). **transaction_name** must be initialized to zero before it is first referenced in a SET TRANSACTION statement. **transaction_name** can be used in subsequent SQL statements to force them to use the named transaction rather than the default.

The **READ WRITE** clause indicates that the transaction can read and write tables. This is the default.

The **READ ONLY** clause indicates that the transaction can only read tables.

The **WAIT** clause indicates that the transaction will wait for access if it encounters a lock conflict with another transaction. This is the default.

The **NO WAIT** clause indicates that the transaction will return an error immediately if it encounters a lock conflict with another transaction.

The **ISOLATION LEVEL** clause specifies the behaviour of this transaction when attempting to access the same tables as other simultaneous transactions. The words **ISOLATION LEVEL** themselves are optional. **SNAPSHOT** is the default setting. See the preceding section for a more complete discussion of isolation levels.

The **RESERVING** clause enables a transaction to register its desired level of access for specified tables when the transaction starts instead of when the transaction attempts its operations on that table.

- **PROTECTED WRITE** allows the current transaction to write to the table and blocks all other writes.
- **PROTECTED READ** disallows writing to the table by any transaction, including the current one.
- **SHARED WRITE** allows any **SNAPSHOT** read-write or **READ COMMITTED** read-write transaction to update rows in the table as long as no transaction requests exclusive use.
- **SHARED READ** allows any read-write transaction to update the table.

The **USING** clause limits the number of databases that the transaction can access.

Examples

Start a read, write transaction that can read only the most recently committed version of rows and will wait for lock to be released before continuing.

```
EXEC SQL
    SET TRANSACTION READ WRITE WAIT READ COMMITTED NO RECORD_VERSION;
```

Start a transaction which will be the only transaction allowed to update the LOADS table.

```
EXEC SQL
    SET TRANSACTION READ WRITE WAIT SNAPSHOT RESERVING LOADS
        FOR PROTECTED WRITE;
```

3.3. Saving Your Changes

When a transaction's tasks are complete the transaction must be ended to set the database to a consistent state and to make any changes made by the transaction permanent. Successful transactions are ended by the COMMIT statement.

The COMMIT statement:

- Writes all updates to the database.
- Makes the transaction's updates visible to other transactions.
- Closes open cursors, unless the **RETAIN** clause is given.

3.3.1. COMMIT

```
COMMIT [WORK] [TRANSACTION transaction_name] [RELEASE] [RETAIN [SNAPSHOT]];
```

WORK is an optional word used for compatibility with other relational databases which require it.

The **TRANSACTION** clause causes the transaction named **transaction_name** to be committed to the database. If this clause is omitted the default transaction is committed.

The **RELEASE** clause disconnects the application from the database after the commit is complete. This clause is only available for compatibility with previous versions on Interbase. To close a database use the DISCONNECT statement.

The **RETAIN [SNAPSHOT]** clause commits the changes and retains the current transaction context. This has the effect of leaving currently open cursors open rather than closing them. Be aware the use of the **RETAIN** clause leaves the transaction in an active state. Leaving a **READ WRITE** transaction open for long periods of time will have a negative impact on system performance. Every transaction must eventually be closed by a COMMIT or ROLLBACK statement without a RETAIN clause.

Examples

Commit a transaction and close all open cursors.

```
EXEC SQL
    COMMIT;
```

Commit a transaction leaving all cursors open.

```
EXEC SQL
    COMMIT RETAIN;
```

3.4. Discarding Your Changes

When an error is encountered during transaction processing and you want to discard your changes and return the database to the state it had prior to the start of the transaction you use the ROLLBACK statement.

The ROLLBACK statement:

- Discards all changes made to the database since the transaction was started.
- Closes all open cursors.

3.4.1. ROLLBACK

```
ROLLBACK [WORK] [TRANSACTION transaction_name] [RELEASE] [RETAIN [SNAPSHOT]];
```

WORK is an optional word used for compatibility with other relational databases which require it.

The **TRANSACTION** clause causes the transaction named **transaction_name** to be rolled back. If this clause is omitted the default transaction is rolled back.

The **RELEASE** clause disconnects the application from the database after the rollback is complete. This clause is only available for compatibility with previous versions on Interbase. To close a database use the DISCONNECT statement.

The **RETAIN [SNAPSHOT]** clause rolls back the changes and retains the current transaction context. This has the effect of leaving currently open cursors open rather than closing them. Be aware the use of the **RETAIN** clause leaves the transaction in an active state. Leaving a **READ WRITE** transaction open for long periods of time will have a negative impact on system performance. Every transaction must eventually be closed by a COMMIT or ROLLBACK statement without a RETAIN clause.

Examples

Rollback the default transaction.

```
EXEC SQL
    ROLLBACK;
```

Rollback the transaction named TRANS2.

```
EXEC SQL
    ROLLBACK TRANSACTION TRANS2;
```

4. Accessing the Data

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

- SELECT statements read or retrieve existing data from a database.
- INSERT statements write new rows of data to a table.
- UPDATE statements modify existing rows of data in a table.
- DELETE statements delete existing rows of data from a table.

This chapter will describe the basics of using these statements to retrieve and update data. A complete discussion of SQL is outside the scope of this manual. For more information see the Interbase Language Reference by Borland or The Firebird Book by Helen Borrie.

4.1. Supported Data Types

Firebird supports thirteen fundamental data types which are described in the following table.

Data Type	*Size*	*Range/Precision*	*Description*
BIGINT	64 bits	· 2^{-63} to 2^{63} - 1	· Signed 64 bit integer. Only available in dialect 3 databases.
BLOB	Variable	· None · Blob segment size is limited to 32K	· Dynamically sizable data type for holding graphics, sound, etc.
CHAR(n)	n characters	· 1 to 32,767 bytes · Character set character size determines the number of characters that can fit in 32K	· Fixed length text string
DATE	64 bits	· 1 Jan 100 BC to 29 Feb 32768 CE	· In dialect 3 databases this type holds a date only. In dialect 1 databases this type is equivalent to TMESTAMP.
DECIMAL(precision, scale)	16, 32 or 64 bits	· precision = 1 to 18 digits. Column must be able to store **at least** this many digits. · scale = 1 to 18 digits. Specifies the number of decimal places. Must be less than or equal to precision.	· Numeric data type with **scale** digits to the right of the decimal point. · Example: DECIMAL(10,3) holds numbers accurately in the following format: ppppppp.sss
DOUBLE PRECISION	64 bits	· $2.225x10^{-308}$ to $1.797x10^{308}$	· IEEE double precision floating point number with maximum 15 digits precision. Use when range of number to be stored is not know in advance.
FLOAT	32 bits	· $1.175x10^{-38}$ to $3.492x10^{38}$	· IEEE single precision floating point number with maximum 7 digits precision. Use when range of number to be stored is not know in advance.
INTEGER	32 bits	· -2,147,483,648 to 2,147,483,647	· Signed 32 bit integer.
NUMERIC(precision, scale)	16, 32 or 64 bits	· precision = 1 to 18 digits. Column must be able to store **exactly** this many digits. · scale = 1 to 18 digits. Specifies the number of decimal places. Must be less than or equal to precision.	· Numeric data type with **scale** digits to the right of the decimal point. · Example: DECIMAL(10,3) holds numbers accurately in the following format: ppppppp.sss
SMALLINT	16 bits	· -32,768 to 32,767	· Signed 16 bit integer
TIME	64 bits	· 00:00 AM to 23:59:59.9999 PM	· Time of day. Not available in dialect 1 databases.
TIMESTAMP	64 bits	· 1 Jan 100 00:00:00 BC to 29 Feb 32768 23:59:59.9999 CE	· Date and time of day. Not available in dialect 1 databases, use DATE instead.
VARCHAR(n)	n characters	· 1 to 32,767 bytes · Character set character size determines the number of characters that can fit in 32K	· Variable length text string

Table 1 Firebird Data Types

When reading data from the database or writing data to the database, **gpre** always moves your application's data through

temporary variables that are created by **gpre** and are guaranteed to be the proper size and type to hold the data. These temporary variables are then moved to and from your host variables by **gpre** using the Cobol MOVE verb. This means that, in general, you can define your variables to be whatever size and format you desire as long as host variables for numeric fields can be expected to correctly receive values from and send values to numeric data items according to the rules of the Cobol MOVE verb.

The exception to this rule is date fields. Date fields will be moved into your host variables in two different ways according to the setting of the **gpre -dfm** command line argument. If the **-dfm** argument is omitted your host variable will receive a raw, 64 bit date field in Firebird internal format. Your application will then have to use the **isc_encode_date** and **isc_decode_date** functions to translate the date to and from Cobol format. If the **-dfm** argument is specified then the date will be moved to and from your host variable according to the format string given to the **-dfm** argument. The format string controls the formatting of dates according to the presence of format characters in the string. **yy** or **yyyy** indicates the position of the year, **mm** the month, **dd** the day, **hh** the hour, **nn** the minutes and **ss** the seconds. For example, **-dfm yyyymmddhhnnss** would cause Firebird to format a **DATE** column as a four digit year, two digit month, two digit day of the month, two digit hour, two digit minutes and two digit seconds.

4.2. SQL Expressions

All SQL data manipulation statements support SQL expressions. Expressions are used for comparing and evaluating columns, constants and host variables to produce a single value.

For example, the WHERE clause is used to specify a search condition that determines if a row qualifies for retrieval or update. The search condition is an SQL expression. When an expression is used as a search condition it evaluates to a value of True or False.

Expressions can also be used to calculate values that will be inserted or updated into a column. When inserting or updating a numeric value the expression is usually arithmetic, such as multiplying one number by another. When inserting or updating a string value the expression can concatenate two strings to produce a single, new string.

Element	*Description*
Column names	Columns from specified tables.
Host variables	Program variables containing changeable values. Host variables must be preceded by a colon (:).
Constants	Hard coded numbers or strings. i.e. 12345 or 'string value'.
Concatenation operator	||, used to concatenate two strings.
Arithmetic operators	+, -, * and /
Logical operators	The keywords NOT, AND and OR.
Comparison operators	<, >, <=, >=, = and <>. Other, more specialized comparison operators include ALL, ANY, BETWEEN, CONTAINING, EXISTS, FIRST, IN, IS, LIKE, NULL, SINGULAR, SOME, and STARTING WITH.
COLLATE clause	Comparison of text strings can sometimes take advantage of the COLLATE clause to force the way the strings 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.
Subqueries	SELECT statements that return values that are to be compared with the result set of the main SELECT statement, or return values that are to be inserted into columns in an INSERT statement.
Parenthesis	Used to group expressions into hierarchies.
Date literals	String values that can be entered in quotes and interpreted as date values. Possible values are 'NOW', 'TODAY', 'YESTERDAY' and 'TOMORROW'.
Context variables	A number of system maintained variable values in the context of the current client connection. There is a whole collection of these, of which the following are applicable to to embedded SQL applications: CURRENT_DATE, CURRENT_ROLE, CURRENT_SESSION, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_TRANSACTION, CURRENT_USER and USER.

Table 2Elements of SQL Expressions

4.2.1. Precedence of Operators

Precedence determines the order in which operators are evaluated in an expression. Operators are grouped into four main types which are evaluated using the following precedence; concatenation, arithmetic, comparison and logical. 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. When there is a conflict, operator precedence within the type determines the order of evaluation.

4.2.2. String Concatenation

The 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.

Example

```
01  WS-TEXT                    PIC X(80).

...

EXEC SQL
    SELECT LAST_NAME || ', ' || FIRST_NAME
        INTO :WS-TEXT
        FROM ACCOUNTS;
```

4.2.3. Arithmetic Operators

Firebird supports the following arithmetic operators:

Multiplication	*
Division	/
Addition	+
Subtraction	-

Arithmetic expressions are evaluated left to right, except when ambiguities arise. In these cases, Firebird evaluates multiplication and division first followed by addition and subtraction.

Example

```
1 + 4 * 8
```

This yields the result 33.

4.2.4. Comparison Operators

Comparison operators test 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. Every test produces a result that can be either True or False.

Firebird supports the following comparison operators:

Equality	=
Inequality	<>
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

Firebird also supports some more specialized comparison operators. These are:

BETWEEN
CONTAINING
IN
LIKE
IS NULL
STARTING WITH
ALL
ANY or SOME
EXISTS
SINGULAR

4.2.4.1. BETWEEN

BETWEEN tests whether a value falls within a range of values. The range of values is inclusive, that is, the range includes both the low value and the high value. The complete syntax for the **BETWEEN** operator is:

```
<value1> [NOT] BETWEEN <value2> AND <value3>
```

Example

The following cursor declaration retrieves all load numbers where the shipper's zip code falls on or between 19400 and 19499.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_SHIP_ZIP BETWEEN '19400' AND '19499';
```

4.2.4.2. CONTAINING

CONTAINING tests to see if a string value contains a string literal. String comparisons are case insensitive. “String”, “STRING” and “string” are equivalent for **CONTAINING**. The complete syntax for the **CONTAINING** operator is:

```
<value1> [NOT] CONTAINING '<string>'
```

Example

The following cursor declaration retrieves all load numbers where the shipper's name contains "ZEHRS".

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_SHIP_NAME CONTAINING 'ZEHRS';
```

4.2.4.3. IN

IN tests to see if a value equals at least one value in a list of values. A list is either a set of values separated by commas and enclosed by parentheses or the results of a subquery. The complete syntax for the **IN** operator is:

```
<value> [NOT] IN (<value1> [, <value2] ...)

OR

<value> [NOT] IN (<subquery>)
```

Example

The following cursor retrieves the shipper names for loads 100101, 100102 and 100103.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_SHIP_NAME FROM LOADS
        WHERE LD_LOAD_NUMBER IN (100101, 100102, 100103);
```

The following cursor retrieves the shipper names for all loads whose zip code is 19462.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_SHIP_NAME FROM LOADS
        WHERE LD_LOAD_NUMBER IN (SELECT LD_LOAD_NUMBER FROM LOADS
                                 WHERE LD_SHIP_ZIP = 19462);
```

4.2.4.4. LIKE

LIKE tests a value against a case sensitive string containing wildcards. Wildcards are characters that substitute for a single, variable character or a number 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 complete syntax for the **LIKE** operator is:

```
<value1> [NOT] LIKE <value2> [ESCAPE 'character']
```

To test a value to see if it contains one of the wildcard characters, precede the wildcard character with the character given in the **ESCAPE** clause.

Example

The following cursor retrieves all loads whose shipper's name begins with "3M".

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_SHIP_NAME LIKE '3M%';
```

The following cursor retrieves all loads whose shipper's name contains an underscore.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_SHIP_NAME LIKE '%@_%' ESCAPE '@';
```

4.2.4.5. IS NULL

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

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

Example

The following cursor retrieves all loads which have no value in the pro number column.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_PRO_NUMBER IS NULL;
```

4.2.4.6. STARTING WITH

STARTING WITH tests a value to see if it begins with a case sensitive string. The complete syntax of the **STARTING WITH** operator is:

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

Example

The following cursor retrieves all loads whose shipper's name begins with “3M”.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_SHIP_NAME STARTING WITH '3M';
```

4.2.4.7. ALL

ALL tests that a condition is true when compared to every value in a list returned by a subquery. The complete syntax of the **ALL** operator is:

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

Example

The following cursor retrieves all loads where the total revenue is greater than all loads whose shipper zip code begins with “194”.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE LD_TTL_REVENUE > ALL (SELECT LD_TTL_REVENUE FROM LOADS
                                    WHERE LD_SHIP_ZIP STARTING WITH '194');
```

4.2.4.8. ANY or SOME

ANY or **SOME** tests that a condition is true when compared to any value in a list returned by a subquery. The complete syntax for the **ANY** operator is:

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

Example

The following cursor retrieves the driver's first and last names for any driver currently assigned to loads whose shipper's zip code begins with “194”.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT OPDRV_LAST_NAME, OPDRV_FIRST_NAME FROM OP_DRIVERS
        WHERE OPDRV_ID = ANY (SELECT LD_DRIVER1 FROM LOADS
                              WHERE LD_SHIP_ZIP STARTING WITH '194');
```

4.2.4.9. EXISTS

EXISTS tests that there is at least one qualifying row meeting the criteria for the search condition given by a subquery. This is the fastest possible way to test for the existence of a value in a table. The complete syntax of the **EXISTS** operator is:

```
[NOT] EXISTS (<subquery>)
```

Example

The following cursor retrieves all loads where the shipper's state does not exist in the state code table.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE NOT EXISTS (SELECT * FROM CT_STATE
                          WHERE CTST_CODE = LD_SHIP_STATE);
```

4.2.4.10. SINGULAR

SINGULAR tests that there is *exactly* one qualifying row meeting the criteria for the search condition given by a subquery. The complete syntax for the **SINGULAR** operator is:

```
[NOT] SINGULAR (<subquery>)
```

Example

The following cursor selects all loads which have one line item.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LD_LOAD_NUMBER FROM LOADS
        WHERE SINGULAR (SELECT * FROM LD_STOP_ITEMS
                        WHERE LDS_LOAD_NUMBER = LD_LOAD_NUMBER);
```

4.2.5. Logical Operators

Firebird provides three logical operators: **NOT**, **AND** and **OR**.

- **NOT** evaluates to the negative of the condition to which it is applied and has the highest precedence.
  - **AND** evaluates to True if both of the conditions to which it applies are True. It evaluates to False otherwise. It has the next highest precedence after **NOT**.
  - **OR** evaluates to True if either of the conditions to which is applies is True. It evaluates to False otherwise. It has the same precedence as **AND**.

Example

```
NOT COLUMN1 = COLUMN2
```

Evaluates to True if COLUMN1 is not equal to COLUMN2.

```
COLUMN1 = COLUMN2 AND COLUMN3 = COLUMN4
```

Evaluates to True if both COLUMN1 equals COLUMN2 and COLUMN3 equals COLUMN4.

```
COLUMN1 = COLUMN2 OR COLUMN3 = COLUMN4
```

Evaluates to True if either COLUMN1 equals COLUMN2 or COLUMN3 equals COLUMN4.

4.2.6. NULLs in Expressions

NULL values can be quite a conceptual challenge for those who have previously worked in environments, such as Cobol, that don't have NULL values. In SQL, any data item will be stored with a NULL indicator if no value is ever provided for it in a DML statement or through a default. A NULL is not equivalent to a blank text column or a zero numeric column. NULL values are treated quite differently in expressions than blank or zero values. Comparisons that encounter NULL on either the left or right side of the operator always follow SQL rules of logic and evaluate the result of the comparison as NULL and return False.

NULL is not a value, so it cannot be equal to anything. For example, an expression such as:

```
WHERE (COL1 = NULL)
```

will return an error because the equality operator is not value for NULLs. The correct expression to test for NULLs is:

```
WHERE (COL1 IS NULL)
```

Two NULLs are not equal to each other. Be aware of the case where an expression must resolve to:

```
WHERE (<null_value> = <null_value>)
```

because False is always the result when two NULLs are compared.

In an expression where a column identifier “stands in” for the current value of a data item, a NULL operand in a calculation will produce NULL as the result of the calculation. For example, the following:

```
UPDATE TABLEA SET COL4 = COL4 + COL5;
```

will set COL4 to NULL if COL5 is NULL.

In aggregate expressions using operators like SUM(), AVG(), and COUNT() rows containing NULL are ignored.

Semmantically, if an expression returns NULL, it is neither false nor true. However, in SQL comparisons resolve as either true or false, a comparison that does not evaluate to true is false. This can trip you up if you are using NOT in expressions that involve NULLs because:

```
NOT <condition evaluating to false> evaluates to true
```

but

```
NOT <condition evaluating to NULL> evaluates to NULL
```

Example

```
NOT (COLUMNA = COLUMNB)
```

If both COLUMNA and COLUMNB have values and they are not equal, the inner expression evaluates to false. The expression NOT (false) returns true.

However, if either of the columns is NULL, the inner expression evaluates to NULL. The expression NOT (NULL) evaluates to NULL and returns NULL.

4.2.7. Computed Columns

SQL is not limited to returning the values of columns as the result of a query, it can also return the value of expressions. Output columns which are created using expressions are referred to as computed columns. The value of a computed column is always read only since it is not a stored value and, as such, cannot be updated to a new value. Any expression that returns a single value can be used to specify a computed column.

To enable you to provide run time names for computed columns Firebird supports the SQL column aliasing standard, which allows any column to be output using an alias. For example, the following:

```
SELECT COLUMN2 || ',' || COLUMN3 AS COMPUTED_COLUMN FROM TABLE
```

returns a column concatenating two other column values separated by a comma and names it COMPUTED_COLUMN.

4.2.8. Built in Functions

Firebird comes with a minimal set of internally implemented SQL functions. It is also possible to extend Firebird with user written, external functions called UDFs (User Defined Functions).

Function	*Type*	*Description*
CAST()	Conversion	Converts a column from one data type to another.
EXTRACT()	Conversion	Extracts date and time parts from DATE, TIME and TIMESTAMP values
SUBSTRING()	String	Retrieves any sequence of characters from a string.
UPPER()	String	Converts a string to all upper case characters.
GEN_ID()	General	Returns the value from a generator.
AVG()	Aggregating	Calculates the average of a set of values.
COUNT()	Aggregating	Returns the number of rows that satisfy a query's search condition.
MAX()	Aggregating	Retrieves the maximum value from a set of values.
MIN()	Aggregating	Retrieves the minimum value from a set of values.
SUM()	Aggregating	Retrieves the total of a set of numeric values.
CASE()	General	Returns the value of an expression based on the value of a group of mutually exclusive conditions.
COALESCE()	General	Returns the value of the first non-null argument.
NULLIF()	General	Returns either a a value or NULL depending on whether or not the two arguments match.

Table 3Internal Functions

4.2.8.1. CAST

The **CAST** function allows a value to be converted from one data type to another, compatible data type. The syntax of the **CAST** function is:

```
CAST(<value> AS <datatype>)
```

From Data Type	*To Data Type*
Numeric	Character, varying character, date
Character, varying character	Numeric, date
Date	Character, varying character, date

Table 4Compatible Data Types for CAST()

Example

In the following WHERE clause, a character value, INTERVIEW_DATE, is converted to a date data type for comparison to a date value, HIRE_DATE.

```
WHERE HIRE_DATE = CAST(INTERVIEW_DATE AS DATE)
```

4.2.8.2. EXTRACT

The **EXTRACT** function extracts part of a DATE, TIME or TIMESTAMP field as a number. The syntax of the **EXTRACT** function is:

```
EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | WEEKDAY | YEARDAY}
        FROM <value>)
```

All parts return a SMALLINT except for SECOND which returns DECIMAL(6,4).

Example

The following statement returns the shipper's appointment date as three numeric columns:

```
EXEC SQL
    SELECT EXTRACT(YEAR FROM LD_SHIP_APPT_LOW) AS YEAR,
           EXTRACT(MONTH FROM LD_SHIP_APPT_LOW) AS MONTH,
           EXTRACT(DAY FROM LD_SHIP_APPT_LOW) AS DAY
        FROM LOADS;
```

4.2.8.3. SUBSTRING

The **SUBSTRING** function returns a string of consecutive characters from another string. The syntax of the **SUBSTRING** is:

```
SUBSTRING(<value> FROM <startpos> [FOR <length>])
```

SUBSTRING returns characters from <value> beginning with the character at <startpos>. If the optional **FOR** <length> clause is given the function will return the lesser of <length> bytes or the number of bytes to the end of <value>. If the **FOR** clause is omitted the function returns the number of bytes from <startpos> to the end of <value>. The first character of <value> is position 1.

Example

The following function call will return all characters from character position 4 to the end of COLUMN.

```
SUBSTRING(COLUMN FROM 4)
```

The following function call will return up to 50 characters from COLUMN starting at position 4.

```
SUBSTRING(COLUMN FROM 4 FOR 50)
```

4.2.8.4. UPPER

The **UPPER** function converts a string value to all upper case characters. The syntax for **UPPER** is:

```
UPPER(<value>)
```

Example

The following function call returns “THIS IS A TEST”.

```
UPPER('This is a test')
```

4.2.8.5. GEN_ID

The **GEN_ID** function calculates and returns a value from a generator. Generators are a means of generating a series of numbers which are ideally suited to use as autoincrementing keys. The syntax of the **GEN_ID** function is:

```
GEN_ID(<generator>, <increment>)
```

GEN_ID returns the last value of <generator> plus the value of <increment>. It then updates <generator> to the value just returned.

Example

If the generator named “MyGenerator” has a value of 4 then the following function call would return 6 and set the value of “MyGenerator” to 6.

```
GEN_ID('MyGenerator', 2);
```

4.2.8.6. AVG

The **AVG** function returns the average of the values of a single column returned by a query. The syntax of the **AVG** function is:

```
AVG(<column>)
```

Example

The following select statement returns the average revenue of all loads:

```
SELECT AVG(LD_TTL_REVENUE) FROM LOADS;
```

4.2.8.7. COUNT

The **COUNT** function returns the number of rows that satisfy a query's search condition. The syntax of the **COUNT** function is:

```
COUNT({* | [ALL] <column> | DISTINCT <column>})
```

“*” returns the number of rows in the result set, including NULL values.

ALL returns the number of non-NULL values in <column>

DISTINCT returns the number of unique, non-NULL values in a column

Example

The following select statement returns the number of rows in the loads table.

```
SELECT COUNT(*) FROM LOADS;
```

4.2.8.8. MAX

The **MAX** function returns the maximum value from a set of values. The syntax of the **MAX** function is:

```
MAX(<column>)
```

Example

The following select statement returns the maximum revenue of all loads.

```
SELECT MAX(LD_TTL_REVENUE) FROM LOADS;
```

4.2.8.9. MIN

The **MIN** function returns the minimum value from a set of values. The syntax of the **MIN** function is:

```
MIN(<column>)
```

Example

The following select statement returns the minimum revenue of all loads.

```
SELECT MIN(LD_TTL_REVENUE) FROM LOADS;
```

4.2.8.10. SUM

The **SUM** function returns the total of a set of numeric values. The syntax of the **SUM** function is:

```
SUM(<column>)
```

Example

The following select statement returns the total revenue for all loads.

```
SELECT SUM(LD_TTL_REVENUE) FROM LOADS;
```

4.2.8.11. CASE

The **CASE** function returns a value determined by the outcome of evaluating a group of mutually exclusive conditions. The syntax of the **CASE** function is:

```
CASE {<value1> | <empty_clause>}
  WHEN {NULL | <value2> | <condition1>} THEN {<result1> | NULL}
  WHEN ... THEN {<result2> | NULL}
  [WHEN ... THEN {<resultn> | NULL}] ...
  [ELSE {<resultn + 1> | NULL}]
END
```

WHEN ... THEN are the keywords in each condition / result clause. At least one condition / result clause is required.

ELSE precedes an optional “last resort” clause, to be returned if none of the conditions in the preceding clauses is met.

<value1> is the identifier of the column value that is to be evaluated. It can be omitted, in which case each **WHEN** clause must be a condition that references the same column identifier.

<value2> is the value that is matched against **<value1>**. It is either a literal or an expression that evaluates to a data type that is compatible with **<value1>**'s data type.

<condition1> is a conditional expression which must be used when **<value1>** is omitted. In this case, both **<value1>** and **<value2>** must be contained in **<condition1>**.

<result1> is the value that will be returned if **<value1>** matches **<value2>**.

Example

The following select statement will return an English language description of the unit types of all units in the OPERATIONS table.

```
SELECT CASE OP_UNIT_TYPE
         WHEN 1 THEN 'Tractor'
         WHEN 2 THEN 'Trailer'
         WHEN 5 THEN 'Driver'
         ELSE 'Unknown type'
       END
    FROM OPERATIONS;
```

The following select statement also returns an English language description of the unit types of all units in the OPERATIONS table.

```
SELECT CASE
         WHEN OP_UNIT_TYPE = 1 THEN 'Tractor'
         WHEN OP_UNIT_TYPE = 2 THEN 'Trailer'
         WHEN OP_UNIT_TYPE = 5 THEN 'Driver'
         ELSE 'Unknown type'
       END
    FROM OPERATIONS;
```

4.2.8.12. COALESCE

The **COALESCE** function evaluates a series of expressions. The value of the first expression to return a non-null value is returned. The syntax of the **COALESCE** function is:

```
COALESCE(<value1> [, <value2> ...])
```

Example

The following select statement returns either a load's pro number or the string 'Not billed' if no pro number exists.

```
SELECT COALESCE(LD_PRO_NUMBER, 'Not billed') FROM LOADS;
```

4.2.8.13. NULLIF

The **NULLIF** function returns either the value of its first argument or NULL if the value of the first argument matches the value of the second argument. The syntax of the **NULLIF** function is:

```
NULLIF(<value1>, <value2>)
```

Example

The following update statement sets the value LD_TTL_REVENUE to NULL if it equals zero.

```
UPDATE LOADS SET LD_TTL_REVENUE = NULLIF(LD_TTL_REVENUE, 0);
```

4.3. Retrieving Data

All data stored in Firebird tables is retrieved in only one way, by querying it using a SELECT statement. A query defines a logical collection of data items arranged in order from left to right in one or more columns known as a set. The data items may come from a single table or multiple tables. A query may consist of a single row or multiple rows. The rows can be in no particular order or they can be returned as a sorted set.

4.3.1. The SELECT statement

```
SELECT [TRANSACTION transaction_name]
    [FIRST (m)] [SKIP (n)]
    [DISTINCT | ALL] {* | <val> [, <val> ...]}
    [INTO :var [, :var ...]]
    FROM {table | view | stored_procedure}
      [[[INNER] | {LEFT | RIGHT | FULL} [OUTER]]
       JOIN {table | view | stored_procedure}
       ON <join_condition> [JOIN ...]]
    [WHERE <search_condition>]
    [GROUP BY col [, col ...]]
```

```
    [HAVING <search_condition>]
    [UNION <select_expression>]
    [PLAN <plan_expression>]
    [ORDER BY <order_list>]
    [FOR UPDATE] [OF col [, col ...] [WITH LOCK]

<val> = expression [AS alias] | *

<join_condition> = a conditional expression relating columns from the table to
be joined to the master table.  Values from the rows which match the conditions
will be included in the result set.

<search_condition) = a conditional expression specifying which rows are to be
included from the master table.

<selection_condition> = a subquery specifying additional rows to be appended to
the result set.

<plan_expression> = a clause which tells the database manager which indices
should be used to evaluate the query.  This clause is rarely used and is not
discussed in this manual.

<order_list> = {col | integer} [ASC[ENDING] | DESC[ENDING]]
               [, <order_list>]
```

If you can't make much sense out of the formal definition of the SELECT statement, don't worry. Each clause will be discussed in detail in the following sections.

4.3.2. Named Transactions

If your application is making use of named transactions, you can tell the database manager which transaction to use to control the SELECT statement by including the optional **TRANSACTION** clause. The **TRANSACTION** keyword is simply followed by a transaction name that has been declared in a previous SET TRANSACTION statement.

4.3.3. Set Qualifiers

The optional set qualifiers **FIRST**, **SKIP**, **ALL** and **DISTINCT** can be included to govern the inclusion or suppression of rows in the result set once they have met all other conditions.

ALL is the default qualifier and is usually omitted. It returns all rows which meet the search conditions to the result set.

The **DISTINCT** qualifier suppresses all duplicate rows in the result set. All columns in two rows must be identical for them to be considered duplicates.

The **FIRST (m)** and **SKIP (n)** qualifiers provide the means to include the first m rows of a result set and to skip the first n rows of the result set respectively. The arguments m and n are integers or expressions that evaluate to integers.

Example

The following cursor returns 5 rows starting at row 101 of the result set.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT FIRST (5) SKIP (100) LD_LOAD_NUMBER
        FROM LOADS
        ORDER BY LD_LOAD_NUMBER;
```

4.3.4. List of Columns

The SELECT statement must return at least one column, which does not have to be part of a table. Each column specification is really an expression that can return any value an expression can. The asterisk (*) is a special symbol that returns every column from a table. It is possible to provide an alternate name, or alias, for a column by using the **AS** clause. Ambiguous column names can be qualified by prefixing the column name with the appropriate table's alias.

Example
The following cursor returns all rows from the LOADS table.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
```

SELECT * FROM LOADS;

The following cursor returns all columns from the LOADS table as well as the tractor id from the OPERATIONS table.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LOADS.*, OPERATIONS.OP_UNIT_ID
        FROM LOADS
          LEFT JOIN OPERATIONS ON OP_ID = LD_TRACTOR;
```

The following cursor returns the year, month and day from the LD_STATUS_DATE column as separate columns aliased as STAT_YEAR, STAT_MONTH and STAT_DAY respectively.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT EXTRACT(YEAR FROM LD_STATUS_DATE) AS STAT_YEAR,
               EXTRACT(MONTH FROM LD_STATUS_DATE) AS STAT_MONTH,
               EXTRACT(DAY FROM LD_STATUS_DATE) AS STAT_DAY
        FROM LOADS;
```

4.3.5. Specifying Host Variables for Returned Columns

A SELECT statement that returns a single row (singleton SELECT) returns data to a list of host variables specified by the **INTO** clause. Each host variable in the **INTO** clause must be preceded by a colon (:) and separated from the preceding host variable by a comma. The number, order and data type of the host variables must correspond to the number, order and data type of the columns retrieved. Otherwise, overflow, data conversion or compile errors may result.

Example

The following select statement retrieves the load number, status and status date into three host variables.

```
EXEC SQL
    SELECT LD_LOAD_NUMBER, LD_STATUS, LD_STATUS_DATE
    INTO :WS-LOAD-NUMBER, :WS-STATUS, :WS-STATUS-DATE
    FROM LOADS
    WHERE LD_LOAD_NUMBER = 123456;
```

4.3.6. Data Source

The **FROM** clause specifies the source of the data, which may be a table, a view or a stored procedure that has output arguments. If the statement involves joining two or more structures, the **FROM** clause specifies the leftmost structure. Other tables are added to the specification by way on succeeding **JOIN** clauses.

It is also possible to join multiple tables using the SQL-89 implicit inner join syntax. This involves listing the tables to be joined on the **FROM** clause, separated by commas. This method of joining tables is deprecated and should be avoided in favour of the explicit **JOIN** syntax.

Example

The following cursor returns all rows from the LOADS table.

EXEC SQL
DECLARE C1 CURSOR FOR
SELECT * FROM LOADS;

The following cursor returns all rows from the LOADS table as well as the tractor id from the OPERATIONS table.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LOADS.*, OPERATIONS.OP_UNIT_ID
        FROM LOADS
          LEFT JOIN OPERATIONS ON OP_ID = LD_TRACTOR;
```

The following cursor returns all rows from the LOADS table as well as the tractor id from the OPERATIONS table using the SQL-89 inner join syntax.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT LOADS.*, OPERATION.OP_UNIT_ID
        FROM LOADS, OPERATIONS
```

```
        WHERE OP_ID = LD_TRACTOR;
```

4.3.7. Search Conditions

The **WHERE** clause specifies the search conditions which limit the number of rows returned. Search conditions my be a simple match condition on a single column or a complex expression involving the **AND**, **OR** and **NOT** operators, type casting, function calls and more. Search conditions can contains references to host variables which can be used to specify values in the expression at run time. Host variables must be preceded by a colon (:).

Example

The following cursor returns all columns from the LOADS table for rows whose status date is on or between two user supplied values.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
    SELECT * FROM LOADS
    WHERE LD_STATUS_DATE BETWEEN :WS-LOW-DATE AND :WS-HIGH-DATE;
```

The above example could be restated as follows:

```
EXEC SQL
    DECLARE C1 CURSOR FOR
    SELECT * FROM LOADS
    WHERE LD_STATUS_DATE >= :WS-LOW-DATE AND
          LD_STATUS_DATE <= :WS-HIGH-DATE;
```

4.3.8. Grouping Output

The output from the SELECT statement can be optionally be partitioned into one or more groups that summarize the sets of data returned at each level. This is accomplished using the **GROUP** BY clause. These groupings often include aggregating expressions which work on multiple values, such as totals, averages, row counts and minimum / maximum values. A grouped select returns one row for each group value. For example, the result set 192, 193, 193, 194, 194 would return one row for each of 192, 193 and 194.

TIP: It is usually necessary to include an ORDER BY clause in SELECT statements in which you want to group the output. This is due to the fact that result sets are unordered unless you specify the ORDER BY clause. While **GROUP BY** will group the output in this case you will get multiple occurrences of the same group value if it occurs sporadically throughout the result set. For example, the result set 194, 192, 192, 194 ... would return rows for 194, 192 and 194.

Example

The following cursor returns the total and average revenue for all loads summarized by the first three digits of the zip code.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
    SELECT SUBSTRING(LD_SHIP_ZIP FROM 1 FOR 3),
           SUM(LD_TTL_REVENUE),
           AVG(LD_TTL_REVENUE)
    FROM LOADS
    GROUP BY SUBSTRING(LD_ZHIP_ZIP FROM 1 FOR 3)
    ORDER BY SUBSTRING(LD_SHIP_ZIP FROM 1 FOR 3);
```

4.3.9. Limiting Groups Returned

The **HAVING** clause may be used in conjunction with the **GROUP BY** clause to include or exclude groups similar to the way the **WHERE** clause limits output.

Example

The previous example can be modified to limit the output to groups where the total revenue is greater than zero.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT SUBSTRING(LD_SHIP_ZIP FROM 1 FOR 3),
      SUM(LD_TTL_REVENUE),
```

```
    AVG(LD_TTL_REVENUE)
FROM LOADS
GROUP BY SUBSTRING(LD_ZHIP_ZIP FROM 1 FOR 3)
HAVING SUM(LD_TTL_REVENUE) > 0
ORDER BY SUBSTRING(LD_SHIP_ZIP FROM 1 FOR 3);
```

4.3.10. Combining Queries

Result sets from two or more queries can be combined into one result set using the **UNION** clause. Each column in the queries to be combined must agree in order, data type and size with all of the other queries. By default a **UNION** result set suppresses duplicates. To retain the duplicates, include the **ALL** keyword.

Example
The following cursor returns all loads whose shipper zip code is 19462 or 19463. There are easier ways to do this, this just servers as an example of the **UNION** clause.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT A.*
  FROM LOADS A
  WHERE A.LD_SHIP_ZIP = '19462'
  UNION SELECT B.*
      FROM LOADS B
      WHERE B.LD_SHIP_ZIP = '19463';
```

4.3.11. Ordering Output

By default, the SELECT statement returns rows unordered. This is true even if the search condition is such that the query can be satisfied using an index.

The **ORDER BY** clause allows you to specify an ordering on the result set. The **ORDER BY** keywords are followed by a list of columns, by which the result set will be ordered. These columns do not need to appear in the output specification nor do they need to appear in an index. The result set will be sorted if necessary.

The full syntax of the **ORDER BY** clause is:

```
ORDER BY {col | <expression> | degree} [ASC | DESC] [NULLS FIRST | NULLS LAST]
         [, ...]
```

col is the name of a column.

<expression> is any non-conditional expression

degree is a number representing the position of a value in the SELECT statement's list of returned values. 1 is the first value.

ASC indicates that the ordering is the be ascending for the column, expression or degree.

DESC indicates that the ordering is the be descending for the column, expression or degree.

NULLS FIRST/LAST indicates that **NULL** values are to come first/last in the ordering. This feature is available from Firebird 1.53. forward.

Example

The following cursor returns the all loads, sorted by the shipper's appointment date.

```
EXEC SQL
    SELECT *
    FROM LOADS
    ORDER BY LD_SHIP_APPT_LOW;
```

4.3.12. Controlling Row Locking

With Firebird, locking is governed by the relative ages of transactions and the records managed by Firebird's versioning engine. All locking applies at the row level, except when a transaction is operating in SNAPSHOT STABILITY isolation or with a table reservation restriction that blocks write access.

The timing of a lock on a row in normal read write activity is optimistic, no locking is in force on any row until the moment it is actually required. Until an update of the row is posted to the server, the row is free to be "won" by any read write transaction.

Pessimistic, or preemptive, locking can be applied to sets of rows or to entire tables. The table locking options have already been introduced in the chapter discussing transaction management. Pessimistic, row level locking is managed by use of the **FOR UPDATE** and **WITH LOCK** clauses. Pessimistic locks are applied at the time the row is read.

It is important to note that a row is not necessarily read when you might think. Under normal circumstances, the Firebird database manager buffers rows and returns them to the client in blocks. This means that rows are read, and possibly locked, before your application actually receives them using the FETCH statement. Like most things with Firebird, there are a couple of exceptions to this. First, if you use **UPDATE ... WHERE CURRENT OF** to update rows from the cursor, buffering does not occur. Second, if you are using DSQL and use **SELECT ... FOR UPDATE** buffering does not occur. In both of these cases records are read as they are FETCHed.

All locks are released when the transaction is committed or rolled back.

Optimistic locking is conceptually difficult for Cobol programmers who are used to using pessimistic locking. When using pessimistic locking, lock conflicts are not detected until the row is UPDATEd or until the transaction is committed, at which time the application will receive a "lock conflict" error. The application must then decide whether to simply rollback the current transaction and discard the changes or rollback the current transaction and retry the entire transaction again. While this approach is often more efficient due to reduced potential for deadlocks, it is much more difficult to program then the pessimistic approach.

As mentioned above, the **FOR UPDATE** clause is used to disable buffering. **FOR UPDATE** is only effective if you are using DSQL. While this clause is allowed in embedded SQL applications, it is ignored and has no effect on buffering.

Normally rows are locked at the time that they are updated using optimistic locking. If you want to use pessimistic locking and lock rows at the time they are read you must use the **WITH LOCK** clause.

Example

The following code fragment shows an application that will work in much the same way as a standard Cobol application using ISAM files and traditional record locking techniques.

```
EXEC SQL
    SET TRANSACTION READ WRITE READ COMMITTED WAIT NO RECORD_VERSION;

EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT *
        FROM LOADS
        FOR UPDATE WITH LOCK;

EXEC SQL
    OPEN C1;

    PERFORM UNTIL EOF
        EXEC SQL
            FETCH C1 INTO :WS-VAR1, :WA-VAR2, ...;

        EXEC SQL
            UPDATE LOADS SET COL1=:WS-VAR1, ...
            WHERE CURRENT OF C1;

        EXEC SQL
            COMMIT RETAIN;
    END-PERFORM.

EXEC SQL
    CLOSE C1;

EXEC SQL
```

```
COMMIT;
```

4.3.13. Selecting a Single Row

An operation that retrieves a single row of data is called a *singleton select*. To select a single row from a table or to retrieve an aggregate value like **COUNT()** or **AVG()** use the following SELECT statement syntax.

```
SELECT col [, col ...]
    INTO :var [, :var ...]
    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 on 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 the only one row is retrieved, otherwise the SELECT fails.

4.3.14. Selecting Multiple Rows

Many queries specify search conditions that retrieve more than one row. Since host variables can only hold a single column value at a time, a query that returns multiple rows must build a temporary table called a result set. Rows are extracted from the result, one at time, in sequential order. The database manager keeps track of the next row to process from the result set by establishing a pointer to it, called a *cursor*.

To retrieve multiple rows into a result set, establish a cursor into the table and process the individual rows in the table, SQL provides the following sequence of statements:

- DECLARE CURSOR establishes a name for the cursor and specifies the query to perform.
- OPEN executes the query, builds the result set and positions the cursor at the start of the set.
- FETCH retrieves a single row from the result set int host variables.
- CLOSE releases system resource when all rows are retrieved.

4.3.14.1. Declaring a Cursor

To declare a cursor and specify rows of data to retrieve, use the **DECLARE CURSOR** statement. **DECLARE CURSOR** is a non-executable statement which prepares system resources for the cursor but does not actually perform the query. The syntax for **DECLARE CURSOR** is:

```
EXEC SQL
    DECLARE cursor_name CURSOR FOR
    SELECT ...;
```

cursor_name is the name of the cursor which is used in subsequent **OPEN**, **FETCH** and **CLOSE** statements.

The SELECT statement inside the **DECLARE CURSOR** statement is the same as a singleton select except it cannot include an **INTO** clause.

4.3.14.2. 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 the result set. The syntax of the **OPEN** statement is:

```
EXEC SQL
    OPEN cursor_name;
```

cursor_name is the name of the cursor as declared in a previous **DECLARE CURSOR** statement.

When Firebird opens the cursor it is positioned to the first row of the result set.

4.3.14.3. Fetching Rows from a Cursor

Once a cursor is opened, rows can be retrieved from the result set using the **FETCH** statement. **FETCH**:

- Retrieves the next available row from the result set.
- Copies the columns for that row into the host variables specified in the **INTO** clause of the **FETCH**statement.
  - Advances the cursor to the next available row or sets **SQLCODE** to 100 to indicate that the cursor is at the end of the result set.

The syntax of the **FETCH** statement is:

```
EXEC SQL
    FETCH cursor_name
    INTO :var [[INDICATOR] :indicator] [, :var [[INDICATOR] :indicator] ...]
```

cursor_name is the name of the cursor as declared in a previous **DECLARE CURSOR** statement.

var is the name of a host variable which will receive the value of the corresponding column

indicator is the name of a host variable that will receive the NULL status flag for the column.

Every **FETCH** statement should be tested to see if the end of the result set is reached. You do this by testing the **SQLCODE** variable. A value of 100 indicates that the end of the result has been reached. A value of less than zero indicates that an error was encountered. A value of zero indicates that a row was successfully fetched.

Any column can have a NULL value, except those defined with the NOT NULL or UNIQUE integrity constraints. To determine if the value returned for a column is NULL, follow each variable named in the **INTO** clause with the optional **INDICATOR** keyword and the name of a PIC S9 variable, called the *indicator variable* where Firebird will store the NULL status flag for the column. A value of -1 indicates that the column is NULL. A value of zero indicates that the column is NOT NULL.

It is only possible to read forward through a cursor. To revisit previously fetched rows, close the cursor and reopen it.

4.3.14.4. Closing a Cursor

When you are done with a cursor it should be closed to free up system resources. To close a cursor use the **CLOSE** statement. The syntax of the **CLOSE** statement is:

```
EXEC SQL
    CLOSE cursor_name
```

cursor_name is the name of the cursor as declared in a previous **DECLARE CURSOR** statement.

4.4. Inserting Data

New rows of data are added to one table at a time with the **INSERT** statement. **INSERT** assigns values to columns listed between the first set of parentheses. Columns not listed are given a NULL value. The **INSERT** statement allows data insertion from two different sources; a list of host variables or a SELECT statement that retrieves values from one table to add to another.

4.4.1. The INSERT Statement

Format 1

```
INSERT TRANSACTION transaction_name
INTO table_name (col [, col ...])
VALUES (<value> [:indicator] [, <value> [:indicator] ...]);
```

Format 2

```
INSERT TRANSACTION transaction_name
INTO table_name (col [, col ...])
SELECT ...;
```

Use **Format 1** to insert a new row using values from your application.

Use **Format 2** to insert a new row using values retrieved from another table or tables. This methods adds one new row for each row returned by the SELECT statement.

transaction_name is the name of a transaction declared by a previous SET TRANSACTION statement.

table_name is the name of the table to receive the new row.

col is the name of a column which is to receive a new value.

<value> is an expression giving a new value. It may be any expression that returns a value compatible with the data type of the corresponding column.

indicator is a optional host variable containing the NULL status flag of the corresponding column. A value of -1 indicates that the column is NULL, zero indicates NOT NULL.

Example
The following **INSERT** statement adds one new driver (OP_UNIT_TYPE = 5) to the OPERATIONS table.

```
EXEC SQL
   INSERT INTO OPERATIONS
   (OP_ID, OP_UNIT_ID, OP_UNIT_TYPE)
   VALUES (:WS-ID, :WS-UNIT-ID, 5);
```

The following **INSERT** statement adds a series of new rows to the OPERATIONS table from a temporary table holding rows to be imported.

```
EXEC SQL
    INSERT INTO OPERATIONS (OP_ID, OP_UNIT_ID, OP_UNIT_TYPE)
        SELECT TEMP_ID, TEMP_UNIT_ID, TEMP_UNIT_TYPE
        FROM TEMP_TABLE;
```

4.5. Updating Data

To change values for existing rows of data in a table, use the **UPDATE** statement. **UPDATE** changes the values of columns specified in the **SET** clause. Columns not appearing in the **SET** clause are not changed. A single **UPDATE** statement can modify any number of rows in a table.

4.5.1. The UPDATE statement

```
UPDATE TRANSACTION transaction_name table_name
SET col=<value> [, col=<value> ...]
[WHERE <search_condition> | WHERE CURRENT OF cursor_name]
```

transaction_name is the name of a transaction declared in a previous SET TRANSACTION statement.

table_name is the name of the table to be modified.

col is the name of a column to be modified.

<value> is an expression giving the new value. It may any expression that returns a value compatible with the data type of the corresponding column.

<search_condition> is a conditional expression which determines which row(s) of the table are to be modified.

cursor_name is the name of a cursor declared in a previous DECLARE CURSOR statement. The most recently fetched row from this cursor will be modified if this clause is used.

If neither **<search_condition>** or **cursor_name** is given then the **UPDATE** statement will affect every row in the table.

Example

The following **UPDATE** statement will change the total revenue for load 123456 to $999.00.

```
EXEC SQL
    UPDATE LOADS
    SET LD_TTL_REVENUE = 999.00
    WHERE LD_LOAD_NUMBER = 123456;
```

The following **UPDATE** statement will update the load last read by cursor C1.

```
EXEC SQL
```

```
UPDATE LOADS
SET LD_TTL_REVENUE = 999.00
WHERE CURRENT OF C1;
```

The following **UPDATE** statement will increase the total revenue by 10% for all loads.

```
EXEC SQL
  UPDATE LOADS
  SET LD_TTL_REVENUE = LD_TTL_REVENUE * 1.10;
```

4.6. Deleting Data

To remove rows of data from a table, use the **DELETE** statement. A single **DELETE** statement can be used to remove any number of rows from a table.

4.6.1. The DELETE Statement

```
DELETE TRANSACTION transaction_name
FROM table_name
[WHERE <search_condition> | WHERE CURRENT OF cursor_name]
```

transaction_name is the name of a transaction declared in a previous SET TRANSACTION statement.

table_name is the name of the table to be modified.

<search_condition> is a conditional expression which determines which row(s) of the table are to be deleted.

cursor_name is the name of a cursor declared in a previous DECLARE CURSOR statement. The most recently fetched row from this cursor will be deleted if this clause is used.

If neither **<search_condition>** or **cursor_name** is given then the **DELETE** statement will remove all rows from the table.

Example

The following **DELETE** statement will remove load 123456 from the LOADS table.

```
EXEC SQL
    DELETE FROM LOADS
    WHERE LD_LOAD_NUMBER = 123456;
```

The following **DELETE** statement will remove the last row fetched by cursor C1.

```
EXEC SQL
    DELETE FROM LOADS
    WHERE CURRENT OF C1;
```

The following **DELETE** statement will remove all rows from the LOADS table.

```
EXEC SQL
    DELETE FROM LOADS;
```

5. Accessing Blob Data

A Blob is a dynamically sizable data type that has no specified size and encoding. You can use a Blob to store large amounts of data of various types, including:

- Bitmapped images
- Sound and video
- Text

Because Blobs are large variably sized object Firebird stores them as a series of segments. These segments are indexed by a handle that Firebird generates when you create the Blob. This handle is known as the Blob ID and is a 64 bit value containing a combination of the table and Blob identifiers.

The Blob ID is stored in a column in the table row. The Blob ID points to the first segment of the Blob. You can retrieve the Blob ID by executing a SELECT statement that specifies the Blob as a target. As follows:

```
EXEC SQL
    SELECT OP_LOADS FROM OPERATIONS
    INTO :WS-LOADS-ID
    WHERE OP_UNIT_ID = 'TLR1234';
```

When you create a Blob column in a table, you specify the expected size of the Blob segments for that column. The default segment length is 80 bytes. For most practical purposes the segment size is irrelevant. The exception to this is embedded SQL applications. The embedded SQL precompiler, **gpre**, uses the segment size declared for the column to determine the size of a temporary buffer that is allocated to hold the Blob segments as they are transferred to and from your application. The size of the buffer can be overridden by using the **MAXIMUM_SEGMENT** clause of the DECLARE CURSOR statement. In any case, you do not want to try to read or write more than the segment size in one operation as you will overflow the temporary buffer and the results will be unpredictable. The maximum allowed segment size is 32,767 bytes.

5.1. Retrieving Blob Data

Retrieving a Blob involves two steps; retrieving the Blob ID and retrieving the Blob itself. The Blob ID is retrieved by using a standard singleton select or by using a standard cursor. The Blob data is retrieved using a Blob cursor.

5.1.1. Retrieving the Blob ID

You could retrieve the Blob ID using a singleton select as follows:

```
01  WS-BLOB-ID                  PIC S9(19) USAGE BINARY(8);
...
EXEC SQL
    SELECT OP_LOADS
    INTO :WS-BLOB-ID
    FROM OPERATIONS
    WHERE OP_UNIT_ID = 'TLR1234';
```

Or you could retrieve the Blob ID using a cursor and fetching it as follows:

```
01  WS-BLOB-ID                  PIC S9(19) USAGE BINARY(8).
...
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT OP_LOADS
        FROM OPERATIONS; .
EXEC SQL
    OPEN C1; .
    PERFORM UNTIL SQLCODE = 100
    EXEC SQL
        FETCH C1 INTO :WS-BLOB-ID;
        ...
    END-PERFORM.
    EXEC SQL
        CLOSE C1; .
```

5.1.2. Retrieving the Blob Data

As mentioned previously, Blob data is retrieved using a Blob cursor. This involves several steps; declaring a BLOB READ cursor, opening the cursor, fetching the data and closing the cursor.

5.1.2.1. Declaring READ BLOB Cursors

```
DECLARE cursor_name CURSOR
    FOR READ BLOB col
    FROM table
    [FILTER [FROM subtype] TO subtype]
    MAXIMUM_SEGMENT length
```

cursor_name is the name of the cursor which will be used in subsequent OPEN, FETCH and CLOSE statements.

col is the name of the Blob column.

table is the name of the table containing the Blob.

FILTER ... specifies an optional Blob filter which is used to translate a Blob from one subtype to another. Blob filters are outside the scope of this manual.

length specifies the size of the temporary buffer used to transfer the Blob data to and from the application. Do not try to read or write more than **length** bytes in a single operation or unpredictable results will occur.

Example

The following cursor retrieves the Blob ID for the OP_LOADS column from the OPERATIONS table.

```
EXEC SQL
    DECLARE C1 CURSOR FOR READ BLOB OP_LOADS
    FROM OPERATIONS;
```

5.1.2.2. Opening READ BLOB Cursors

```
OPEN cursor_name USING :var;
```

cursor_name is the name of a cursor previously declared in a DECLARE CURSOR statement.

var is a host variable containing the Blob ID.

Example

The following code snippet declares and opens a read blob cursor.

```
EXEC SQL
    DECLARE C1 CURSOR FOR READ BLOB OP_LOADS
    FROM OPERATIONS;

EXEC SQL
    OPEN C1 USING :WS-BLOB-ID;
```

5.1.2.3. Fetching Blob Data

```
FETCH cursor_name
INTO :var [[INDICATOR] :segment_length];
```

cursor_name is the name of a cursor declared in a previous DECLARE CURSOR statement.

var is the name of the host variable that is to receive the Blob data.

segment_length receives the number of bytes fetched. This may be less then the Blob's segment size when fetching the last segment of a Blob.

FETCH returns to SQLCODE values of interest:

- 100 indicates that there are no more Blob segments to retrieve.
- 101 indicates that the segment was larger that the segment buffer provided by **var**.

Example

The following code snippet declares a Blob cursor, opens it and fetches all segments of the Blob. Fetching of the Blob ID

itself is omitted in the interests of clarity.

```
01  WS-BUFFER        PIC X(1000).
01  WS-LENGTH        PIC 9(4).
01  WS-BLOB-ID       PIC S9(19) BINARY(8).
...
Fetch the Blob ID
...
EXEC SQL
    DECLARE C1 CURSOR FOR READ BLOB OP_LOADS
    FROM OPERATIONS
    MAXIMUM_SEGMENT 1000; .
EXEC SQL
    OPEN C1 USING :WS-BLOB-ID; .
    PERFORM UNTIL SQLCODE = 100
        EXEC SQL
            FETCH C1 INTO :WS-BUFFER :WS-LENGTH;
        ...
    END-PERFORM.
EXEC SQL
    CLOSE C1; .
```

5.1.2.4. Closing READ BLOB Cursors

Read blob cursors are closed in the same way as any other cursor, using the CLOSE statement.

5.2. Updating Blob Data

While it is possible to retrieve Blob data from a table and insert Blob data into a table, it is not possible to directly update a Blob nor to delete one. A Blob is updated by inserting a new Blob and setting the Blob ID to the value of the new Blob's ID. This deletes the old Blob and replaces its ID with the new one. A Blob is deleted in one of two ways; by deleting the row which owns the Blob or by setting the Blob ID to NULL. Inserting a new Blob involves two steps; inserting the Blob using a Blob insert cursor and updating the table with the new Blob ID.

5.2.1. Inserting Blob Data

5.2.1.1. Declaring BLOB INSERT Cursors

```
DECLARE cursor_name CURSOR FOR
    INSERT BLOB col
    INTO table
    [FILTER [FROM subtype] TO subtype]
    MAXIMUM_SEGMENT length;
```

cursor_name is the name of the cursor which will be used in subsequent OPEN, INSERT and CLOSE statements.

col is the name of the Blob column

table is the name of the table containing the Blob

FILTER ... specifies an optional Blob filter which is used to translate a Blob from one subtype to another. Blob filters are outside the scope of this manual.

length specifies the size of the temporary buffer used to transfer the Blob data to and from the application. Do not try to read or write more than **length** bytes in a single operation or unpredictable results will occur.

Example

The following declares a cursor that can be used to insert a new Blob into the OPERATIONS table.

```
EXEC SQL
    DECLARE C1 CURSOR FOR INSERT BLOB OP_LOADS
    INTO OPERATIONS;
```

5.2.1.2. Opening BLOB INSERT Cursors

```
OPEN cursor_name INTO :var;
```

cursor_name is the name of a cursor declared in a previous DECLARE CURSOR statement.

var is the name of a host variable which will receive the new Blob ID.

Example

The following code snippet declares a BLOB INSERT cursor and opens it.

```
EXEC SQL
    DECLARE C1 CURSOR FOR INSERT BLOB OP_LOADS
    INTO OPERATIONS;
EXEC SQL
    OPEN C1 INTO :WS-BLOB-ID;
```

5.2.1.3. Inserting Blob Data

```
INSERT CURSOR cursor_name
VALUES (:buffer [INDICATOR] :segment_length);
```

cursor_name is the name of a cursor declared in a previous DECLARE CURSOR statement.

buffer is a host variable containing the data to be written to the Blob.

segment_length is a host variable holding the number of bytes of data in **buffer**.

Each execution of the **INSERT** statement adds the contents of **buffer** to the Blob. If you need to write more data to the Blob than will fit into a single buffer, which is limited by the segment size of the Blob, you must execute the **INSERT** statement repeatedly until you have written all of the data.

Example

The following code snippet declares a BLOB INSERT cursor, opens it and writes the data to the Blob. Updating the Blob ID in the table is omitted in the interests of clarity.

```
01  WS-BUFFER        PIC X(1000).
01  WS-LENGTH        PIC 9(4).
01  WS-BLOB-ID       PIC S9(19) BINARY(8).
...
EXEC SQL
    DECLARE C1 CURSOR FOR BLOB INSERT OP_LOADS
    INTO OPERATIONS
    MAXIMUM_SEGMENT 1000; .
EXEC SQL
    OPEN C1 INTO :WS-BLOB-ID; .
    PERFORM UNTIL DONE

        ....
        EXEC SQL
            INSERT C1 VALUES(:WS-BUFFER :WS-LENGTH);
    END-PERFORM.
EXEC SQL
    CLOSE C1; .
    ...
    Update Blob ID
    ...
```

5.2.1.4. Closing BLOB WRITE Cursors

Blob write cursors are closed in the same way as any other cursor, using the CLOSE statement.

5.2.2. Updating the Blob ID

After you have obtained a new Blob ID by inserting a Blob into the table you must update the Blob column in the table with the new ID. This is done using a standard UPDATE statement.

You could update the Blob ID using an UPDATE statement as follows:

```
EXEC SQL
    UPDATE OPERATIONS SET OP_LOADS = :WS-BLOB-ID
    WHERE OP_UNIT_ID = 'TLR1234';
```

5.3. Deleting Blob Data

To delete a Blob from a table you could use an UPDATE statement to set the Blob ID to NULL as follows:

```
EXEC SQL
    UPDATE OPERATIONS SET OP_LOADS = NULL
    WHERE OP_UNIT_ID = 'TLR1234';
```

6. Using Stored Procedures

A stored procedure is a self contained set of SQL statements stored in a database as part of its metadata. Stored procedures are created using Firebird's Data Definition Language (DDL) in much the same way that tables and views are created. A discussion of stored procedure programming is outside the scope of this manual.

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.
- *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.

Procedures operate within the context of a transaction. If the transaction is rolled back then any actions performed by the stored procedure are also rolled back. Similarly, a procedure's actions are not final until the transaction is committed.

6.1. Using Select Procedures

Select procedures can be using in singleton selects or cursors in the same way as any table or view. The syntax for calling a select procedure is very similar to that for a table or view. The one difference is that a procedure may have input arguments.

```
SELECT col [, col ...]
[INTO :var, [ :var ...]]
FROM procedure ([arg [, arg ...])
[WHERE ...]
[ORDER BY ...]
```

col is the name of one of the procedure's output parameters.

var is the name of a host variable which will receive the value of the output parameter.

procedure is the name of the stored procedure

arg is the value of one of the procedure's input parameters. The value can be given by either a host variable or a literal that agrees in data type and size with the input parameter.

Example

The following SELECT statement returns the value returned by the OP_NEXT_ID procedure.

```
EXEC SQL
    SELECT OP_ID FROM OP_NEXT_ID()
    INTO :WS-NEXT-ID;
```

The following cursor retrieves values from the GET_OP_HISTORY select procedure.

```
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT OPH_ID, OPH_DATE, OPH_STATUS
        FROM GET_OP_HISTORY(:WS-ID, :WS-LOW-DATE, :WS-HIGH-DATE);
```

6.2. Using Executable Procedures

An executable procedure is called directly by the application. They can receive input parameters from the application and return output parameters to the application. The syntax for calling an executable procedure is:

```
EXECUTE PROCEDURE procedure arg [[INDICATOR] :indicator1]
                           [, arg [[INDICATOR] :indicator1] ...]
                 RETURNING_VALUES :var [[INDICATOR] :indicator2]
                                  [, :var [[INDICATOR] :indicator2] ...];
```

procedure is the name of the stored procedure.

arg is one value of one of the procedure's input parameters. The value can be given by either a host variable or a literal that agrees in data type and size with the input parameter.

indicator1 is the name of a host variable that holds the NULL status flag for the parameter. A value of -1 indicates that the parameter is null. A value of zero indicates not NULL.

var is the name of a host variable which will receive the value of one of the procedure's output parameters.

indicator2 is the name of a host variable which will receive the value of the parameter's NULL status flag. A value of -1 indicates that the parameter is null. A value of zero indicates not NULL.

Example

The following EXECUTE PROCEDURE statement retrieves the value returned by the OP_NEXT_ID procedure.

```
EXEC SQL
    EXECUTE PROCEDURE OP_NEXT_ID
    RETURNING_VALUES :WS-NEXT-ID;
```

7. Using Events

This chapter introduces Firebird events, briefly describes how they work and how to make use of them in your program. The event mechanism allows applications to respond to changes and other database activities that are caused by other, concurrently executing applications without the need for the applications to communicate directly with each other.

A Firebird event is a message passed by a trigger or stored procedure to the Firebird event manager. The message announces the occurrence of a particular condition such as an INSERT, UPDATE or DELETE although the condition is not restricted to these and may, in fact, be anything at all. Event messages are forwarded to waiting applications only when the controlling transaction is committed.

The Firebird event manager maintains a list of event messages submitted by triggers and stored procedures. It also maintains a list of applications that have registered an interest in particular events. Each time the event manager receives a new event message it notifies the interested applications that the event has occurred.

7.1. Signaling Event Occurrence

While a discussion of trigger and stored procedure programming is outside the scope of this document, it may be helpful to provide a quick overview of the method used to signal an event here. An event can only be signaled by a trigger or stored procedure. This is done by using the **POST_EVENT** statement. The **POST_EVENT** statement takes a single, string parameter which is the event name, which are limited to 15 characters in length. This name is used in your applications to register an interest in the event.

Example

To signal an event named “new_order” each time a new row is written to the ORDERS table you might write a trigger like this:

```
CREATE TRIGGER ORDERS_AFTER_INSERT FOR ORDERS
  AFTER INSERT
AS
BEGIN
    POST_EVENT “new_order”;
END
```

7.2. Registering Interest in Events

Your application must register interest in a particular event with the Firebird event manager before waiting for that event to occur. To register interest in an event use the **EVENT INIT** statement.

```
EXEC SQL
    EVENT INIT request_name ( event_name [, event_name ...] );
```

request_name is the name that your application will use to refer to this event. You will use this name to refer to you event in subsequent **EVENT WAIT** statements.

event_name is the name of an event in which you wish to register an interest. You may register an interest in multiple events by specifying the event names here, separated by commas. The event names my be given as string literals or as the content of an application variable.

Examples

```
EXEC SQL
    EVENT INIT MY_EVENT (“event1”, “event2”);
```

```
EXEC SQL
    EVENT INIT MY_EVENT (:WS-EVENT-NAME);
```

7.3. Waiting for an Event

Once you have registered interest in an event using the **EVENT INIT** statement you can wait for event an event to occur by using the **EVENT WAIT** statement.

```
EXEC SQL
    EVENT WAIT request_name;
```

request_name must match the request_name given in the corresponding **EVENT INIT** statement.

Upon executing the **EVENT WAIT** statement your application will be suspending until one of the events listed on the **EVENT INIT** statement occurs. At that time your application will resume executing at the statement following the **EVENT WAIT** statement.

If you registered an interest in more that one event then you must check the **ISC-EVENTS** array to see which event occurred. There will be one entry in this array for each registered event. The order of events in this array correspond to the order in which the event names were given on the **EVENT INIT** statement. Each event that has occurred while your program was waiting will have a non zero value in this array.

Example

```
EXEC SQL
    EVENT INIT MY-EVENT ("event1", "event2"); .

EXEC SQL
    EVENT WAIT MY-EVENT; .
    IF ISC-EVENTS(1) NOT EQUAL 0 THEN
        DISPLAY "Event 1 occurred".
    IF ISC-EVENTS(2) NOT EQUAL 0 THEN
        DISPLAY "Event 2 occurred".
```

8. Miscellaneous Statements

8.1. RELEASE_REQUESTS

Before exiting a subprogram that may be CANCELed by the calling program it is necessary to clean up the request handles generated in the subprogram. This is done using the RELEASE_REQUESTS statement. Simply place this statement before exiting the program via the EXIT PROGRAM verb.

```
EXEC SQL
    RELEASE_REQUESTS FOR database_name;
```

database_name is the name of the database whose handles are to be released, as given in the SET DATABASE statement.

Example

```
EXEC SQL
    RELEASE_REQUESTS FOR TL; .
    EXIT PROGRAM.
```

9. Handling Errors

All embedded SQL application should include mechanisms for trapping and responding to run time errors. Every time an SQL statement is executed, it returns a status indicator in the SQLCODE variable, which is declared automatically during preprocessing with **gpre**. The following table summarized possible SQLCODE values and their meanings.

Value	*Meaning*
0	Success
1-99	Warning or information message
100	End of file
< 0	Error. Statement failed to complete.

Table 5SQLCODE Values

To trap and respond to run time errors, SQLCODE should be checked after each SQL operations. There are three ways to examine SQLCODE and respond to errors:

- Use WHENEVER statements to automate checking SQLCODE and to handle errors when they occur.
- Text SQLCODE directly after individual SQL statements.
- Use a combination of WHENEVER and direct testing.

9.1. WHENEVER

```
WHENEVER {SQLERROR | SQLWARNING | NOT FOUND} {GOTO label | CONTINUE};
```

label is the name of a branch target (label) in your application.

The **WHENEVER** statement allows 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. After **WHENEVER** appears in a program, all subsequent SQL statements automatically jump to the specified code location when the appropriate error or warning occurs. NOTE: **WHENEVER** is a declarative, not executable statement. It applies to all lines of code that occur after the location of **WHENEVER** in the source code.

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 has a value from 1 to 99, indicating that while a statement succeeded 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 FETCH or SELECT.

Omitting a statement for a particular condition means it is not trapped. Error conditions can also be ignored by using the **CONTINUE** clause inside a **WHENEVER** statement.

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 setting overrides any previous setting and remains in effect until overridden in turn.

Example

The following code snippet sets both the SQLERROR and SQLWARNING targets but leaves NOT FOUND detection to the application.

```
EXEC SQL
    WHENEVER SQLERROR GOTO FATAL-ERROR;
EXEC SQL
    WHENEVER SQLWARNING GOTO WARNING-ERROR;
```

The following code snippet sets the SQLERROR target and then resets it to another target.

```
EXEC SQL
    WHENEVER SQLERROR GOTO FATAL-ERROR1;
...
EXEC SQL
    WHENEVER SQLERROR GOTO FATAL-ERROR2;
```

9.2. Checking SQLCODE Directly

An application can test SQLCODE directly after each SQL statement instead of relying on **WHENEVER** to trap all errors. The main advantage to testing SQLCODE directly is that custom error handling routines can be designed for particular situations.

Example

The following code snippet uses **WHENEVER** to trap all fatal errors but testing SQLCODE directly to detect end of file.

```
EXEC SQL
    WHENEVER SQLERROR GOTO FATAL-ERROR;
...
EXEC SQL
   FETCH C1 INTO :WS-COLUMN-VALUE; .
   IF SQLCODE = 100 THEN
       GO TO END-EOF-FILE.
```

9.3. Displaying Error Messages

The same SQLCODE can be returned by many Firebird errors so simply displaying SQLCODE is not an adequate way to report errors that occur in your application. By using the Firebird API it is possible to retrieve the human readable text of the error message which you can then display or log as appropriate.

The text of Firebird error messages is captured using the **isc_interprete** function. This function takes a pointer to the ISC-STATUS-VECTOR host variable and returns the error text associated with the error codes contained therein. To obtain a pointer to the ISC-STATUS-VECTOR host variable, use the **rmc_status_address** function.

9.3.1. rmc_status_address

```
CALL "rmc_status_address" USING ISC-STATUS-VECTOR GIVING ptr
```

ptr is a PIC S9(10) USAGE BINARY(4) host variable which will receive the pointer to ISC-STATUS-VECTOR.

This function creates a temporary copy of ISC-STATUS-VECTOR, converts the error codes from Cobol to native format and returns a pointer to the temporary copy. This means that you must call this function prior to every call to **isc_interprete** in order to have the latest error codes.

NOTE: This function is specific to RM/Cobol and is not a part of the standard Firebird API.

9.3.2. isc_interprete

```
CALL "isc_interprete" USING buffer, ptr GIVING length
```

buffer is a PIC X(512) host variable that will receive the text of the error message. This variable must be large enough to hold the largest expected error message or buffer overflow will occur.

ptr is the host variable containing the status vector pointer returned by **rmc_status_address**.

length is the host variable which will receive the length of the error message that was returned in **buffer**.

It is possible, likely in fact, that multiple error message will be returned for any given status vector. Each time **isc_interprete** is called it returns the text of a single error message and updates **ptr** to point to the next error code. In order to retrieve all error messages it is necessary to call **isc_interprete** repeatedly. **isc_interprete** will return a value of zero in **ptr** when the last error message has been retrieved.

Example

The following is what a typical text mode error paragraph might look like.

```
ISC-ERROR.
    MOVE SQLCODE TO WS-DISP-NUM.
```

```
    DISPLAY "SQL error = ", WS-DISP-NUM.
    CALL "rmc_status_address" USING ISC-STATUS-VECTOR
         GIVING WS-STATUS-PTR.
    CALL "isc_interprete" USING WS-TEXT, WS-STATUS-PTR
         GIVING WS-LENGTH.
    PERFORM UNTIL WS-LENGTH = 0
        DISPLAY WS-TEXT (1:WS-LENGTH)
        CALL "isc_interprete" USING WS-TEXT, WS-STATUS-PTR
             GIVING WS-LENGTH
    END-PERFORM.
EXEC SQL
    WHENEVER SQLERROR CONTINUE; .
EXEC SQL
    ROLLBACK; .
EXEC SQL
    DISCONNECT ALL; .
    STOP RUN.
```

10. Compiling and Running Your Program

Before an embedded SQL program can be compiled, it must be preprocessed with **gpre**. **gpre** translates SQL commands into statements the host language compiler can understand by generating Firebird library function calls. **gpre** translates SQL 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.

10.1. Compiling Your Program

The syntax for the **gpre** command line is:

```
gpre [language] [options] infile [outfile]
```

Language Switches

The language switch specifies the language of the source program. **gpre** will preprocess programs written in the following languages:

Switch	*Language*
-c	C
-cxx	C++
-al[sys]	Ada (Alsys)
-a[da]	Ada (VERDIX, VMS, Telesoft)
-co[bol]	Cobol. In the absence of the -ansi and -rmc switch this switch indicates VMS Cobol.
-ansi	In conjunction with the -cobol switch this switch indicates ANSI-85 Cobol
-rmc	In conjunction with the -cobol switch this switch indicates RM/Cobol
-noqli	Suppresses the recognition and parsing of QLI commands in the source file. QLI can cause problems with some languages, like Cobol, whose reserved word list intersects the QLI reserved word list.
-f[ortran]	Fortran
-pa[scal]	Pascal

Table 6Language Switches

It is possible to avoid the need to specify a language switch when preprocessing your programs by choosing the appropriate extension for the source code file name. The following table lists the file name extension that Firebird recognizes for each language.

Language	*Input File Extension*	*Default Output File Extension*
Ada (VERDIX)	ea	a
Ada (Alsys, Telesoft)	eada	ada
C	e	c
C++	exx	cxx
Cobol	ecbl	cbl
Fortran	ef	f
Pascal	epas	pas

Table 7File Name Extensions

Option Switches

The option switches specify additional, language independent options.

Switch	*Description*
-charset name	Determines the active character set at compile time, where **name** is the character set name.
-d[atabase] path	Declares the database to be used by the current program. Use this option if the program does not contain a DECLARE DATABASE statement.
-d_float	VMS only. Specifies that double precision data will be passed from the application id D_FLOAT format and stored in the database in G_FLOAT format.
-dfm format	Specifies the format to be used to transfer dates and timestamps between the program and Firebird. **format** is a string describing the format. If the switch is omitted, all date and time columns are given as a 64 bit value that must be interpreted using the **isc_encode_date** and **isc_decode_date** functions. See the discussion of timestamp formats below.
-e[ither_case]	Enables **gpre** to recognize SQL commands in either upper case or lower case. If this switch is omitted SQL commands must be upper case.
-m[anual]	Suppresses automatic generation of transactions. Use this switch for programs that perform their own transaction handling.
-n[o_lines]	Suppresses line number for C programs.
-o[utput]	Directs **gpre** output to standard out rather than a file.
-password password	Specifies the database password.
-r[aw]	Prints BLR as raw numbers rather than as their mnemonic equivalents. This makes **gpre** output smaller but less readable.
-sql_dialect n	Sets the SQL dialect. Valid values are 1, 2 and 3.
-user username	Specifies the database user ID.
-x handle	Specifies the database handle given with the -database option an external declaration. This option directs the program to pick up a global declaration from another program.
-z	Prints **gpre**'s version number and the version number of all declared databases.

Table 8Option Switches

Timestamp Format Strings

It is possible to specify the format that is used to transfer DATE, TIME and TIMESTAMP columns between your application and the database manager. This is done using the **-dfm** option switch. The **-dfm** switch takes a string as its argument. This string contains the format string. The date format is represented in this string using a number of character codes to represent the various parts of a timestamp field. To build a format string just concatenate the appropriate selections from the following table:

Format Code	*Description*
yy	The last two digits of the year.
yyyy	The full four digit year including the century.
mm	The two digit month of the year.
dd	The two digit day of the month.
hh	The two digit hour. This is given in military (24 hour) format.
nn	The two digit minutes.
ss	The two digit seconds.

Table 9Timestamp Format Strings

When using the **-dfm** switch, host variables which are to receive a DATE, TIME or TIMESTAMP field must be large enough to hold the date and time format specified by the format string.

10.2. Running Your Program

In order to execute your RM/Cobol application you must provide the name of Firebird's RM/Cobol interface shared library on the command line. This is done using the **L=** command line switch.

For Windows applications your command would look like this:

```
runcobol myapp.cob L=fbrmclib.dll
```

For Unix applications your command line would look like this:

```
runcobol myapp.cob L=fbrmclib.so
```