Firebird Reference Guide

About IBPhoenix

IBPhoenix is located—everywhere! It is an umbrella company for a network of independent developers around the world who have been involved with the open sourcing of InterBase® 6.0 since the end of 1999, when the proposal was first made by some software engineers who had worked with the code as employees of Borland.

The IBPhoenix team consists of people who have worked with the InterBase® dynasty of databases for many years, in R & D— in the past, as well as currently as members of the independent Firebird project—and as application consultants and engineers.

IBPhoenix exists with a commitment to make Firebird a presence amongst open source database software. It is a great RDBMS which we consider underrated by the market. Besides providing a commercial support network for Firebird and InterBase® users, we are devoted to assisting the user community with documentation and forum-based support. We maintain a bank technical articles and an on-line knowledge database at our website, http://www.ibphoenix.com. We continually seek funding opportunities for the Firebird project with a sincere faith in the synergy that a community of dedicated engineers and involved users can bring to Firebird in open source conditions.

This pair of volumes—**Using Firebird** and the **Firebird Reference Guide**—is a digest of information for software engineers and system administrators from many sources in this wonderful community. We hope it is the first of an ongoing series of publications to educate, enlighten and enlarge the rapidly growing community of Firebird users.

Firebird Reference Guide

Table of Contents

Firebird Reference Guide

CHAPTER 1

isql command reference

Command-line **isql** supports the following special commands:

TABLE 1–1 **isql** commands

BLOBDUMP	SET COUNT	SHOW CHECK	SHOW INDICES
EDIT	SET ECHO	SHOW DATABASE	SHOW PROCEDURES
EXIT	SET LIST	SHOW DOMAINS	SHOW ROLES
HELP	SET NAMES	SHOW EXCEPTIONS	SHOW SQL DIALECT
INPUT	SET PLAN	SHOW FILTERS	SHOW SYSTEM
OUTPUT	SET STATS	SHOW FUNCTIONS	SHOW TABLES
QUIT	SET SQL DIALECT	SHOW GENERATORS	SHOW TRIGGERS
SET	SET TERM	SHOW GRANT	SHOW VERSION
SET AUTODDL	SET TIME	SHOW INDEX	SHOW VIEWS
SET BLOBDISPLAY	SHELL		

BLOBDUMP

Places the contents of a BLOB column in a named file for reading or editing.

Syntax `BLOBDUMP` *`blob_id filename`*`;`

Argument	Description
blob_id	System-assigned hexadecimal identifier, made up of two hexadecimal numbers separated by a colon (**:**) • First number is the ID of the table containing the BLOB column • Second number is a sequential number identifying a particular instance of blob data
filename	Name of the file into which to place blob contents

Description BLOBDUMP stores blob data identified by *blob_id* in the file specified by *filename*. Because binary files cannot be displayed, BLOBDUMP is useful for viewing or editing binary data. BLOBDUMP is also useful for saving blocks of text (blob data) to a file.

To determine the blob_id to supply in the BLOBDUMP statement, issue any SELECT statement that selects a column of blob data. When the table's columns appear, any blob columns contain hexadecimal Blob IDs. The display of blob output can be controlled using SET BLOBDISPLAY.

Example Suppose that Blob ID 58:c59 refers to graphical data in JPEG format. To place this blob data into a graphics file named **picture.jpg**, enter:

```
BLOBDUMP 58:c59 picture.jpg;
```

EDIT

Allows editing and re-execution of **isql** commands.

Syntax `EDIT [`*`filename`*`];`

Argument	Description
filename	Name of the file to edit

Description The EDIT command enables you to edit commands in:

- A source file and then execute the commands upon exiting the editor.
- The current **isql** session, then re-execute them.

On Windows 95/98 and Windows NT/2000, EDIT calls the text editor specified by the EDITOR environment variable.

Setting up edit.com as the default DOS editor

To define the EDITOR environment variable to use the DOS GUI editor, **edit.com**, do as follows:

- On NT, Windows 2000 or Windows XP, log in as Administrator
- Open the Registry Editor program **regedit.exe** and locate the key `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\SessionManager\Environment`
- Right-click on the entry and select **New>String value**
- Rename the entry to EDITOR
- Right-click on the entry and select **Modify**
- Type in the unquoted path to Edit.com on your system. For example, on Windows NT: `%SystemRoot%\system32\edit.com`
- Click OK.
- Now, check the corresponding entries for ControlSet002 and CurrentControlSet to make sure the new value was added.
- Exit from **regedit.exe**.

On Linux/UNIX, EDIT calls the text editor specified by either the VISUAL environment variable or EDITOR, in that order. If neither variable is defined, then EDIT uses the **vi** editor.

If given *`filename`* as an argument, EDIT places the contents of filename in an edit buffer. If no file name is given, EDIT places the commands in the current isql session in the edit buffer.
After exiting the editor, isql automatically executes the commands in the edit buffer.

Filenames with spaces You can optionally delimit the filename with double or single quotes. This allows you to use filenames with spaces in EDIT statements.

Examples To edit the commands in a file called **start.sql** and execute the commands when done, enter:

```
EDIT START.SQL;
```

In the next example, a user wants to enter `SELECT DISTINCT JOB_CODE, JOB_TITLE FROM JOB;` interactively: Instead, the user mistakenly omits the DISTINCT reserved word. Issuing the `EDIT` command opens the statement in an editor and then executes the edited statement when the editor exits.

```
EDIT;
SELECT JOB_CODE, JOB_TITLE FROM JOB;
```

An unnamed buffer will be assigned an arbitrary file name, e.g. IB04009 and will be saved into your temporary directory for the duration of the session. Each time you call EDIT, the buffer will be restored into the editor for any further editing you require in the session.

To save the buffer permanently, choose the Save As.. option (or equivalent in your editor), giving a path and file name for where you want to save the file.

See Also **INPUT**, **OUTPUT**, **SHELL**

EXIT

Commits the current transaction, closes the database, and ends the **isql** session.

Syntax `EXIT;`

Description Both EXIT and QUIT close the database and end an **isql** session. EXIT commits any changes made since the last COMMIT or ROLLBACK, whereas QUIT rolls them back.

EXIT is equivalent to the end-of-file character, which differs across systems.

Important EXIT commits changes without prompting for confirmation. Before using EXIT, be sure that no transactions need to be rolled back.

See Also **QUIT**, **SET AUTODDL**

HELP

Displays a list of ISQL commands and short descriptions.

Syntax `HELP;`

Description HELP lists the built-in **isql** commands, with a brief description of each.

Example To save the help screen to a file named **isqlhelp.lst**, enter:

```
 OUTPUT isqlhelp.lst;
 HELP;
OUTPUT;
```

After issuing the HELP command, use OUTPUT to redirect output back to the screen.

INPUT

Reads and executes commands from the named file.

Syntax `INPUT filename;`

Argument	Description
filename	Name of the file containing SQL statements and SQL commands

Description INPUT reads commands from *filename* and executes them as a **block**. In this way, INPUT enables execution of commands without prompting. *filename* must contain SQL statements or **isql** commands.

Input files can contain their own INPUT commands. Nesting INPUT commands enables **isql** to process multiple files. When **isql** reaches the end of one file, processing returns to the previous file until all commands are executed.

The INPUT command is intended for noninteractive use. Therefore, the EDIT command does not work in input files.

Using INPUT *filename* from within an **isql** session has the same effect as using ***-input*** *filename* from the command line.
Unless output is redirected using OUTPUT, any results returned by executing filename appear on the screen. You can optionally delimit the filename with double or single quotes. This allows you to use filenames with spaces in INPUT statements.

Examples For this example, suppose that file **add.lst** contains the following INSERT statement:

```
INSERT INTO COUNTRY (COUNTRY, CURRENCY)
VALUES ('Mexico', 'Peso');
```

To execute the command stored in **add.lst**, enter:

```
INPUT add.lst;
```

For the next example, suppose that the file, **table.lst**, contains the following SHOW commands:

```
SHOW TABLE COUNTRY;
SHOW TABLE CUSTOMER;
SHOW TABLE DEPARTMENT;
SHOW TABLE EMPLOYEE;
SHOW TABLE EMPLOYEE_PROJECT;
SHOW TABLE JOB;
```

To execute these commands, enter:

```
INPUT table.lst;
```

To record each command and store its results in a file named **table.out**, enter

```
SET ECHO ON;
OUTPUT table.out;
INPUT table.lst;
OUTPUT;
```

See Also OUTPUT

OUTPUT

Redirects output to the named file or to standard output.

Syntax `OUTPUT [filename];`

Argument	Description
filename	Name of the file in which to save output; if no file name is given, results appear on the standard output

Description OUTPUT determines where the results of **isql** commands are displayed. By default, results are displayed on standard output (usually a screen). To store results in a file, supply a *filename* argument. To return to the default mode, again displaying results on the standard output, use OUTPUT without specifying a file name.

By default, only data is redirected. Interactive commands are not redirected unless SET ECHO is in effect. If SET ECHO is in effect, **isql** displays each command before it is executed. In this way, **isql** captures both the results and the command that produced them. SET ECHO is useful for displaying the text of a query immediately before the results.

Note Error messages cannot be redirected to an output file.

Using OUTPUT *filename* from within an **isql** session has the same effect as using the option **-output** *filename* from the command line.

You can optionally delimit the filename with double or single quotes. This allows you to use filenames with spaces in OUTPUT statements.

Example The following example stores the results of one SELECT statement in the file, **sales.out**. Normal output processing resumes after the SELECT statement.

```
OUTPUT sales.out;
SELECT * FROM SALES;
OUTPUT;
```

See Also INPUT, SET ECHO

QUIT

Rolls back the current transaction, closes the database, and ends the **isql** session.

Syntax `QUIT;`

Description Both EXIT and QUIT close the database and end an **isql** session. QUIT rolls back any changes made since the last COMMIT or ROLLBACK, whereas EXIT commits the changes.

Important QUIT rolls back uncommitted changes without prompting for confirmation. Before using QUIT, be sure that any changes that need to be committed are committed. For example, if SET AUTODDL is off, DDL statements must be committed explicitly.

See Also **EXIT**, **SET AUTODDL**

SET

Lists the status of the features that control an **isql** session.

Syntax `SET;`

Description **isql** provides several SET commands for specifying how data is displayed or how other commands are processed.

The SET command, by itself, verifies which features are currently set. Some SET commands turn a feature on or off. Other SET commands assign values.

Many **isql** SET commands have corresponding SQL statements that provide similar or identical functionality. In addition, some of the **isql** features controlled by SET commands can also be controlled using **isql** command-line options. SET Statements are used to configure the **isql** environment from a script file. Changes to the session setting from SET statements in a script affect the session only while the script is running. After a script completes, the session settings prior to running the script are restored.

The **isql** SET statements are:

TABLE 1–2 SET statements

Statement	Description	Default
SET AUTODDL	Toggles the commit feature for DDL statements	ON
SET BLOBDISPLAY *n*	Turns on the display of blob type *n*; the parameter *n* is required to display blob types	OFF
SET COUNT	Toggles the count of selected rows on or off	OFF
SET ECHO	Toggles the display of each command on or off	OFF
SET LIST *string*	Displays columns vertically or horizontally	OFF
SET NAMES	Specifies the active character set	OFF
SET PLAN	Specifies whether or not to display the optimizer's query plan	OFF
SET STATS	Toggles the display of performance statistics on or off	OFF
SET TERM *string*	Allows you to change to an alternate terminator character	;
SET TIME	Toggles display of time in DATE values	ON

By default, all settings are initially OFF except AUTODDL and TIME, and the terminator is a semicolon (**;**). Each time you start an **isql** session or execute an **isql** script file, settings begin with their default values.
SET statements are used to configure the **isql** environment from a script file. Changes to the session setting from SET statements in a script affect the session only while the script is running. After a script completes, the session settings prior to running the script are restored to their values before the script was run. So you can modify the settings for interactive use, then change them as needed in an **isql** script, and after running the script they automatically return to their previous configuration.

Example To display the **isql** features currently in effect, enter:

```
SQL> SET;
Print statistics:OFF
Echo commands:OFF
List format:OFF
Row count:OFF
Autocommit DDL:OFF
Access plan:OFF
Display BLOB type:1
Terminator:;
Time:OFF
```

The output shows that **isql** is set to not echo commands, to display blob data if they are of subtype 1 (text), to automatically commit DDL statements, and to recognize a semicolon (**;**) as the statement termination character.

Other SET statements

- SET GENERATOR and SET TRANSACTION (without a transaction name) are not included in this section, as they are DDL statements, not confined exclusively to **isql**. They can be entered interactively in **isql** or your interactive query tool of choice. See SET GENERATOR and SET TRANSACTION in chapter 2, SQL Statement and Function Reference.
- SET DATABASE is exclusively an embedded SQL statement—it cannot be used interactively or in dynamic SQL. See SET DATABASE.

SET AUTODDL

Specifies whether DDL statements are committed automatically after being executed or committed only after an explicit COMMIT.

Syntax `SET AUTO[DDL] [ON | OFF];`

Argument	Description
ON	Turns on automatic commitment of DDL [default]
OFF	Turns off automatic commitment of DDL

Description SET AUTODDL is used to turn on or off the automatic commitment of data definition language (DDL) statements. By default, DDL statements are automatically committed immediately after they are executed, in a separate transaction. This is the recommended behavior.

If the OFF reserved word is specified, auto-commit of DDL is then turned off. In OFF mode, DDL statements can only be committed explicitly through a user's transaction. This mode is useful for database prototyping, because uncommitted changes are easily undone by rolling them back.
SET AUTODDL has a shorthand equivalent, SET AUTO.

Tip The ON and OFF reserved words are optional. If they are omitted, SET AUTO switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Examples The following example shows part of an **isql** script that turns off AUTODDL, creates a table named TEMP, then rolls back the work.

```
. . .
SET AUTO OFF;
CREATE TABLE TEMP (a INT, b INT);
ROLLBACK;
. . .
```

This script creates TEMP and then rolls back the statement. No table is created. because its creation was rolled back.
The next script uses the default AUTODDL ON. It creates the table TEMP and then performs a rollback:

```
    . . .
    CREATE TABLE TEMP (a INT, b INT);
    ROLLBACK;
. . .
```

Because DDL is automatically committed, the rollback does not affect the creation of TEMP.

See Also EXIT, QUIT

SET BLOBDISPLAY

Specifies subtype of blob data to display.

Syntax `SET BLOBDISPLAY [n | ALL | OFF];`

Argument	Description
n	Integer specifying the blob subtype to display • Use 0 for blob data of an unknown subtype • Use 1 for blob data of a text subtype [default] • Use other integer values for other subtypes
ALL	Displays blob data of all subtypes
OFF	Turns off display of blob data of all subtypes

Description SET BLOBDISPLAY has the following uses:

- To display blob data of a particular subtype, use SET BLOBDISPLAY *n*. By default, **isql** displays blob data of text subtype (*n* = 1).
- To display blob data of all subtypes, use SET BLOBDISPLAY ALL.
- To avoid displaying blob data, use SET BLOBDISPLAY OFF. Omitting the OFF reserved word has the same effect. Turn blob display off to make output easier to read.

In any column containing blob data, the actual data does not appear in the column. Instead, the column displays a Blob ID that represents the data. If SET BLOBDISPLAY is on, data associated with a Blob ID appears under the row containing the Blob ID. If SET BLOBDISPLAY is off, the Blob ID still appears even though its associated data does not.
SET BLOBDISPLAY has a shorthand equivalent, SET BLOB.
To determine the subtype of a BLOB column, use SHOW TABLE.

Examples The following examples show output from the same SELECT statement. Each example uses a different SET BLOB command to affect how output appears. The first example turns off blob display.

```
 SET BLOB OFF;
SELECT PROJ_NAME, PROJ_DESC FROM PROJECT;
```

With BLOBDISPLAY OFF, the output shows only the Blob ID:

```
PROJ_NAME PROJ_DESC
========== ==========
Video Database 24:6
DigiPizza 24:8
AutoMap 24:a
MapBrowser port 24:c
Translator upgrade 24:3b
Marketing project 3   24:3d
```

The next example restores the default by setting BLOBDISPLAY to subtype 1 (text).

```
 SET BLOB 1;
SELECT PROJ_NAME, PROJ_DESC FROM PROJECT;
```

Now the contents of the blob appear below each Blob ID:

```
PROJ_NAME PROJ_DESC
===================================
Video Database 24:6
============================================================
PROJ_DESC:
Design a video data base management system for
controlling on-demand video distribution.
PROJ_NAME PROJ_DESC
===================================
DigiPizza 24:8
============================================================
PROJ_DESC:
Develop second generation digital pizza maker
with flash-bake heating element and
digital ingredient measuring system.
. . .
```

See Also BLOBDUMP

SET COUNT

Specifies whether to display number of rows retrieved by queries.

Syntax `SET COUNT [ON | OFF];`

Argument	Description
ON	Turns on display of the “rows returned” message
OFF	Turns off display of the “rows returned” message [default]

Description By default, when a SELECT statement retrieves rows from a query, no message appears to say how many rows were retrieved.

Use SET COUNT ON to change the default behavior and display the message. To restore the default behavior, use SET COUNT OFF.

Tip The ON and OFF reserved words are optional. If they are omitted, SET COUNT switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Example The following example sets COUNT ON to display the number of rows returned by all following queries:

```
SET COUNT ON;
SELECT * FROM COUNTRY
  WHERE CURRENCY LIKE '%FRANC%';
```

The output displayed would then be:

```
COUNTRY CURRENCY
=============== ==========
SWITZERLAND SFRANC
FRANCE FFRANC
BELGIUM BFRANC
3 rows returned
```

SET ECHO

Specifies whether commands are displayed to the **isql** Output area before being executed.

Syntax `SET ECHO [ON | OFF];`

Argument	Description
ON	Turns on command echoing [default]
OFF	Turns off command echoing

Description By default, commands in script files are displayed (echoed) in the **isql** Output area, before being executed. Use SET ECHO OFF to change the default behavior and suppress echoing of commands. This can be useful when sending the output of a script to a file, if you want only the results of the script and not the statements themselves in the output file.

Command echoing is useful if you want to see the commands as well as the results in the **isql** Output area.

Tip The ON and OFF reserved words are optional. If they are omitted, SET ECHO switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Example Suppose you execute the following script from IBConsole ISQL:

```
 . . .
 SET ECHO OFF;
 SELECT * FROM COUNTRY;
 SET ECHO ON;
 SELECT * FROM COUNTRY;
EXIT;
```

The output (in a file or the **isql** Output area) looks like this:

```
    . . .
    SET ECHO OFF;
    COUNTRYCURRENCY
    ==================
    USA Dollar
    EnglandPound
    . . .
    SELECT * FROM COUNTRY;
    COUNTRYCURRENCY
    ==================
    USA Dollar
    EnglandPound
  . . .
```

The first SELECT statement is not displayed, because ECHO is OFF. Notice also that the SET ECHO ON statement itself is not displayed, because when it is executed, ECHO is still OFF. After it is executed, however, the second SELECT statement is displayed.

See Also INPUT, OUTPUT

SET LIST

Specifies whether output appears in tabular format or in list format.

Syntax `SET LIST [ON | OFF];`

Argument	Description
ON	Turns on list format for display of output
OFF	Turns off list format for display of output [default]

Description By default, when a SELECT statement retrieves rows from a query, the output appears in a tabular format, with data organized in rows and columns.

Use SET LIST ON to change the default behavior and display output in a list format. In list format, data appears one value per line, with column headings appearing as labels. List format is useful when columnar output is too wide to fit nicely on the screen.

Tip The ON and OFF reserved words are optional. If they are omitted, SET LIST switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Example Suppose you execute the following statement in a script file:

```
SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY, JOB_TITLE FROM JOB
  WHERE JOB_COUNTRY = 'Italy';
```

The output is:

```
JOB_CODE   JOB_GRADE   JOB_COUNTRY   JOB_TITLE
========   =========   ===========   ====================
SRep       4           Italy         Sales Representative
```

Now suppose you precede the SELECT with SET LIST ON:

```
SET LIST ON;
SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY, JOB_TITLE FROM JOB
  WHERE JOB_COUNTRY = 'Italy';
```

The output is:

```
JOB_CODE SRep
JOB_GRADE 4
JOB_COUNTRY Italy
JOB_TITLE Sales Representative
```

SET NAMES

Specifies the active character set to use in database transactions.

Syntax `SET NAMES [charset];`

Argument	Description
charset	Name of the active character set; default is NONE

Description SET NAMES specifies the character set to use for subsequent database connections in **isql**. It enables you to override the default character set for a database. To return to using the default character set, use SET NAMES with no argument.

Use SET NAMES before connecting to the database whose character set you want to specify. For a complete list of character sets recognized by Firebird, see the *Language Reference*.
Choice of character set limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

Example The following statement at the beginning of a script file indicates to set the active character set to ISO8859_1 for the subsequent database connection:

```
SET NAMES ISO8859_1;
CONNECT 'jupiter:/usr/interbase/examples/employee.gdb';
. . .
```

SET PLAN

Specifies whether to display the optimizer's query plan.

Syntax `SET PLAN [ON | OFF];`

Argument	Description
ON	Turns on display of the optimizer's query plan
OFF	Turns off display of the optimizer's query plan [default]

Description By default, when a SELECT statement retrieves rows from a query, **isql** does not display the query plan used to retrieve the data.

Use SET PLAN ON to change the default behavior and display the query optimizer plan. To restore the default behavior, use SET PLAN OFF.
To change the query optimizer plan, use the PLAN clause in the SELECT statement.

Tip The ON and OFF reserved words are optional. If they are omitted, SET PLAN switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Example The following example shows part of a script that sets PLAN ON:

```
SET PLAN ON;
SELECT JOB_COUNTRY, MIN_SALARY FROM JOB
WHERE MIN_SALARY > 50000
      AND JOB_COUNTRY = 'France';
```

The output then includes the query optimizer plan used to retrieve the data as well as the results of the query:

```
PLAN (JOB INDEX (RDB$FOREIGN3,MINSALX,MAXSALX))
JOB_COUNTRYMIN_SALARY
=============== ======================
France 118200.00
```

SET PLANONLY

specifies to use the optimizer's query plan and display just the plan, without executing the actual query. (Available in Firebird 1 and higher).

Syntax

```
SQL> SET PLANONLY ;
```

No arguments: the command works as a toggle switch.

SET STATS

Specifies whether to display performance statistics after the results of a query.

Syntax `SET STATS [ON | OFF];`

Argument	Description
ON	Turns on display of performance statistics
OFF	Turns off display of performance statistics [default]

Description By default, when a SELECT statement retrieves rows from a query, **isql** does not display performance statistics after the results. Use SET STATS ON to change the default behavior and display performance statistics. To restore the default behavior, use SET STATS OFF. Performance statistics include:

- Current memory available, in bytes
- Change in available memory, in bytes
- Maximum memory available, in bytes

- Elapsed time for the operation
- CPU time for the operation
- Number of cache buffers used
- Number of reads requested
- Number of writes requested
- Number of fetches made

Performance statistics can help determine if changes are needed in system resources, database resources, or query optimization.

Tip The ON and OFF reserved words are optional. If they are omitted, SET STATS switches from one mode to the other. Although you can save typing by omitting the optional reserved word, including the reserved word is recommended because it avoids potential confusion.

Do not confuse SET STATS with the SQL statement SET STATISTICS, which recalculates the selectivity of an index.

Example The following part of a script file turns on display of statistics and then performs a query:

```
SET STATS ON;
SELECT JOB_COUNTRY, MIN_SALARY FROM JOB
WHERE MIN_SALARY > 50000
      AND JOB_COUNTRY = 'France';
```

The output displays the results of the SELECT statement and the performance statistics for the operation:

```
JOB_COUNTRYMIN_SALARY
============== ====================
France 118200.00

Current memory = 407552
Delta memory = 0
Max memory = 412672
Elapsed time= 0.49 sec
Cpu = 0.06 sec
Buffers = 75
Reads = 3
Writes = 2
Fetches = 441
```

See Also SHOW DATABASE

SET TERM

Specifies which character or characters signal the end of a command.

Syntax `SET TERM` *`string`*`;`

Argument	Description
string	Specifies a character or characters to use in terminating a statement; default is semicolon (**;**)

Description By default, when a line ends with a semicolon, **isql** interprets it as the end of a command. Use SET TERM to change the default behavior and define a new termination character.

SET TERM is typically used with CREATE PROCEDURE or CREATE TRIGGER. Procedures and triggers are defined using a special “procedure and trigger language” in which statements end with a semicolon. If **isql** were to

interpret semicolons as statement terminators, then procedures and triggers would execute during their creation, rather than when they are called.

A script file containing CREATE PROCEDURE or CREATE TRIGGER definitions should include one SET TERM command before the definitions and a corresponding SET TERM after the definitions. The beginning SET TERM defines a new termination character; the ending SET TERM restores the semicolon (;) as the default.

Example The following example shows a text file that uses SET TERM in creating a procedure. The first SET TERM defines "##" as the termination characters. The matching SET TERM restores ";" as the termination character.

```
SET TERM ## ;
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
BEGIN
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
VALUES (:emp_no, :proj_id);
WHEN SQLCODE -530 DO
EXCEPTION UNKNOWN_EMP_ID;
END
RETURN;
END ##
SET TERM ; ##
```

SET TIME

Specifies whether to display the time portion of a dialect 1 DATE value—it has no meaning with respect to dialect 3 DATE type and has no effect on Timestamp columns.

Syntax `SET TIME [ON | OFF];`

Argument	Description
ON	Turns on display of time in DATE value
OFF	Turns off display of time in DATE value [default]

Description The dialect 1 DATE data type includes a date portion (including day, month, and year) and a time portion (including hours, minutes, and seconds).

By default, **isql** displays only the date portion of dialect 1 Date values. SET TIME ON turns on the display of time values. SET TIME OFF turns off the display of time values.

Tip The ON and OFF reserved words are optional. If they are omitted, the command toggles time display from ON to OFF or OFF to ON.

Example The following example shows the default display of a DATE datatype, which is to display day, month, and year:

```
SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;
HIRE_DATE
--------------------
2-MAY-1994
```

This example shows the effects of SET TIME ON, which causes the hours, minutes and seconds to be displayed as well:

```
SET TIME ON;
SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;
HIRE_DATE
--------------------
2-MAY-1994 12:25:00
```

SHELL

Allows execution of an operating system command or temporary access to an operating system shell.

Syntax `SHELL [<os_command>];`

Argument	Description
os_command	An operating system command; if no command is specified, **isql** provides interactive access to the operating system

Description The SHELL command provides temporary access to operating system commands in an **isql** session. Use SHELL to execute an operating-system command without ending the current **isql** session.

If *os_command* is specified, the operating system executes the command and then returns to **isql** when complete.

If no command is specified, an operating system shell prompt appears, enabling you to execute a sequence of commands. To return to **isql**, type `exit`. For example, SHELL can be used to edit an input file and run it at a later time. By contrast, if an input file is edited using the EDIT command, the input file is executed as soon as the editing session ends.

Using SHELL does not commit transactions before it calls the shell.

Example The following example uses SHELL to display the contents of the current directory:

```
SHELL DIR;
```

See Also EDIT

SHOW CHECK

Displays all CHECK constraints defined for a specified table.

Syntax `SHOW CHECK table;`

Argument	Description
table	Name of an existing table in the current database

Description SHOW CHECK displays CHECK constraints for a named table in the current database. Only user-defined metadata is displayed. To see a list of existing tables, use SHOW TABLE.

Example The following example shows CHECK constraints defined for the JOB table. The SHOW TABLES command is used first to display a list of available tables.

```
SHOW TABLES;
COUNTRYCUSTOMER
DEPARTMENTEMPLOYEE
EMPLOYEE_PROJECTJOB
PHONE_LISTPROJECT
PROJ_DEPT_BUDGETSALARY_HISTORY
SALES

SHOW CHECK JOB;
  CHECK (min_salary < max_salary)
```

See Also SHOW TABLES

SHOW DATABASE

Displays information about the current database.

Syntax `SHOW [DATABASE | DB];`

Description SHOW DATABASE displays the current database's file name, page size and allocation, and sweep interval.

The output of SHOW DATABASE is used to verify data definition or to administer the database. For example, use the backup and restore utilities to change page size or reallocate pages among multiple files, and use the database maintenance utility to change the sweep interval.
SHOW DATABASE has a shorthand equivalent, SHOW DB.

Example The following example connects to a database and displays information about it:

```
CONNECT 'employee.gdb';
Database: employee.gdb

SHOW DB;
Database: employee.gdb
Owner: SYSDBA
PAGE_SIZE 4096
Number of DB pages allocated = 422
  Sweep interval = 20000
```

SHOW DOMAINS

Lists all domains or displays information about a specified domain.

Syntax `SHOW {DOMAINS | DOMAIN` *`name`*`};`

Argument	Description
name	Name of an existing domain in the current database

Options To see a list of existing domains, use SHOW DOMAINS without specifying a domain name. SHOW DOMAIN name displays information about the named domain in the current database. Output includes a domain's datatype, default value, and any CHECK constraints defined. Only user-defined metadata is displayed.

Example The following example lists all domains and then shows the definition of the domain, SALARY:

```
SHOW DOMAINS;
FIRSTNAME LASTNAME
PHONENUMBER COUNTRYNAME
ADDRESSLINE EMPNO
DEPTNO PROJNO
CUSTNO JOBCODE
JOBGRADE SALARY
BUDGET PRODTYPE
PONUMBER

SHOW DOMAIN SALARY;
SALARY NUMERIC(15, 2) Nullable
DEFAULT 0
                  CHECK (VALUE > 0)
```

SHOW EXCEPTIONS

Lists all exceptions or displays the text of a specified exception.

Syntax `SHOW {EXCEPTIONS | EXCEPTION` *`name`*`};`

Argument	Description
name	Name of an existing exception in the current database

Description SHOW EXCEPTIONS displays an alphabetical list of exceptions. SHOW EXCEPTION name displays the text of the named exception.

Examples To list all exceptions defined for the current database, enter:

```
SHOW EXCEPTIONS;
Exception Name Used by, Type
================= ======================================
UNKNOWN_EMP_ID ADD_EMP_PROJ, Stored procedure
Invalid employee number or project ID.
. . .
```

To list the message for a specific exception and the procedures or triggers that use it, enter the exception name:

```
SHOW EXCEPTION CUSTOMER_CHECK;
Exception Name                    Used by, Type
==========================  ====================================
CUSTOMER_CHECK               SHIP_ORDER, Stored procedure
 Overdue balance -- can't ship.
```

SHOW FILTERS

Lists all blob filters or displays information about a specified filter.

Syntax `SHOW {FILTERS | FILTER` *name*`};`

Argument	Description
name	Name of an existing blob filter in the current database

Options To see a list of existing filters, use SHOW FILTERS. SHOW FILTER name displays information about the named filter in the current database. Output includes information previously defined by the DECLARE FILTER statement, the input subtype, output subtype, module (or library) name, and entry point name.

Example The following example lists all filters and then shows the definition of the filter, DESC_FILTER:

```
SHOW FILTERS;
DESC_FILTER

SHOW FILTER DESC_FILTER;
BLOB Filter: DESC_FILTER
Input subtype: 1 Output subtype -4
Filter library is: desc_filter
  Entry point is: FILTERLIB
```

SHOW FUNCTIONS

Lists all user-defined functions (UDFs) defined in the database or displays information about a specified UDF.

Syntax `SHOW {FUNCTIONS | FUNCTION` *`name`*`};`

Argument	Description
name	Name of an existing UDF in the current database

Options To see a list of existing functions defined in the database, use SHOW FUNCTIONS. SHOW FUNCTION name displays information about the named function in the current database. Output includes information previously defined by the DECLARE EXTERNAL FUNCTION statement: the name of the function and function library, the name of the entry point, and the datatypes of return values and input arguments.

Example The following example lists all UDFs and then shows the definition of the MAXNUM() function:

```
SHOW FUNCTIONS;
ABS MAXNUM
TIME UPPER_NON_C
UPPER

SHOW FUNCTION maxnum;
Function MAXNUM:
Function library is /usr/firebird/lib/gdsfunc.so
Entry point is FN_MAX
Returns BY VALUE DOUBLE PRECISION
Argument 1: DOUBLE PRECISION
  Argument 2: DOUBLE PRECISION
```

SHOW GENERATORS

Lists all generators or displays information about a specified generator.

Syntax `SHOW {GENERATORS | GENERATOR` *`name`*`};`

Argument	Description
name	Name of an existing generator in the current database

Description To see a list of existing generators, use SHOW GENERATORS. SHOW GENERATOR name displays information about the named generator in the current database. Output includes the name of the generator and its next value.

SHOW GENERATOR has a shorthand equivalent, SHOW GEN.

Example The following example lists all generators and then shows information about EMP_NO_GEN:

```
SHOW GENERATORS;
Generator EMP_NO_GEN, Next value: 146
Generator CUST_NO_GEN, Next value: 1016

SHOW GENERATOR EMP_NO_GEN;
  Generator EMP_NO_GEN, Next value: 146
```

SHOW GRANT

Displays privileges for a database object.

Syntax `SHOW GRANT` *`object`*`;`

Argument	Description
object	Name of an existing table, view, or procedure in the current database

Description SHOW GRANT displays the privileges defined for a specified table, view, or procedure. Allowed privileges are DELETE, EXECUTE, INSERT, SELECT, UPDATE, or ALL. To change privileges, use the SQL statements GRANT or REVOKE.

Before using SHOW GRANT, you might want to list the available database objects. Use SHOW PROCEDURES to list existing procedures; use SHOW TABLES to list existing tables; use SHOW VIEWS to list existing views.

Example To display GRANT privileges on the JOB table, enter:

```
SHOW GRANT JOB;
GRANT SELECT ON JOB TO ALL
  GRANT DELETE, INSERT, SELECT, UPDATE ON JOB TO MANAGER
```

SHOW GRANT can also show role membership:

```
SHOW GRANT DOITALL;
  GRANT DOITALL TO SOCKS
```

See Also SHOW PROCEDURES, SHOW TABLES, SHOW VIEWS

SHOW INDEX

Displays index information for a specified index, for a specified table, or for all tables in the current database.

Syntax `SHOW {INDICES | INDEX {`*`index`* `|` *`table`*`} };`

Argument	Description
index	Name of an existing index in the current database
table	Name of an existing table in the current database

Description SHOW INDEX displays the index name, the index type (for example, UNIQUE or DESC), and the columns on which an index is defined.

If the index argument is specified, SHOW INDEX displays information only for that index. If table is specified, SHOW INDEX displays information for all indexes in the named table; to display existing tables, use SHOW TABLES. If no argument is specified, SHOW INDEX displays information for all indexes in the current database. SHOW INDEX has a shorthand equivalent, SHOW IND. SHOW INDICES is also a synonym for SHOW INDEX. SHOW INDEXES is not supported.

Examples To display indexes for database **employee.gdb**, enter:

```
SHOW INDEX;
RDB$PRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
CUSTREGION INDEX ON CUSTOMER(COUNTRY, CITY)
RDB$FOREIGN23 INDEX ON CUSTOMER(COUNTRY)
. . .
```

To display index information for the SALES table, enter:

```
SHOW IND SALES;
NEEDX INDEX ON SALES(DATE_NEEDED)
QTYX DESCENDING INDEX ON SALES(ITEM_TYPE, QTY_ORDERED)
RDB$FOREIGN25 INDEX ON SALES(CUST_NO)
RDB$FOREIGN26 INDEX ON SALES(SALES_REP)
RDB$PRIMARY24 UNIQUE INDEX ON SALES(PO_NUMBER)
  SALESTATX INDEX ON SALES(ORDER_STATUS, PAID)
```

See Also SHOW TABLES

SHOW PROCEDURES

Lists all procedures or displays the text of a specified procedure.

Syntax `SHOW {PROCEDURES | PROCEDURE` *`name`*`};`

Argument	Description
name	Name of an existing procedure in the current database

Description SHOW PROCEDURES displays an alphabetical list of procedures, along with the database objects they depend on. Deleting a database object that has a dependent procedure is not allowed. To avoid an **isql** error, delete the procedure (using DROP PROCEDURE) before deleting the database object.

SHOW PROCEDURE name displays the text and parameters of the named procedure.
SHOW PROCEDURE has a shorthand equivalent, SHOW PROC.

Examples To list all procedures defined for the current database, enter:

```
SHOW PROCEDURES;
```

```
Procedure Name Dependency Type
================ ================== =======
ADD_EMP_PROJ EMPLOYEE_PROJECTTable
 UNKNOWN_EMP_IDException
DELETE_EMPLOYEE DEPARTMENTTable
EMPLOYEETable
EMPLOYEE_PROJECTTable
PROJECTTable
REASSIGN_SALESException
SALARY_HISTORYTable
SALESTable
DEPT_BUDGET DEPARTMENTTable
DEPT_BUDGETProcedure
. . .
```

To display the text of the procedure, ADD_EMP_PROJ, enter:

```
SHOW PROC ADD_EMP_PROJ;
```

```
Procedure text:
========================================================================

BEGIN
BEGIN
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES
(:emp_no,:proj_id);
WHEN SQLCODE -530 DO
EXCEPTION UNKNOWN_EMP_ID;
END
RETURN;
END
========================================================================
Parameters:
EMP_NO INPUT SMALLINT
  PROJ_ID INPUT CHAR(5)
```

SHOW ROLES

Displays the names of SQL roles for the current database.

Syntax `SHOW {ROLES | ROLE}`

Description SHOW ROLES displays the names of all roles defined for the current database. To show user membership in roles, use SHOW GRANT *rolename*.

Example `SHOW ROLES;`

```
DOITALLDONOTHING
  DOONETHING   DOSOMETHING
```

See Also SHOW GRANT

SHOW SYSTEM

Displays the names of system tables and system views for the current database.

Syntax `SHOW SYSTEM [TABLES];`

Description SHOW SYSTEM lists system tables and system views in the current database. SHOW SYSTEM accepts an optional reserved word, TABLES, which does not affect the behavior of the command.

SHOW SYSTEM has a shorthand equivalent, SHOW SYS.

Example To list system tables and system views for the current database, enter:

```
SHOW SYS;
RDB$CHARACTER_SETS RDB$CHECK_CONSTRAINTS
RDB$COLLATIONS RDB$DATABASE
RDB$DEPENDENCIES RDB$EXCEPTIONS
RDB$FIELDS RDB$FIELD_DIMENSIONS
RDB$FILES RDB$FILTERS
RDB$FORMATS RDB$FUNCTIONS
RDB$FUNCTION_ARGUMENTS RDB$GENERATORS
RDB$INDEX_SEGMENTS RDB$INDICES
RDB$LOG_FILES RDB$PAGES
RDB$PROCEDURES RDB$PROCEDURE_PARAMETERS
RDB$REF_CONSTRAINTS RDB$RELATIONS
RDB$RELATION_CONSTRAINTS RDB$RELATION_FIELDS
RDB$ROLESRDB$SECURITY_CLASSES
RDB$TRANSACTIONS RDB$TRIGGERS
RDB$TRIGGER_MESSAGESRDB$TYPES
RDB$USER_PRIVILEGES RDB$VIEW_RELATIONS
```

See Also For more information about system tables, see the *Language Reference*.

SHOW TABLES

Lists all tables or views, or displays information about a specified table or view.

Syntax `SHOW {TABLES | TABLE name};`

Argument	Description
name	Name of an existing table or view in the current database

Description SHOW TABLES displays an alphabetical list of tables and views in the current database. To determine which listed objects are views rather than tables, use SHOW VIEWS.

SHOW TABLE name displays information about the named object. If the object is a table, command output lists column names and definitions, PRIMARY KEY, FOREIGN KEY, and CHECK constraints, and triggers. If the object is a view, command output lists column names and definitions, as well as the SELECT statement that the view is based on.

Examples To list all tables or views defined for the current database, enter:

```
SHOW TABLES;
COUNTRY CUSTOMER
DEPARTMENT EMPLOYEE
EMPLOYEE_PROJECT JOB
PHONE_LIST PROJECT
PROJ_DEPT_BUDGET SALARY_HISTORY
  SALES
```

To show the definition for the COUNTRY table, enter:

```
SHOW TABLE COUNTRY;
COUNTRY (COUNTRYNAME) VARCHAR(15) NOT NULL
CURRENCY VARCHAR(10) NOT NULL
  PRIMARY KEY (COUNTRY)
```

See Also SHOW VIEWS

SHOW TRIGGERS

Lists all triggers or displays information about a specified trigger.

Syntax `SHOW {TRIGGERS | TRIGGER` *`name`*`};`

Argument	Description
name	Name of an existing trigger in the current database

Description SHOW TRIGGERS displays all triggers defined in the database, along with the table they depend on. SHOW TRIGGER name displays the name, sequence, type, activation status, and definition of the named trigger.

SHOW TRIGGER has a shorthand equivalent, SHOW TRIG.
Deleting a table that has a dependent trigger is not allowed. To avoid an **isql** error, delete the trigger (using DROP TRIGGER) before deleting the table.

Examples To list all triggers defined for the current database, enter:

```
SHOW TRIGGERS;
Table name Trigger name
========== ============
EMPLOYEE SET_EMP_NO
EMPLOYEE SAVE_SALARY_CHANGE
CUSTOMER SET_CUST_NO
  SALES           POST_NEW_ORDER
```

To display information about the SET_CUST_NO trigger, enter:

```
SHOW TRIG SET_CUST_NO;

Triggers:
SET_CUST_NO, Sequence: 0, Type: BEFORE INSERT, Active
AS
BEGIN
new.cust_no = gen_id(cust_no_gen, 1);
  END
```

SHOW VERSION

Displays information about software versions.

Syntax `SHOW VERSION;`

Description SHOW VERSION displays the software version of **isql**, the Firebird engine, and the on-disk structure (ODS) of the database to which the session is attached.

Certain tasks might not work as expected if performed on databases that were created using older InterBase versions. To check the versions of software that are running, use SHOW VERSION.
SHOW VERSION has a shorthand equivalent, SHOW VER.

Example To display software versions, enter:

```
SHOW VER;
ISQL Version: WI-V1.0.0.794 Firebird 1.0
```

See Also SHOW DATABASE

SHOW VIEWS

Lists all views or displays information about a specified view.

Syntax `SHOW {VIEWS | VIEW` *`name`*`};`

Argument	Description
name	Name of an existing view in the current database

Description SHOW VIEWS displays an alphabetical list of all views in the current database. SHOW VIEW name displays information about the named view.

Example To list all views defined for the current database, enter:

```
SHOW VIEWS;
PHONE_LIST
```

See Also SHOW TABLES

CHAPTER 2

SQL Statement and Function Reference

This topic presents the Firebird SQL statements and functions in alphabetical order, showing availability, syntax, and usage for each, and including examples.
Most statements can be used in ESQL, DSQL, and ISQL. In many cases, the syntax is nearly identical, except that ESQL statements must always be preceded by the EXEC SQL reserved words. EXEC SQL is omitted from syntax statements for clarity.
In other cases there are small, but significant differences among ESQL, DSQL, and ISQL syntax. In these cases, separate syntax statements appear under the statement heading. Only a few of the statements are available in PSQL.

List of statements and functions

DECLARE CURSOR
DECLARE CURSOR (BLOB)
DECLARE EXTERNAL FUNCTION
DECLARE FILTER
DECLARE STATEMENT
DECLARE TABLE
DELETE
DESCRIBE
DISCONNECT
DROP DATABASE
DROP DOMAIN
DROP EXCEPTION
DROP EXTERNAL FUNCTION
DROP FILTER
DROP GENERATOR
DROP INDEX
DROP PROCEDURE
DROP ROLE
DROP SHADOW
DROP TABLE
DROP TRIGGER
DROP VIEW
END DECLARE SECTION
EVENT INIT
EVENT WAIT
EXECUTE
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
EXTRACT( )
FETCH
FETCH (BLOB)
FIRST(m) SKIP(n)
GEN_ID( )
GRANT
INSERT
INSERT CURSOR (BLOB)
MAX( )
MIN( )
OPEN
OPEN (BLOB)
PREPARE
RECREATE PROCEDURE
RECREATE TABLE
REVOKE
ROLLBACK
SELECT
SET DATABASE
SET GENERATOR
SET NAMES
SET SQL DIALECT
SET STATISTICS
SET TRANSACTION
SHOW SQL DIALECT
SUBSTRING( )
SUM( )
UPDATE
UPPER( )
WHENEVER
BEGIN … END
Comment

DECLARE VARIABLE | EXCEPTION | EXECUTE PROCEDURE
EXIT | FOR SELECT…INTO…DO | IF…THEN … ELSE
Input parameters | NEW context variables | OLD context variables
Output parameters | POST_EVENT | SELECT
SUSPEND | WHEN … DO | WHILE … DO

ALTER DATABASE

Adds secondary files to the current database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax
```
ALTER {DATABASE | SCHEMA}
        ADD <add_clause>;

<add_clause> = FILE 'filespec' [<fileinfo>] [<add_clause>]

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
SCHEMA	Alternative reserved word for DATABASE
ADD FILE '*filespec*'	Adds one or more secondary files to receive database pages after the primary file is filled; for a remote database, associate secondary files with the same node
LENGTH [=] *int* [PAGE[S]]	Specifies the range of pages for a secondary file by providing the number of pages in each file

Argument	Description
STARTING [AT [PAGE]] *int*	Specifies a range of pages for a secondary file by providing the starting page number

Description ALTER DATABASE adds secondary files to an existing database. Secondary files permit databases to spread across storage devices, but they must remain on the same node as the primary database file. A database can be altered by its creator, the SYSDBA user. On Linux/UNIX, the root user or one with root privileges may also alter a database..

ALTER DATABASE requires exclusive access to the database.
Firebird dynamically expands the last file in a database as needed until it reaches the operating system size limit for shared access files.

Note Be aware that specifying a LENGTH for such files has no effect.

The limits are as follows:

TABLE 2–1 Operating system file size limits

Operating system	Filesystem	Max.size of shared access file
Linux	Older kernels: ext2	2 Gb
(varies according to kernel)	Kernel 2.2.10	4 Gb
	ext2/ext3	16 Gb—currently not reliable
Windows 32	FAT32	2 Gb
	NTFS	16 Gb

You cannot use ALTER DATABASE to split an existing database file. For example, if your existing database is 80,000 pages long and you add a secondary file STARTING AT 50000, Firebird starts the new database file at page 80,001.

Tip To split an existing database file into smaller files, back it up and restore it. When you restore a database, you are free to specify secondary file sizes at will, without reference to the number and size of the original files.

Additional files are very important for accommodating the potential growth of the database. It is prudent to plan for growth and add new files in anticipation.

Example The following ISQL statement adds two secondary files to an existing database. The command creates a secondary database file called **employee2.gdb** that is 10,000 pages long and another called **employee3.gdb**. Firebird starts using **employee2.gdb** only when the primary file reaches 10,000 pages.

```
ALTER DATABASE
ADD FILE 'employee2.gdb'
STARTING AT PAGE 10001 LENGTH 10000
ADD FILE 'employee3.gdb';
```

See Also CREATE DATABASE, DROP DATABASE

See the *Data Definition Guide* for more information about multifile databases and the *Operations Guide* for more information about exclusive database access.

ALTER DOMAIN

Changes a domain definition.

Availability DSQL ESQL ISQL ~~PSQL~~

```
ALTER DOMAIN { name | old_name TO new_name }
SET DEFAULT {literal | NULL | USER}
| DROP DEFAULT
| ADD [CONSTRAINT] CHECK (<dom_search_condition>)
| DROP CONSTRAINT    | new_col_name
  | TYPE datatype;
```

```
<dom_search_condition> =
   VALUE <operator> <val>
   | VALUE [NOT] BETWEEN <val> AND <val>
   | VALUE [NOT] LIKE <val> [ESCAPE <val>]
   | VALUE [NOT] IN (<val> [, <val> ...])
   | VALUE IS [NOT] NULL
   | VALUE [NOT] CONTAINING <val>
   | VALUE [NOT] STARTING [WITH] <val>
   | (<dom_search_condition>)
   | NOT <dom_search_condition>
   | <dom_search_condition> OR <dom_search_condition>
   | <dom_search_condition> AND <dom_search_condition>

 <operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

TABLE 2–2 ALTER DOMAIN syntax elements

Argument	Description
name	Name of an existing domain
SET DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: • *literal*—Inserts a specified string, numeric value, or date value • NULL—Enters a NULL value. • USER—Enters the user name of the current user; column must be of compatible text type to use the default • Defaults set at column level override defaults set at the domain level
DROP DEFAULT	Drops an existing default
ADD [CONSTRAINT] CHECK *dom_search_condition*	Adds a CHECK constraint to the domain definition; a domain definition can include only one CHECK constraint
DROP CONSTRAINT	Drops CHECK constraint from the domain definition
new_col_name	Changes the domain name
TYPE data_type	Changes the domain datatype

Description ALTER DOMAIN changes any aspect of an existing domain except its NOT NULL setting. Changes made to a domain definition affect all column definitions based on the domain that have not been overridden at the table level.

Note To change the NOT NULL setting of a domain, drop the domain and recreate it with the desired combination of features.

The TYPE clause of ALTER DOMAIN does not allow you to make datatype conversions that could lead to data loss.
A domain can be altered by its creator, the SYSDBA user and, on Linux/UNIX, the root user and any user with root privileges.

Example The following ISQL statements create a domain that must have a value > 1,000, then alter it by setting a default of 9,999:

```
CREATE DOMAIN CUSTNO
AS INTEGER
CHECK (VALUE > 1000);
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

See Also **CREATE DOMAIN**, **CREATE TABLE**, **DROP DOMAIN**

For a complete discussion of creating domains, and using them to create column definitions, see Firebird domains in *Using Firebird*— Domains and Generators (ch. 15 p. 285).

ALTER EXCEPTION

Changes the message associated with an existing exception.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `ALTER EXCEPTION` *`name`* `'`*`message`*`'`

Argument	Description
name	Name of an existing exception message
'message'	Quoted string containing ASCII values

Description ALTER EXCEPTION changes the text of an exception error message.

An exception can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Example This ISQL statement alters the message of an exception:

```
ALTER EXCEPTION CUSTOMER_CHECK 'Hold shipment for customer
remittance.';
```

See Also ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions,refer to *Using Firebird*— Error trapping and handling (ch. 25 p. 549).

ALTER INDEX

Activates or deactivates an index.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `ALTER INDEX` *`name`* `{ACTIVE | INACTIVE};`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index
ACTIVE	Changes an INACTIVE index to an ACTIVE one
INACTIVE	Changes an ACTIVE index to an INACTIVE one

Description ALTER INDEX makes an inactive index available for use, or disables the use of an active index. Deactivating an index is exactly like dropping it, except that the index definition remains in the database. Activating an index creates a new index structure.

Before inserting, updating, or deleting a large number of rows, deactivate a table's indexes to avoid altering the index incrementally. When finished, reactivate the index. A reasonable metric is that if you intend to add or delete more than 15% of the rows in a table, or update an indexed column in more than 10% of the rows, you should consider deactivating and reactivating the index.

If an index is in use, ALTER INDEX does not take effect until the index is no longer in use. ALTER INDEX fails and returns an error if the index is defined for a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. To alter such an index, use DROP INDEX to delete the index, then recreate it with CREATE INDEX. An index can be altered by its creator, the SYSDBA user and, on Linux/UNIX, the root user and any user with root privileges.

Note To add or drop index columns or keys, use DROP INDEX to delete the index, then recreate it with CREATE INDEX.

Example The following ISQL statements deactivate and reactivate an index to rebuild it:

```
ALTER INDEX BUDGETX INACTIVE;
ALTER INDEX BUDGETX ACTIVE;
```

See Also ALTER TABLE, CREATE INDEX, DROP INDEX, SET STATISTICS

ALTER PROCEDURE

Changes the definition of an existing stored procedure.

Availability DSQL ~~ESQL~~ ISQL ~~PSQL~~

Syntax

```
ALTER PROCEDURE name
            [(param <datatype> [, param <datatype> ...])]
            [RETURNS (param <datatype> [, param <datatype> ...])]
            AS <procedure_body> [terminator]
```

Argument	Description
name	Name of an existing procedure
param datatype	Input parameters used by the procedure; valid datatypes are listed under CREATE PROCEDURE

Argument	Description
RETURNS *param datatype*	Output parameters used by the procedure; valid datatypes are listed under CREATE PROCEDURE
procedure_body	The procedure body includes: • Local variable declarations • A block of statements in procedure and trigger language See CREATE PROCEDURE for a complete description
terminator	Terminator defined by the ISQL SET TERM command to signify the end of the procedure body; required by ISQL

Description ALTER PROCEDURE changes an existing stored procedure without affecting its dependencies. It can modify a procedure's input parameters, output parameters, and body.

The complete procedure header and body must be included in the ALTER PROCEDURE statement. The syntax is exactly the same as CREATE PROCEDURE, except CREATE is replaced by ALTER.

Important Be careful about changing the type, number, and order of input and output parameters to a procedure, since existing application code may assume the procedure has its original format. Check for dependencies between procedures before changing parameters. Should you change parameters and find that another procedure can neither be altered to accept the new parameters or deleted, change the original procedure back to its original parameters, fix the calling procedure, then change the called procedure.

Important Because triggers use semicolons internally, you must change the default terminator character for ISQL before attempting to alter or create a procedure. Use the SET TERM command to specify the new termination characters. The syntax of the SET TERM command is

Syntax `SET TERM <new terminator> <old terminator>`

The `<old terminator>` is not part of the command, but the command terminator. Because SET TERM is exclusively an ISQL command, the command terminator is always required.
A procedure can be altered by its creator, the SYSDBA user and, on Linux/UNIX, the root user and any user with root privileges..

Procedures in use are not altered until they are no longer in use.
ALTER PROCEDURE changes take effect when they are committed. Changes are then reflected in all applications that use the procedure without recompiling or relinking.

Example The following ISQL statements alter the GET_EMP_PROJ procedure, changing the return parameter to have a datatype of VARCHAR(20):

```
SET TERM !! ;
ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID VARCHAR(20)) AS
BEGIN
FOR SELECT PROJ_ID
FROM EMPLOYEE_PROJECT
WHERE EMP_NO = :emp_no
INTO :proj_id
DO
SUSPEND;
END !!
SET TERM ; !!
```

See Also **CREATE PROCEDURE**, **DROP PROCEDURE**, **EXECUTE PROCEDURE**

For more information on creating and using procedures, see *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).
For a complete description of the statements in procedure and trigger language, refer to PSQL-Firebird Procedural Language.

ALTER TABLE

Changes a table by adding, dropping, or modifying columns or integrity constraints.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `ALTER TABLE` *`table`* `<operation> [, <operation> ...];`

```
 <operation> = ADD <col_def>
 | ADD <tconstraint>
   | ALTER [COLUMN] column_name <alt_col_clause>
   | DROP col
   | DROP CONSTRAINT constraint

<alt_col_clause> = TO new_col_name
   | TYPE new_col_datatype
   | POSITION new_col_position

        <col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
           [DEFAULT {literal | NULL | USER}]
           [NOT NULL]
           [<col_constraint>]
           [COLLATE collation]

        <datatype> =
           {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[<array_dim>]
           | (DATE | TIME | TIMESTAMP}[<array_dim>]
           | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
           | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
              [<array_dim>] [CHARACTER SET charname]
           | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
              [VARYING] [(int)] [<array_dim>]
           | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
              [CHARACTER SET charname]
           | BLOB [(seglen [, subtype])]

        <array_dim> = [[x:]y [, [x:]y ...]]

        <expr> = a valid SQL expression that results in a single value
```

```
<col_constraint> = [CONSTRAINT constraint]
    { UNIQUE
    | PRIMARY KEY
    | REFERENCES other_table [(other_col [, other_col …])]
       [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
       [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<tconstraint> = [CONSTRAINT constraint]
    {{PRIMARY KEY | UNIQUE} (col [, col ...])
    | FOREIGN KEY (col [, col ...])
       REFERENCES other_table [(other_col [, other_col ...])]
          [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
          [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}

<search_condition> = <val> <operator> {<val> | (<select_one>)}
    | <val> [NOT] BETWEEN <val> AND <val>
    | <val> [NOT] LIKE <val> [ESCAPE <val>]
    | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
    | <val> IS [NOT] NULL
    | <val> {>= | <=} <val>
    | <val> [NOT] {= | < | >} <val>
    | {ALL | SOME | ANY} (<select_list>)
    | EXISTS (<select_expr>)
    | SINGULAR (<select_expr>)
    | <val> [NOT] CONTAINING <val>
    | <val> [NOT] STARTING [WITH] <val>
    | (<search_condition>)
    | NOT <search_condition>
    | <search_condition> OR <search_condition>
    | <search_condition> AND <search_condition>
```

```
<val> = { col [<array_dim>] | :variable
    | <constant> | <expr> | <function>
    | udf ([<val> [, <val> ...]])
    | NULL | USER | RDB$DB_KEY | ? }
    [COLLATE collation]

<constant> = num | 'string' | _charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
    | SUM ([ALL] <val> | DISTINCT <val>)
    | AVG ([ALL] <val> | DISTINCT <val>)
    | MAX ([ALL] <val> | DISTINCT <val>)
    | MIN ([ALL] <val> | DISTINCT <val>)
    | CAST (<val> AS <datatype>)
    | UPPER (<val>)
    | GEN_ID (generator, <val>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column; returns exactly one value.

<select_list> = SELECT on a single column; returns zero or more values.

<select_expr> = SELECT on a list of values; returns zero or more
values.
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Notes on ALTER TABLE syntax

- You cannot specify a COLLATE clause for blob columns.

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array = varchar(6)[5,5]
```

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 20 and ends at 30:

```
my_array = integer[20:30]
```

- For the full syntax of *search_condition*, see CREATE TABLE.

TABLE 2–3 The ALTER TABLE statement

Argument	Description
table	Name of an existing table to modify
operation	Action to perform on the table. Valid options are: • ADD a new column or table constraint to a table • DROP an existing column or constraint from a table
col_def	Description of a new column to add • Must include a column name and *datatype* • Can also include default values, column constraints, and a specific collation order
col	Name of the column to add or drop; column name must be unique within the table
datatype	Datatype of the column; see Supported Datatypes.
ALTER [COLUMN]	Modifies column names, datatypes, and positions.

TABLE 2–3 The ALTER TABLE statement (*continued*)

Argument	Description
COMPUTED [BY] *expr*	Specifies that the value of the column's data is calculated from *expr* at runtime and is therefore not allocated storage space in the database • *expr* can be any arithmetic expression valid for the datatypes in the expression • Any columns referenced in *expr* must exist before they can be used in *expr* • *expr* cannot reference blob columns • *expr* must return a single value, and cannot return an array
domain	Name of an existing domain
DEFAULT	Specifies a default value for column data; this value is entered when no other entry is made; possible values are: • *literal*: Inserts a specified string, numeric value, or date value • NULL: Enters a NULL value. This is the default DEFAULT. • USER: Enters the user name of the current user; column must be of compatible text type to use the default Defaults set at column level override defaults set at the domain level
CONSTRAINT *constraint*	Name of a column or table constraint; the constraint name must be unique within the table
constraint_def	Specifies the kind of column constraint; valid options are UNIQUE, PRIMARY KEY, CHECK, and REFERENCES
CHECK *search_condition*	An attempt to enter a new value in the column fails if the value does not meet the *search_condition*

TABLE 2–3 The ALTER TABLE statement (*continued*)

Argument	Description
REFERENCES	Specifies that the column values are derived from column values in another table; if you do not specify column names, Firebird looks for a column with the same name as the referencing column in the referenced table
ON DELETE | ON UPDATE	Used with REFERENCES: Changes a foreign key whenever the referenced primary key changes; valid options are: • [Default] NO ACTION: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks • CASCADE: For ON DELETE, deletes the corresponding foreign key; for ON UPDATE, updates the corresponding foreign key to the new value of the primary key • SET NULL: Sets all the columns of the corresponding foreign key to NULL • SET DEFAULT: Sets every column of the corresponding foreign key to whatever default value is in effect when the referential integrity constraint is defined. When the default for a foreign column changes *after* the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint.
NOT NULL	Specifies that a column cannot contain a NULL value • If a table already has rows, a new column cannot be NOT NULL • NOT NULL is a column attribute only
DROP CONSTRAINT	Drops the specified table constraint

TABLE 2–3 The ALTER TABLE statement (*continued*)

Argument	Description
table_constraint	Description of the new table constraint; constraints can be PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK
COLLATE *collation*	Establishes a default sorting behavior for the column; see Character Sets and Collation Orders for more information

Description ALTER TABLE modifies the structure of an existing table. A single ALTER TABLE statement can perform multiple adds and drops.

- A table can be altered by its creator, the SYSDBA user and, on Linux/UNIX, the root user and any user with root privileges..
- ALTER TABLE fails if the new data in a table violates a PRIMARY KEY or UNIQUE constraint definition added to the table. Dropping or altering a column fails if any of the following are true:
- The column is part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint
- The column is used in a CHECK constraint
- The column is used in the *value* expression of a computed column
- The column is referenced by another database object such as a view

Important When a column is dropped, all data stored in it is lost.

Constraints

- Referential integrity constraints include optional ON UPDATE and ON DELETE clauses. They define the change to be made to the referencing column when the referenced column is updated or deleted.The values for these cascading referential integrity options are given in **Table 2–3, "The ALTER TABLE statement," on page 58**.
- To delete a column referenced by a computed column, you must drop the computed column before dropping the referenced column. To drop a column referenced in a FOREIGN KEY constraint, you must drop the constraint before dropping the referenced column. To drop a PRIMARY KEY or UNIQUE

constraint on a column that is referenced by FOREIGN KEY constraints, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references.

- You can create a FOREIGN KEY reference to a table that is owned by someone else only if that owner has explicitly granted you the REFERENCES privilege on that table using GRANT. Any user who updates your foreign key table must have REFERENCES or SELECT privileges on the referenced primary key table.
- You can add a check constraint to a column that is based on a domain, but be aware that changes to tables that contain CHECK constraints with subqueries may cause constraint violations.
- Naming column constraints is optional. If you do not specify a name, Firebird assigns a system-generated name. Assigning a descriptive name can make a constraint easier to find for changing or dropping, and more descriptive when its name appears in a constraint violation error message.
- When creating new columns in tables with data, do not use the UNIQUE constraint. If you use the NOT NULL constraint on a table with data, you should also specify a default value.
- The following ISQL statement adds a column to a table and drops a column:

```
ALTER TABLE COUNTRY
ADD CAPITAL VARCHAR(25),
DROP CURRENCY;
```

This statement results in the loss of all data in the dropped CURRENCY column.
The next **ISQL** statement changes the name of the LARGEST_CITY column to BIGGEST_CITY:

```
ALTER TABLE COUNTRY ALTER LARGEST_CITY TO BIGGEST_CITY;
```

See Also **ALTER DOMAIN**, **CREATE DOMAIN**, **CREATE TABLE**

For more information about altering tables, see *Using Firebird*— Altering tables (ch. 17 p. 340).

ALTER TRIGGER

Changes an existing trigger.

Availability DSQL ~~ESQL~~ ISQL ~~PSQL~~

Syntax

```
ALTER TRIGGER name
            [ACTIVE | INACTIVE]
            [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
            [POSITION number]
            [AS <trigger_body>] [terminator]
```

Argument	Description
name	Name of an existing trigger
ACTIVE	[Default] Specifies that a trigger action takes effect when fired
INACTIVE	Specifies that a trigger action does *not* take effect
BEFORE	Specifies the trigger fires before the associated operation takes place
AFTER	Specifies the trigger fires after the associated operation takes place
DELETE|INSERT |UPDATE	Specifies the table operation that causes the trigger to fire
POSITION *number*	Specifies order of firing for triggers before the same action or after the same action • *number* must be an integer between 0 and 32,767, inclusive • Lower-number triggers fire first • Triggers for a table need not be consecutive; triggers on the same action with the same position number fire in random order
trigger_body	Body of the trigger: a block of statements in procedure and trigger language • See CREATE TRIGGER for a complete description
terminator	Terminator defined by the ISQL SET TERM command to signify the end of the trigger body; not needed when altering only the trigger header

Description ALTER TRIGGER changes the definition of an existing trigger. If any of the arguments to ALTER TRIGGER are omitted, then they default to their current values, that is the value specified by CREATE TRIGGER, or the last ALTER TRIGGER.

ALTER TRIGGER can change:

- Header information only, including the trigger activation status, when it performs its actions, the event that fires the trigger, and the order in which the trigger fires compared to other triggers.
- Body information only, the trigger statements that follow the AS clause.
- Header and trigger body information. In this case, the new trigger definition replaces the old trigger definition.

A trigger can be altered by its creator, the SYSDBA user and, on Linux/UNIX, the root user and any user with root privileges.

Note To alter a trigger defined automatically by a CHECK constraint on a table, use ALTER TABLE to change the constraint definition.

It is important to understand the timing of alterations to a trigger. If dependent transactions are under way, the changes will be deferred until the first opportunity when there are no dependent transactions. For a detailed explanation, see *Using Firebird*— Timing of modifications (ch. 25 p. 517).

Examples The following ISQL statement modifies the trigger, SET_CUST_NO, to be inactive:

```
ALTER TRIGGER SET_CUST_NO INACTIVE;
```

The next ISQL statement modifies the trigger, SET_CUST_NO, to insert a row into the table, NEW_CUSTOMERS, for each new customer.

```
SET TERM !! ;
ALTER TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
BEGIN
NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
INSERT INTO NEW_CUSTOMERS(NEW.CUST_NO, 'TODAY')
END !!
SET TERM ; !!
```

See Also **CREATE TRIGGER**, **DROP TRIGGER**

For a complete description of the statements in procedure and trigger language, PSQL-Firebird Procedural Language.
For more information, see *Using Firebird*— Triggers (ch. 25 p. 532).

AVG( )

Calculates the average of numeric values in a specified column or expression.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `AVG ([ALL] value | DISTINCT value)`

Argument	Description
ALL	Returns the average of all values
DISTINCT	Eliminates duplicate values before calculating the average
value	A column or expression that evaluates to a numeric datatype

Description AVG() is an aggregate function that returns the average of the values in a specified column or expression. Only numeric datatypes are allowed as input to AVG().

If a field value involved in a calculation is NULL or unknown, it is automatically excluded from the calculation. Automatic exclusion prevents averages from being skewed by meaningless data.
AVG() computes its value over a range of selected rows. If the number of rows returned by a SELECT is zero, AVG() returns a NULL value.

Examples The following ESQL statement returns the average of all rows in a table:

```
EXEC SQL
  SELECT AVG (BUDGET) FROM DEPARTMENT INTO :avg_budget;
```

The next ESQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX() over a subset of rows in a table:

```
EXEC SQL
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also COUNT( ), MAX( ), MIN( ), SUM( )

BASED ON

Declares a host-language variable based on a column.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax BASED [ON] [*dbhandle*.]*table.col*[.SEGMENT] *variable*;

Argument	Description
dbhandle	Handle for the database in which a table resides in a multi-database program; *dbhandle* must be previously declared in a SET DATABASE statement
table.col	Name of table and name of column on which the variable is based
.SEGMENT	Bases the local variable size on the segment length of the blob column during BLOB FETCH operations; use only when *table.col* refers to a column of BLOB datatype
variable	Name of the host-language variable that inherits the characteristics of a database column

Description BASED ON is a preprocessor directive that creates a host-language variable based on a column definition. The host variable inherits the attributes described for the column and any characteristics that make the variable type consistent with the programming language in use. For example, in C, BASED ON adds one byte to CHAR and VARCHAR variables to accommodate the NULL character terminator.

Use BASED ON in a program's variable declaration section.

Note BASED ON does not require the EXEC SQL reserved words.

To declare a host-language variable large enough to hold a blob segment during FETCH operations, use the SEGMENT option of the BASED ON clause. The variable's size is derived from the segment length of a blob column. If the segment length for the blob column is changed in the database, recompile the program to adjust the size of host variables created with BASED ON.

Examples The following embedded statements declare a host variable based on a column:

```
EXEC SQL
BEGIN DECLARE SECTION
BASED ON EMPLOYEE.SALARY salary;
EXEC SQL
END DECLARE SECTION;
```

See Also BEGIN DECLARE SECTION, CREATE TABLE, END DECLARE SECTION

BEGIN DECLARE SECTION

Identifies the start of a host-language variable declaration section.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `BEGIN DECLARE SECTION;`

Description BEGIN DECLARE SECTION is used in ESQL applications to identify the start of host-language variable declarations for variables that will be used in subsequent SQL statements. BEGIN DECLARE SECTION is also a preprocessor directive that instructs **gpre** to declare SQLCODE automatically for the applications programmer.

Important BEGIN DECLARE SECTION must always appear within a module's global variable declaration section.

Example The following ESQL statements declare a section and a host-language variable:

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED ON EMPLOYEE.SALARY salary;
EXEC SQL
END DECLARE SECTION;
```

See Also BASED ON, END DECLARE SECTION

CAST( )

Converts a column from one datatype to another.

Availability DSQL ESQL ISQL PSQL

Syntax CAST (*value* AS *datatype*)

Argument	Description
val	A column, constant, or expression; in SQL, *val* can also be a host-language variable, function, or UDF
datatype	Datatype to which to convert

Description CAST() allows mixing of numerics and characters in a single expression by converting *val* to a specified datatype.

Normally, only similar datatypes can be compared in search conditions. CAST() can be used in search conditions to translate one datatype into another for comparison purposes.
Datatypes can be converted as shown in the following table:

TABLE 2–4 Compatible datatypes for CAST()

From datatype class	To datatype class
Numeric	Numeric, character, varying character, date, time, timestamp
Character, varying character	Numeric, date, time, timestamp
Date	Character, varying character, timestamp
Time	Character, varying character, timestamp
Timestamp	Character, varying character, date, time
Blob, arrays	—

An error results if a given datatype cannot be converted into the datatype specified in CAST().

Example In the following WHERE clause, CAST() is used to translate a CHARACTER datatype, INTERVIEW_DATE, to a DATE datatype to compare against a DATE datatype, HIRE_DATE:

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

To cast to a VARCHAR datatype, you must specify the length of the string:

```
UPDATE client SET charef = CAST (clientref AS VARCHAR(20));
```

Casting to a character datatype allows for specifying a character set for the result.

See Also UPPER( )

CLOSE

Closes an open cursor.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `CLOSE` *`cursor;`*

Argument	Description
cursor	Name of an open cursor

Description CLOSE terminates the specified cursor, releasing the rows in its active set and any associated system resources. A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows in turn and update in place.

There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and releases system resources

FETCH statements cannot be issued against a closed cursor. Until a cursor is closed and reopened, Firebird does not reevaluate values passed to the search conditions. Another user can commit changes to the database while a cursor is open, making the active set different the next time that cursor is reopened.

Note In addition to CLOSE, COMMIT and ROLLBACK automatically close all cursors in a transaction.

Example The following ESQL statement closes a cursor:

```
EXEC SQL
CLOSE BC;
```

See Also **CLOSE (BLOB)**, **COMMIT**, **DECLARE CURSOR**, **FETCH**, **OPEN**, **ROLLBACK**

CLOSE (BLOB)

Terminates a specified blob cursor and releases associated system resources.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `CLOSE blob_cursor;`

Argument	Description
blob_cursor	Name of an open blob cursor

Description CLOSE closes an opened read or insert blob cursor. Generally a blob cursor should be closed only after:

- Fetching all the blob segments for BLOB READ operations.
- Inserting all the segments for BLOB INSERT operations.

Example The following ESQL statement closes a blob cursor:

```
EXEC SQL
CLOSE BC;
```

See Also **DECLARE CURSOR (BLOB)**, **FETCH (BLOB)**, **INSERT CURSOR (BLOB)**, **OPEN (BLOB)**

COMMIT

Makes a transaction's changes to the database permanent, and ends the transaction.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `COMMIT [WORK] [TRANSACTION name] [RELEASE] [RETAIN [SNAPSHOT]];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
WORK	An optional word used for compatibility with other relational databases that require it
TRANSACTION *name*	Commits transaction *name* to database (ESQL only). Without this option, COMMIT affects the default transaction
RELEASE	Available for compatibility with earlier versions of Firebird
RETAIN [SNAPSHOT]	Commits changes and retains current transaction context

Description COMMIT is used to end a transaction and:

- Write all updates to the database.
- Make the transaction's changes visible to subsequent SNAPSHOT transactions or READ COMMITTED transactions.
- Close open cursors, unless the RETAIN argument is used.

A transaction ending with COMMIT is considered a successful termination. Always use COMMIT or ROLLBACK to end the default transaction.

Tip After read-only transactions, which make no database changes, use COMMIT rather than ROLLBACK. The effect is the same, but the performance of subsequent transactions is better and the system resources used by them are reduced.

Important The RELEASE argument is available only for compatibility with previous InterBase versions. To detach from a database use DISCONNECT.

Examples The following ISQL statement makes permanent the changes to the database made by the default transaction:

```
COMMIT;
```

The next ESQL statement commits a named transaction:

```
EXEC SQL
  COMMIT TR1;
```

The following ESQL statement uses COMMIT RETAIN to commit changes while maintaining the current transaction context:

```
EXEC SQL
  COMMIT RETAIN;
```

See Also DISCONNECT, ROLLBACK

For more information about handling transactions, see *Using Firebird*— Transactions in Firebird (ch. 8 p. 90).

CONNECT

Attaches to one or more databases.

Availability ~~DSQL~~ ESQL ISQL* ~~PSQL~~
*A subset of CONNECT options is available in ISQL.

Syntax ISQL **form:**

```
CONNECT 'filespec' [USER 'username'][PASSWORD 'password']
[CACHE int] [ROLE 'rolename']
```

ESQL form:

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

<db_specs> = dbhandle
   | {'filespec' | :variable} AS dbhandle

<config_opts> = [USER {'username' | :variable}]
   [PASSWORD {'password' | :variable}]
   [ROLE {'rolename' | :variable}]
   [CACHE int [BUFFERS]]
```

Argument	Description
{ALL | DEFAULT}	Connects to all databases specified with SET DATABASE; options specified with CONNECT TO ALL affect all databases.
'filespec'	Database file name; can include path specification and node. The filespec must be in single quotes if it includes spaces. See Connecting to a Microsoft Windows Server or Connecting to a Microsoft Windows Server **in ch. 6 of** of Using Firebird for details about file specification.
dbhandle	Database handle declared in a previous SET DATABASE statement; available in ESQL but not in ISQL.
:variable	Host-language variable specifying a database, user name, or password; available in ESQL but not in ISQL.
AS *dbhandle*	Attaches to a database and assigns a previously declared handle to it; available in ESQL but not in ISQL.
USER {'*username*' | :*variable*}	String or host-language variable that specifies a user name for use when attaching to the database. The server checks the user name against the security database. User names are case insensitive on the server.

Argument	Description
PASSWORD {'*password*' | :*variable*}	String or host-language variable, up to 8 characters in size, that specifies password for use when attaching to the database. The server checks the user name and password against the security database. Case sensitivity is retained for the comparison.
ROLE {'*rolename*' | :*variable*}	String or host-language variable, up to 31 characters in size, which specifies the role that the user adopts on connection to the database. The user must have previously been granted membership in the role to gain the privileges of that role. Regardless of role memberships granted, the user has the privileges of a role at connect time only if a ROLE clause is specified in the connection. The user can adopt at most one role per connection, and cannot switch roles except by reconnecting.
CACHE *int* [BUFFERS]	Sets the number of cache buffers for a database, which determines the number of database pages a program can use at the same time. Values for *int*: • Default: 256 • Maximum value: system-dependent Do not use the *filespec* form of database name with cache assignments.

Description The CONNECT statement:

- Initializes database data structures.
- Determines if the database is on the originating node (a *local database*) or on another node (a *remote database*). An error message occurs if Firebird cannot locate the database.

- Optionally specifies one or more of a user name, password, or role for use when attaching to the database. PC clients must always send a valid user name and password. Firebird recognizes only the first eight characters of a password.

If an Firebird user has ISC_USER and ISC_PASSWORD environment variables set and the user defined by those variables is not in the **isc4.gdb**, the user will receive the following error when attempting to view **isc4.gdb** users : “undefined user name and password.”

- Attaches to the database and verifies the header page. The database file must contain a valid database, and the on-disk structure (ODS) version number of the database must be the one recognized by the installed version of Firebird on the server, or Firebird returns an error.
- Optionally establishes a database handle declared in a SET DATABASE statement.
- Specifies a cache buffer for the process attaching to a database.

In SQL programs before a database can be opened with CONNECT, it must be declared with the SET DATABASE statement. ISQL does not use SET DATABASE.

In SQL programs while the same CONNECT statement can open more than one database, use separate statements to keep code easy to read.

When CONNECT attaches to a database, it uses the default character set (NONE), or one specified in a previous SET NAMES statement.

In SQL programs the CACHE option changes the database cache size count (the total number of available buffers) from the default of 75. This option can be used to:

- Sets a new default size for all databases listed in the CONNECT statement that do not already have a specific cache size.
- Specifies a cache for a program that uses a single database.
- Changes the cache for one database without changing the default for all databases used by the program.

The size of the cache persists as long as the attachment is active. If a database is already attached through a multi-client server, an increase in cache size due to a new attachment persists until all the attachments end. A decrease in cache size does not affect databases that are already attached through a server.

A subset of CONNECT features is available in ISQL: database file name, USER, and PASSWORD. ISQL can only be connected to one database at a time. Each time CONNECT is used to attach to a database, previous attachments are disconnected.

Examples The following statement opens a database for use in ISQL. It uses all the CONNECT options available to ISQL:

```
CONNECT 'employee.gdb' USER 'ACCT_REC' PASSWORD 'peanuts';
```

The next statement, from an embedded application, attaches to a database file stored in the host-language variable and assigns a previously declared database handle to it:

```
EXEC SQL
SET DATABASE DB1 = 'employee.gdb';
EXEC SQL
  CONNECT :db_file AS DB1;
```

The following ESQL statement attaches to **employee.gdb** and allocates 150 cache buffers:

```
EXEC SQL
  CONNECT 'accounts.gdb' CACHE 150;
```

The next ESQL statement connects the user to all databases specified with previous SET DATABASE statements:

```
EXEC SQL
CONNECT ALL USER 'ACCT_REC' PASSWORD 'peanuts'
  CACHE 50;
```

The following ESQL statement attaches to the database, **employee.gdb**, with 80 buffers and database **employee2.gdb** with the default of 75 buffers:

```
EXEC SQL
  CONNECT 'employee.gdb' CACHE 80, 'employee2.gdb';
```

The next ESQL statement attaches to all databases and allocates 50 buffers:

```
EXEC SQL
  CONNECT ALL CACHE 50;
```

The following ESQL statement connects to EMP1 and v, setting the number of buffers for each to 80:

```
EXEC SQL
  CONNECT EMP1 CACHE 80, EMP2 CACHE 80;
```

The next ESQL statement connects to two databases identified by variable names, setting different user names and passwords for each:

```
EXEC SQL
CONNECT
:orderdb AS DB1 USER 'ACCT_REC' PASSWORD 'peanuts',
:salesdb AS DB2 USER 'ACCT_PAY' PASSWORD 'payout';
```

See Also DISCONNECT, SET DATABASE, SET NAMES

See *Using Firebird*— Configuring the database cache (ch. 5 p. 67) for more information about cache buffers and Managing Security in ch. 22 of the same volume for more information about database security.

COUNT( )

Calculates the number of rows that satisfy a query's search condition.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `COUNT ( * | [ALL] value | DISTINCT value)`

Argument	Description
*	Retrieves the number of rows in a specified table, including NULL values
ALL	Counts all non-NULL values in a column
DISTINCT	Returns the number of unique, non-NULL values for the column
val	A column or expression

Description COUNT() is an aggregate function that returns the number of rows that satisfy a query's search condition. It can be used in views and joins as well as in tables.

Example The following ESQL statement returns the number of unique currency values it encounters in the COUNTRY table:

```
EXEC SQL
  SELECT COUNT (DISTINCT CURRENCY) FROM COUNTRY INTO :cnt;
```

See Also AVG( ), MAX( ), MIN( ) SUM( )

CREATE DATABASE

Creates a new database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

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

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

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
'*filespec*'	A new database file specification; file naming conventions are platform-specific. See *Using Firebird*— Creating a database (ch. 14 p. 274) for details about database file specification.
USER '*username*'	Checks the *username* against valid user name and password combinations in the security database on the server where the database will reside • Windows client applications must provide a user name on attachment to a server • Any client application attaching to a database on NT or NetWare must provide a user name on attachment
PASSWORD '*password*'	Checks the *password* against valid user name and password combinations in the security database on the server where the database will reside; can be up to 8 characters • Windows client applications must provide a user name and password on attachment to a server • Any client application attaching to a database on NT or NetWare must provide a password on attachment
PAGE_SIZE [=] *int*	Size, in bytes, for database pages *int* can be 1024 (default), 2048, 4096, 8192 or 16384.
DEFAULT CHARACTER SET *charset*	Sets default character set for a database *charset* is the name of a character set; if omitted, character set defaults to NONE

Argument	Description
FILE '*filespec*'	Names one or more secondary files to hold database pages after the primary file is filled. For databases created on remote servers, secondary file specifications cannot include a node name.
STARTING [AT [PAGE]] *int*	Specifies the starting page number for a secondary file.
LENGTH [=] *int* [PAGE[S]]	Specifies the length of a primary or secondary database file. Use for primary file only if defining a secondary file in the same statement.

Description CREATE DATABASE creates a new, empty database and establishes the following characteristics for it:

- The name of the primary file that identifies the database for users.

By default, databases are contained in single files.

- The name of any secondary files in which the database is stored.

A database can reside in more than one disk file if additional file names are specified as secondary files. If a database is created on a remote server, secondary file specifications cannot include a node name.

- The size of database pages.

Increasing page size can improve performance for the following reasons:

- Indexes work faster because the depth of the index is kept to a minimum.
- Keeping large rows on a single page is more efficient.
- Blob data is stored and retrieved more efficiently when it fits on a single page.

If most transactions involve only a few rows of data, a smaller page size might be appropriate, since less data needs to be passed back and forth and less memory is used by the disk cache.

- The number of pages in each database file.
- The dialect of the database.

The initial dialect of the database is the dialect of the client that creates it. For example, if you are using ISQL, either start it with the **-sql_dialect** *n* switch or issue the SET SQL DIALECT *n* command before issuing the

CREATE DATABASE command. Typically, you would create all databases in dialect 3. Dialect 1 exists to ease the migration of legacy databases.

Note To change the dialect of a database, use **gfix.** See the Migration chapter in xxxxxx for information about migrating databases.

- The character set used by the database.

For a list of the character sets recognized by Firebird, see Character sets and collations available in Firebird on page 249. For a discussion of character set usage, see *Using Firebird*— Character Sets and Collation Orders (ch. 16 p. 301).

Choice of DEFAULT CHARACTER SET limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In that case, no transliteration is performed between the source and destination character sets, and transliteration errors may occur during assignment.

- System tables that describe the structure of the database.

After creating the database, you define its tables, views, indexes, and system views as well as any triggers, generators, stored procedures, and UDFs that you need.

Important In DSQL, you must execute CREATE DATABASE EXECUTE IMMEDIATE. The database handle and transaction name, if present, must be initialized to zero prior to use.

Read-only databases

Databases are always created in *read-write mode*. You can change a database to *read-only mode* in either of two ways: You can specify **mode -read_only** when you restore a backup or you can use **gfix -mode read_only** to change the mode of a read-write database to read-only. See *Using Firebird*— Read-only databases (ch. 20 p. 375).

About file sizes

Firebird dynamically expands the last file in a database as needed until it reaches the filesystem limit for shared access files. This applies to single-file database as well as to the last file of multifile databases. It is important to be aware of the maximum size allowed for shared access files in the filesystem environment where your databases live. Firebird database files are limited to 2GB in many environments. The total file size is the product of the number of database pages times the page size. The default page size is 4KB and the maximum page size is 16KB. However, Firebird files are small at creation time and increase in size as needed. The product of number of pages times page size represents a potential maximum size, not the size at creation.

Examples The following ISQL statement creates a database in the default directory using ISQL:

```
CREATE DATABASE 'employee.gdb';
```

The next ESQL statement creates a database with a page size of 2048 bytes rather than the default of 4096:

```
EXEC SQL
  CREATE DATABASE 'employee.gdb' PAGE_SIZE 2048;
```

The following ESQL statement creates a database stored in two files and specifies its default character set:

```
EXEC SQL
CREATE DATABASE 'employee.gdb'
DEFAULT CHARACTER SET ISO8859_1
FILE 'employee2.gdb' STARTING AT PAGE 10001;
```

See Also ALTER DATABASE, DROP DATABASE

See also the following discussion topics in Using Firebird:

- Multi-file databases
- Character Sets and Collation Orders
- Specifying database page size

CREATE DOMAIN

Creates a column definition that is global to the database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
CREATE DOMAIN domain [AS] <datatype>
            [DEFAULT {literal | NULL | USER}]
            [NOT NULL] [CHECK (<dom_search_condition>)]
            [COLLATE collation];

 <datatype> =
 {SMALLINT|INTEGER|FLOAT|DOUBLE PRECISION}[<array_dim>]
 | {DATE | TIME | TIMESTAMP}[<array_dim>]
 | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
 | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
 [<array_dim>] [CHARACTER SET charname]
 | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
 [VARYING] [(int)] [<array_dim>]
 | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
 [CHARACTER SET charname]
   | BLOB [(seglen [, subtype])]

<array_dim> = [[x:]y [, [x:]y ...]]

<dom_search_condition> =
   VALUE <operator> value
   | VALUE [NOT] BETWEEN value AND value
   | VALUE [NOT] LIKE value [ESCAPE value]
   | VALUE [NOT] IN (value [, value ...])
   | VALUE IS [NOT] NULL
   | VALUE [NOT] CONTAINING value
   | VALUE [NOT] STARTING [WITH] value
   | (<dom_search_condition>)
   | NOT <dom_search_condition>
   | <dom_search_condition> OR <dom_search_condition>
   | <dom_search_condition> AND <dom_search_condition>
```

```
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Note on the CREATE DOMAIN syntax

- COLLATE is useful only for text data, not for numeric types. Also, you cannot specify a COLLATE clause for BLOB columns.
- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array = varchar(6)[5,5]
```

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 20 and ends at 30:

```
my_array = integer[20:30]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
domain	Unique name for the domain
datatype	SQL datatype
DEFAULT	Specifies a default column value that is entered when no other entry is made; possible values are: *literal*—Inserts a specified string, numeric value, or date value NULL—Enters a NULL value USER—Enters the user name of the current user; column must be of compatible character type to use the default

Argument	Description
NOT NULL	Specifies that the values entered in a column cannot be NULL
CHECK (*dom_search_condition*)	Creates a single CHECK constraint for the domain
VALUE	Placeholder for the name of a column eventually based on the domain
COLLATE *collation*	Specifies a collation sequence for the domain

Description CREATE DOMAIN builds an inheritable column definition that acts as a template for columns defined with CREATE TABLE or ALTER TABLE. The domain definition contains a set of characteristics, which include:

- Datatype
- An optional default value
- Optional disallowing of NULL values
- An optional CHECK constraint
- An optional collation clause

The CHECK constraint in a domain definition sets a *dom_search_condition* that must be true for data entered into columns based on the domain. The CHECK constraint cannot reference any domain or column.

Note Be careful not to create a domain with contradictory constraints, such as declaring a domain NOT NULL and assigning it a DEFAULT value of NULL.

The datatype specification for a CHAR or VARCHAR text domain definition can include a CHARACTER SET clause to specify a character set for the domain. Otherwise, the domain uses the default database character set. For a complete list of character sets recognized by Firebird, see chapter 4, Character Sets and Collation Orders (p. 249). If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In these cases, no transliteration is performed between the source and destination character sets, so errors can occur during assignment.

The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and NCHAR text datatypes. Choice of collation order is restricted to those supported for the domain's given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the datatype definition. For a complete list of collation orders recognized by Firebird, see chapter 4, Character Sets and Collation Orders (p. 249).

Columns based on a domain definition inherit all characteristics of the domain. The domain default, collation clause, and NOT NULL setting can be overridden when defining a column based on a domain. A column based on a domain can add additional CHECK constraints to the domain CHECK constraint.

Examples The following ISQL statement creates a domain that must have a positive value greater than 1,000, with a default value of 9,999. The reserved word VALUE substitutes for the name of a column based on this domain.

```
CREATE DOMAIN CUSTNO
AS INTEGER
DEFAULT 9999
  CHECK (VALUE > 1000);
```

The next ISQL statement limits the values entered in the domain to four specific values:

```
CREATE DOMAIN PRODTYPE
  AS VARCHAR(12)
  CHECK (VALUE IN ('software', 'hardware', 'other', 'N/A'));
```

The following ISQL statement creates a domain that defines an array of CHAR datatype:

```
CREATE DOMAIN DEPTARRAY AS CHAR(31) [4:5];
```

In the following ISQL example, the first statement creates a domain with USER as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
   DEFAULT USER;

CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME,
   ORDER_AMT DECIMAL(8,2));

INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
   VALUES ('1-MAY-93', 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so Firebird automatically inserts the user name of the current user, JSMITH:

```
SELECT * FROM ORDERS;

1-MAY-93 JSMITH 512.36
```

The next ISQL statement creates a BLOB domain with a TEXT subtype that has an assigned character set:

```
CREATE DOMAIN DESCRIPT AS
   BLOB SUB_TYPE TEXT SEGMENT SIZE 80
   CHARACTER SET SJIS;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE TABLE, DROP DOMAIN

For more information about character set specification and collation orders, see xxxxx.

CREATE EXCEPTION

Creates a used-defined error and associated message for use in stored procedures and triggers.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `CREATE EXCEPTION name 'message';`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name associated with the exception message; must be unique among exception names in the database
'*message*'	Quoted string containing alphanumeric characters and punctuation; maximum length = 78 characters.

Description CREATE EXCEPTION creates an exception, a user-defined error with an associated message. Exceptions may be raised in triggers and stored procedures.

Exceptions are global to the database. The same message or set of messages is available to all stored procedures and triggers in an application. For example, a database can have English and French versions of the same exception messages and use the appropriate set as needed.

When raised by a trigger or a stored procedure, an exception:

- Terminates the trigger or procedure in which it was raised and undoes any actions performed (directly or indirectly) by it.
- Returns an error message to the calling application. In ISQL, the error message appears on the screen, unless output is redirected.

Exceptions may be trapped and handled with a WHEN statement in a stored procedure or trigger.

Examples This ISQL statement creates the exception, UNKNOWN_EMP_ID:

```
CREATE EXCEPTION UNKNOWN_EMP_ID 'Invalid employee number
   or project id.';
```

The following statement from a stored procedure raises the previously created exception when SQLCODE -530 is set, which is a violation of a FOREIGN KEY constraint:

```
. . .
WHEN SQLCODE -530 DO
EXCEPTION UNKNOWN_EMP_ID;
. . .
```

See Also ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions, see the *Using Firebird*— Error trapping and handling (ch. 25 p. 549).

CREATE GENERATOR

Declares a generator to the database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `CREATE GENERATOR` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name for the generator

Description CREATE GENERATOR declares a generator to the database and sets its starting value to zero. A generator is a sequential number that can be automatically inserted in a column with the GEN_ID() function. A generator is often used to ensure a unique value in a PRIMARY KEY, such as an invoice number, that must uniquely identify the associated row.

A database can contain any number of generators. Generators are global to the database, and can be used and updated in any transaction. Firebird does not assign duplicate generator values across transactions.
You can use SET GENERATOR to set or change the value of an existing generator when writing triggers, procedures, or SQL statements that call GEN_ID().

Example The following ISQL script fragment creates the generator, EMPNO_GEN, and the trigger, CREATE_EMPNO. The trigger uses the generator to produce sequential numeric keys, incremented by 1, for the NEW.EMPNO column:

```
CREATE GENERATOR EMPNO_GEN;
COMMIT;
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
BEFORE INSERT POSITION 0
AS BEGIN
NEW.EMPNO = GEN_ID(EMPNO_GEN, 1);
END
SET TERM ; !!
```

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE TRIGGER in ISQL. Use SET TERM before CREATE TRIGGER to specify a terminator other than a semicolon. After CREATE TRIGGER, include another SET TERM to change the terminator back to a semicolon.

See Also GEN_ID( ), SET GENERATOR, DROP GENERATOR

CREATE INDEX

Creates an index on one or more columns in a table.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax
```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX index
          ON table (col [, col ...]);
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
UNIQUE	Prevents insertion or updating of duplicate values into indexed columns
ASC[ENDING]	Sorts columns in ascending order, the default order if none is specified
DESC[ENDING]	Sorts columns in descending order
index	Unique name for the index
table	Name of the table on which the index is defined
col	Column in *table* to index

Description Creates an index on one or more columns in a table. Use CREATE INDEX to improve speed of data access. Using an index for columns that appear in a WHERE clause may improve search performance.

Important You cannot index blob columns or arrays.

A UNIQUE index cannot be created on a column or set of columns that already contains duplicate or NULL values.

ASC and DESC specify the order in which an index is sorted. For faster response to queries that require sorted values, use the index order that matches the query's ORDER BY clause. Both an ASC and a DESC index can be created on the same column or set of columns to access data in different orders.

Tip To improve index performance, use SET STATISTICS to recompute index selectivity, or rebuild the index by making it inactive, then active with sequential calls to ALTER INDEX.

Examples The following ISQL statement creates a unique index:

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

The next ISQL statement creates a descending index:

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

The following ISQL statement creates a two-column index:

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

See Also ALTER INDEX, DROP INDEX, SELECT, SET STATISTICS

CREATE PROCEDURE

Creates a stored procedure, its input and output parameters, and its actions.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
CREATE PROCEDURE name
            [(param <pdatatype> [, param <pdatatype> ...])]
            [RETURNS param <pdatatype> [, param <pdatatype> ...])]
            AS <procedure_body> [terminator]

 <pdatatype> = BLOB | <datatype>

<procedure_body> =
   [<variable_declaration_list>]
   <block>

<variable_declaration_list> =
   DECLARE VARIABLE var <datatype>;
   [DECLARE VARIABLE var <datatype>; ...]

<block> =
BEGIN
   <compound_statement>
   [<compound_statement> ...]
END

<compound_statement> = <block> | statement;
```

```
<datatype> = SMALLINT
| INTEGER
| FLOAT
| DOUBLE PRECISION
| {DECIMAL | NUMERIC} [(precision [, scale])]
| {DATE | TIME | TIMESTAMP)
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[(int)] [CHARACTER SET charname]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]
```

Argument	Description
name	Name of the procedure. Must be unique among procedure, table, and view names in the database
param datatype	Input parameters that the calling program uses to pass values to the procedure: *param*: Name of the input parameter, unique for variables in the procedure *datatype*: An Firebird datatype
RETURNS *param datatype*	Output parameters that the procedure uses to return values to the calling program: *param*: Name of the output parameter, unique for variables within the procedure *datatype*: An Firebird datatype The procedure returns the values of output parameters when it reaches a SUSPEND statement in the procedure body
AS	Reserved word that separates the procedure header and the procedure body

Argument	Description
DECLARE VARIABLE *var datatype*	Declares local variables used only in the procedure; must be preceded by DECLARE VARIABLE and followed by a semicolon (;). *var* is the name of the local variable, unique for variables in the procedure.
statement	Any single statement in Firebird procedure and trigger language; must be followed by a semicolon (;) except for BEGIN and END statements
terminator	Terminator defined by SET TERM • Signifies the end of the procedure body; • Used only in ISQL

Description CREATE PROCEDURE defines a new stored procedure to a database. A stored procedure is a self-contained program written in Firebird procedure and trigger language, and stored as part of a database's metadata. Stored procedures can receive input parameters from and return values to applications.

Firebird procedure and trigger language includes all SQL data manipulation statements and some powerful extensions, including IF … THEN … ELSE, WHILE … DO, FOR SELECT … DO, exceptions, and error handling. There are two types of procedures:

- *Select* procedures that an application can use in place of a table or view in a SELECT statement. A select procedure must be defined to return one or more values, or an error will result.
- *Executable* procedures that an application can call directly, with the EXECUTE PROCEDURE statement. An executable procedure need not return values to the calling program.

A stored procedure is composed of a *header* and a *body*.

The procedure header contains:

- The *name* of the stored procedure, which must be unique among procedure and table names in the database.
- An optional list of *input parameters* and their datatypes that a procedure receives from the calling program.

- RETURNS followed by a list of *output parameters* and their datatypes if the procedure returns values to the calling program.

The procedure body contains:

- An optional list of *local variables* and their datatypes.
- A *block* of statements in Firebird procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there may be many levels of nesting.

Important Because each statement in a stored procedure body must be terminated by a semicolon (except the BEGIN and END statements), you must define a different symbol to terminate the CREATE PROCEDURE statement in ISQL. Use SET TERM before CREATE PROCEDURE to specify a terminator other than a semicolon. After the CREATE PROCEDURE statement, include another SET TERM to change the terminator back to a semicolon.

Firebird does not allow database changes that affect the behavior of an existing stored procedure (for example, DROP TABLE or DROP EXCEPTION). To see all procedures defined for the current database or the text and parameters of a named procedure, use the ISQL internal commands SHOW PROCEDURES or SHOW PROCEDURE *procedure*.

Firebird procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- SQL operators and expressions, including generators and UDFs that are linked with the database.
- Extensions to SQL, including assignment statements, control-flow statements, context variables (for triggers), event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for stored procedures. For a complete description of each statement, see PSQL-Firebird Procedural Language on page 222.

TABLE 2–5 Language extensions for stored procedures

Statement	Description
BEGIN … END	Defines a block of statements that executes as one • The BEGIN reserved word starts the block; the END reserved word terminates it • Neither should end with a semicolon
variable = *expression*	Assignment statement: assigns the value of *expression* to *variable*, a local variable, input parameter, or output parameter
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text
EXCEPTION *exception_name*	Raises the named exception: an exception is a user-defined error that returns an error message to the calling application unless handled by a WHEN statement
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name,* with the listed input arguments, returning values in the listed output arguments following RETURNING_VALUES; input and output arguments must be local variables
EXIT	Jumps to the final END statement in the procedure.
FOR *select_statement* DO *compound_statement*	Repeats the statement or block following DO for every qualifying row retrieved by *select_statement* *select_statement* is like a normal SELECT statement
compound_statement	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END

TABLE 2–5 Language extensions for stored procedures (*continued*)

Statement	Description
IF (*condition*) THEN *compound_statement* [ELSE *compound_statement*]	Tests *condition*, and if it is TRUE, performs the statement or block following THEN; otherwise, performs the statement or block following ELSE, if present *condition*: a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an event name
SUSPEND	In a SELECT procedure: • Suspends execution of procedure until the current result set is read by the calling application • Returns output values, if any, to the calling application • Not recommended for executable procedures

TABLE 2–5 Language extensions for stored procedures (*continued*)

Statement	Description
WHILE (*condition*) DO *compound_statement*	While *condition* is TRUE, keep performing *compound_statement* • *Tests condition*, andperforms *compound_statement* if *condition* is TRUE • Repeats this sequence until *condition* is no longer TRUE
WHEN {*error* [, *error* …] | ANY} DO *compound_statement*	Error-handling statement: when one of the specified errors occurs, performs *compound_statement* • WHEN statements, if present, must come at the end of a block, just before END • *error*: EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number* • ANY: Handles any errors

The stored procedure and trigger language does not include many of the statement types available in DSQL or ESQL. The following statement types are not supported in triggers or stored procedures:

- Data definition language statements: CREATE, ALTER, DROP, DECLARE EXTERNAL FUNCTION, and DECLARE FILTER
- Transaction control statements: SET TRANSACTION, COMMIT, ROLLBACK
- Dynamic SQL statements: PREPARE, DESCRIBE, EXECUTE
- CONNECT/DISCONNECT, and sending SQL statements to another database
- GRANT/REVOKE
- SET GENERATOR
- EVENT INIT/EVENT WAIT
- BEGIN DECLARATION/END DECLARE SECTION

- BASED ON
- WHENEVER
- DECLARE CURSOR
- OPEN
- FETCH

Examples The following procedure, SUB_TOT_BUDGET, takes a department number as its input parameter, and returns the total, average, minimum, and maximum budgets of departments with the specified HEAD_DEPT.

```
/* Compute total, average, smallest, and largest department budget.
*Parameters:
* department id
*
*Returns:
* total budget
* average budget
* min budget
* max budget */
SET TERM !! ;
CREATE PROCEDURE SUB_TOT_BUDGET (HEAD_DEPT CHAR(3))
RETURNS (tot_budget DECIMAL(12, 2), avg_budget DECIMAL(12, 2),
min_budget DECIMAL(12, 2), max_budget DECIMAL(12, 2))
AS
BEGIN
SELECT SUM(BUDGET), AVG(BUDGET), MIN(BUDGET), MAX(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
```

```
  INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
  EXIT;
  END !!
SET TERM ; !!
```

See Also **ALTER EXCEPTION**, **ALTER PROCEDURE**, **CREATE EXCEPTION**, **DROP EXCEPTION**, **DROP PROCEDURE**, **EXECUTE PROCEDURE**, **SELECT**

For more information on creating and using procedures, see *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).
For a complete description of the statements in procedure and trigger language, see chapter 3, PSQL-Firebird Procedural Language (p. 222).

CREATE ROLE

Creates a role.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `CREATE ROLE rolename;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
rolename	Name associated with the role; must be unique among role names in the database

Description Roles created with CREATE ROLE can be granted privileges just as users can. These roles can be granted to users, who then inherit the privilege list that has been granted to the role. Users must specify the role at connect time. Use GRANT to grant privileges (ALL, SELECT, INSERT, UPDATE, DELETE, EXECUTE, REFERENCES) to a role and to grant a role to users. Use REVOKE to revoke them.

Example The following statement creates a role called “administrator.”

```
CREATE ROLE administrator;
```

See Also GRANT, REVOKE, DROP ROLE

CREATE SHADOW

Creates one or more duplicate, in-sync copies of a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
    'filespec' [LENGTH [=] int [PAGE[S]]]
    [<secondary_file>];

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

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
set_num	Positive integer that designates a shadow set to which all subsequent files listed in the statement belong
AUTO	Specifies the default access behavior for databases in the event no shadow is available • All attachments and accesses succeed • Deletes all references to the shadow and detaches the shadow file
MANUAL	Specifies that database attachments and accesses fail until a shadow becomes available, *or* until all references to the shadow are removed from the database

Argument	Description
CONDITIONAL	Creates a new shadow, allowing shadowing to continue if the primary shadow becomes unavailable or if the shadow replaces the database due to disk failure
'*filespec*'	Explicit path name and file name for the shadow file; must be a local filesystem and must not include a node name or be on a neworked filesystem
LENGTH [=] *int* [PAGE[S]]	Length in database pages of an additional shadow file; page size is determined by the page size of the database itself
secondary_file	Specifies the length of a primary or secondary shadow file; use for primary file only if defining a secondary file in the same statement
STARTING [AT [PAGE]] *int*	Starting page number at which a secondary shadow file begins

Description CREATE SHADOW is used to guard against loss of access to a database by establishing one or more copies of the database on secondary storage devices. Each copy of the database consists of one or more shadow files, referred to as a *shadow set*. Each shadow set is designated by a unique positive integer.

Disk shadowing has three components:

- A database to shadow.
- The RDB$FILES system table, which lists shadow files and other information about the database.
- A shadow set, consisting of one or more shadow files.

When CREATE SHADOW is issued, a shadow is established for the database most recently attached by an application. A shadow set can consist of one or multiple files. In case of disk failure, the database administrator (DBA) activates the disk shadow so that it can take the place of the database. If CONDITIONAL is specified, then when the DBA activates the disk shadow to replace an actual database, a new shadow is established for the database.

If a database is larger than the space available for a shadow on one disk, use the *secondary_file* option to define multiple shadow files. Multiple shadow files can be spread over several disks.

Tip To add a secondary file to an existing disk shadow, drop the shadow with DROP SHADOW and use CREATE SHADOW to recreate it with the desired number of files.

Examples The following ISQL statement creates a single, automatic shadow file for **employee.gdb**:

```
CREATE SHADOW 1 AUTO 'employee.shd';
```

The next ISQL statement creates a conditional, single, automatic shadow file for **employee.gdb**:

```
CREATE SHADOW 2 CONDITIONAL 'employee.shd' LENGTH 1000;
```

The following ISQL statements create a multiple-file shadow set for the **employee.gdb** database. The first statement specifies starting pages for the shadow files; the second statement specifies the number of pages for the shadow files.

```
CREATE SHADOW 3 AUTO
'employee.sh1'
FILE 'employee.sh2'
STARTING AT PAGE 1000
FILE 'employee.sh3'
STARTING AT PAGE 2000;
CREATE SHADOW 4 MANUAL 'employee.sdw'
LENGTH 1000
FILE 'employee.sh1'
LENGTH 1000
FILE 'employee.sh2';
```

See Also DROP SHADOW

See also *Using Firebird*— Database shadows (ch. 20 p. 379).

CREATE TABLE

Creates a new table in an existing database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec']
    (<col_def> [, <col_def> | <tconstraint> ...]);

<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
    [DEFAULT {literal | NULL | USER}]
    [NOT NULL]
    [<col_constraint>]
[COLLATE collation]

<datatype> =
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[<array_dim>]
    | (DATE | TIME | TIMESTAMP}[<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
        [<array_dim>] [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
        [VARYING] [(int)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
        [CHARACTER SET charname]
| BLOB [(seglen [, subtype])]

<array_dim> = [[x:]y [, [x:]y ...]]

<expr> = a valid SQL expression that results in a single value

<col_constraint> = [CONSTRAINT constraint]
    { UNIQUE
    | PRIMARY KEY
    | REFERENCES other_table [(other_col [, other_col …])]
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>)}
```

```
<tconstraint> = [CONSTRAINT constraint]
   {{PRIMARY KEY | UNIQUE} (col [, col ...])
   | FOREIGN KEY (col [, col ...])
      REFERENCES other_table [(other_col [, other_col ...])]
         [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
         [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>)}

<search_condition> = <val> <operator> {<val> | (<select_one>)}
   | <val> [NOT] BETWEEN <val> AND <val>
   | <val> [NOT] LIKE <val> [ESCAPE <val>]
   | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
   | <val> IS [NOT] NULL
   | <val> {>= | <=} <val>
   | <val> [NOT] {= | < | >} <val>
   | {ALL | SOME | ANY} (<select_list>)
   | EXISTS (<select_expr>)
   | SINGULAR (<select_expr>)
   | <val> [NOT] CONTAINING <val>
   | <val> [NOT] STARTING [WITH] <val>
   | (<search_condition>)
   | NOT <search_condition>
   | <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>
```

```
<val> = { col [<array_dim>] | :variable
    | <constant> | <expr> | <function>
    | udf ([<val> [, <val> ...]])
    | NULL | USER | RDB$DB_KEY | ? }
[COLLATE collation]

<constant> = num | 'string' | _charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
    | SUM ([ALL] <val> | DISTINCT <val>)
    | AVG ([ALL] <val> | DISTINCT <val>)
    | MAX ([ALL] <val> | DISTINCT <val>)
    | MIN ([ALL] <val> | DISTINCT <val>)
    | CAST (<val> AS <datatype>)
    | UPPER (<val>)
| GEN_ID (generator, <val>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column; returns exactly one value.

<select_list> = SELECT on a single column; returns zero or more values.

<select_expr> = SELECT on a list of values; returns zero or more
values.
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Notes on the CREATE TABLE statement

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array VARCHAR(6)[5,5]
```

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

```
my_array INTEGER[10:20]
```

- In SQL and ISQL, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and ISQL, *val* cannot be a variable.
- You cannot specify a COLLATE clause for blob columns.
- *expr* is any complex SQL statement or equation that produces a single value.
- When you need to qualify a constant string with a specific character set, prefix the character set name with an underscore, to indicate the name is not a regular SQL identifier. You must use, for example:

```
_WIN1252 'This is my string';
```

Argument	Description
table	Name for the table; must be unique among table and procedure names in the database
EXTERNAL [FILE] '*filespec*'	Declares that data for the table under creation resides in a table or file outside the database; *filespec* is the complete file specification of the external file or table
col	Name for the table column; unique among column names in the table
datatype	SQL datatype for the column; see xxxxxxx

Argument	Description
COMPUTED [BY] (*expr*)	Specifies that the value of the column's data is calculated from *expr* at runtime and is therefore not allocated storage space in the database • *expr* can be any arithmetic expression valid for the datatypes in the expression • Any columns referenced in *expr* must exist before they can be used in *expr* • *expr* cannot reference blob columns • *expr* must return a single value, and cannot return an array
domain	Name of an existing domain
DEFAULT	Specifies a default column value that is entered when no other entry is made; possible values are: • *literal*: Inserts a specified string, numeric value, or date value • NULL: Enters a NULL value • USER: Enters the user name of the current user. Column must be of compatible text type to use the default Defaults set at column level override defaults set at the domain level.
CONSTRAINT *constraint*	Name of a column or table constraint; the constraint name must be unique within the table
constraint_def	Specifies the kind of column constraint; valid options are UNIQUE, PRIMARY KEY, CHECK, and REFERENCES

Argument	Description
REFERENCES	Specifies that the column values are derived from column values in another table; if you do not specify column names, Firebird looks for a column with the same name as the referencing column in the referenced table
ON DELETE | ON UPDATE	Used with REFERENCES: Changes a foreign key whenever the referenced primary key changes; valid options are: • [Default] NO ACTION: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks • CASCADE: For ON DELETE, deletes the corresponding foreign key; for ON UPDATE, updates the corresponding foreign key to the new value of the primary key • SET NULL: Sets all the columns of the corresponding foreign key to NULL • SET DEFAULT: Sets every column of the corresponding foreign key to whatever default value is in effect when the referential integrity constraint is defined. When the default for a foreign column changes *after* the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint.
CHECK *search_condition*	An attempt to enter a new value in the column fails if the value does not meet the *search_condition*
COLLATE *collation*	Establishes a default sorting behavior for the column; see xxxxxx for more information

Description CREATE TABLE establishes a new table, its columns, and integrity constraints in an existing database. The user who creates a table is the table's owner and has all privileges for it, including the ability to GRANT privileges to other users, triggers, and stored procedures.

- CREATE TABLE supports several options for defining columns:
- Local columns specify the name and datatype for data entered into the column.
- Computed columns are based on an expression. Column values are computed each time the table is accessed. If the datatype is not specified, Firebird calculates an appropriate one. Columns referenced in the expression must exist before the column can be defined.
- Domain-based columns inherit all the characteristics of a domain, but the column definition can include a new default value, a NOT NULL attribute, additional CHECK constraints, or a collation clause that overrides the domain definition. It can also include additional column constraints.
- The datatype specification for a CHAR, VARCHAR, or BLOB text column definition can include a CHARACTER SET clause to specify a particular character set for the single column. Otherwise, the column uses the default database character set. If the database character set is changed, all columns subsequently defined have the new character set, but existing columns are not affected. For a complete list of character sets recognized by Firebird, see xxxxxx
- If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.
- The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and NCHAR text datatypes. Choice of collation order is restricted to those supported for the column's given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the datatype definition. For a complete list of collation orders recognized by Firebird, see xxxxxx
- NOT NULL is an attribute that prevents the entry of NULL or unknown values in column. NOT NULL affects all INSERT and UPDATE operations on a column.

Important A DECLARE TABLE must precede CREATE TABLE in embedded applications if the same SQL program both creates a table and inserts data in the table.

- The EXTERNAL FILE option creates a table whose data resides in an external file, rather than in the Firebird database. Use this option to:
- Define a Firebird table composed of data from an external source, such as data in files managed by other operating systems or in non-database applications.
- Transfer data to an existing Firebird table from an external file.

Referential integrity constraints

- You can define integrity constraints at the time you create a table. These constraints are rules that validate data entries by enforcing column-to-table and table-to-table relationships. They span all transactions that access the database and are automatically maintained by the system. CREATE TABLE supports the following integrity constraints:
- A PRIMARY KEY is one or more columns whose collective contents are guaranteed to be unique. A PRIMARY KEY column must also define the NOT NULL attribute. A table can have only one primary key.
- UNIQUE keys ensure that no two rows have the same value for a specified column or ordered set of columns. A unique column must also define the NOT NULL attribute. A table can have one or more UNIQUE keys. A UNIQUE key can be referenced by a FOREIGN KEY in another table.
- Referential constraints (REFERENCES) ensure that values in the specified columns (known as the *foreign key*) are the same as values in the referenced UNIQUE or PRIMARY KEY columns in another table. The UNIQUE or PRIMARY KEY columns in the referenced table must be defined before the REFERENCES constraint is added to the secondary table. REFERENCES has ON DELETE and ON UPDATE clauses that define the action on the foreign key when the referenced primary key is updated or deleted. The values for ON UPDATE and ON DELETE are as follows:

Action specified	Effect on foreign key
NO ACTION	[Default] The foreign key does not change. This may cause the primary key update or delete to fail due to referential integrity checks.
CASCADE	The corresponding foreign key is updated or deleted as appropriate to the new value of the primary key.
SET DEFAULT	Every column of the corresponding foreign key is set to its default value. If the default value of the foreign key is not found in the primary key, the update or delete on the primary key fails. The default value is the one in effect when the referential integrity constraint was defined. When the default for a foreign key column is changed after the referential integrity constraint is set up, the change does not have an effect on the default value used in the referential integrity constraint.
SET NULL	Every column of the corresponding foreign key is set to NULL.

- You can create a FOREIGN KEY reference to a table that is owned by someone else only if that owner has explicitly granted you REFERENCES privilege on that table. Any user who updates your foreign key table must have REFERENCES or SELECT privileges on the referenced primary key table.
- CHECK constraints enforce a *search_condition* that must be true for inserts or updates to the specified table. *search_condition* can require a combination or range of values or can compare the value entered with data in other columns.

Note Specifying USER as the value for a *search_condition* references the login of the user who is attempting to write to the referenced table.

- Creating PRIMARY KEY and FOREIGN KEY constraints requires exclusive access to the database.
- For unnamed constraints, the system assigns a unique constraint name stored in the RDB$RELATION_CONSTRAINTS system table.

Note Constraints are not enforced on expressions.

Examples The following ISQL statement creates a simple table with a PRIMARY KEY:

```
CREATE TABLE COUNTRY (COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
  CURRENCY VARCHAR(10) NOT NULL);
```

The next ISQL statement creates both a column-level and a table-level UNIQUE constraint:

```
CREATE TABLE STOCK (
   MODEL SMALLINT NOT NULL UNIQUE,
   MODELNAME CHAR(10) NOT NULL,
   ITEMID INTEGER NOT NULL,
   CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

The following ISQL statemnent illustrates table-level PRIMARY KEY, FOREIGN KEY, and CHECK constraints. The PRIMARY KEY constraint is based on three columns. This example also illustrates creating an array column of VARCHAR.

```
CREATE TABLE JOB (
JOB_CODE JOBCODE NOT NULL,
JOB_GRADE JOBGRADE NOT NULL,
JOB_COUNTRY COUNTRYNAME NOT NULL,
JOB_TITLE VARCHAR(25) NOT NULL,
MIN_SALARY SALARY NOT NULL,
MAX_SALARY SALARY NOT NULL,
JOB_REQUIREMENT BLOB(400,1),
LANGUAGE_REQ VARCHAR(15) [5],
PRIMARY KEY (JOB_CODE, JOB_GRADE, JOB_COUNTRY),
FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY),
CHECK (MIN_SALARY < MAX_SALARY));
```

In the next example, the F2 column in table T2 is a foreign key that references table T1 through T1's primary key P1. When a row in T1 changes, that change propagates to all affected rows in table T2. When a row in T1 is deleted, all affected rows in the F2 column of table T2 are set to NULL.

```
CREATE TABLE T1 (P1 INTEGER NOT NULL PRIMARY KEY);
```

```
CREATE TABLE T2 (F2 INTEGER FOREIGN KEY REFERENCES T1.P1
  ON UPDATE CASCADE
  ON DELETE SET NULL);
```

The next ISQL statement creates a table with a calculated column:

```
CREATE TABLE SALARY_HISTORY (
EMP_NO EMPNO NOT NULL,
CHANGE_DATE DATE DEFAULT 'NOW' NOT NULL,
UPDATER_ID VARCHAR(20) NOT NULL,
OLD_SALARY SALARY NOT NULL,
PERCENT_CHANGE DOUBLE PRECISION
DEFAULT 0
NOT NULL
CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
NEW_SALARY COMPUTED BY
(OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
PRIMARY KEY (EMP_NO, CHANGE_DATE, UPDATER_ID),
  FOREIGN KEY (EMP_NO) REFERENCES EMPLOYEE (EMP_NO));
```

In the following ISQL statement the first column retains the default collating order for the database's default character set. The second column has a different collating order, and the third column definition includes a character set and a collating order.

```
CREATE TABLE BOOKADVANCE (
   BOOKNO CHAR(6),
   TITLE CHAR(50) COLLATE ISO8859_1,
   EUROPUB CHAR(50) CHARACTER SET ISO8859_1 COLLATE FR_FR);
```

See Also **CREATE DOMAIN**, **DECLARE TABLE**, **GRANT**, **REVOKE**

For more information, refer to *Using Firebird*— Tables (ch. 17 p. 313) and Managing Security in ch. 22 of the same volume.

CREATE TRIGGER

Creates a trigger, including when it fires, and what actions it performs.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
CREATE TRIGGER name FOR table
            [ACTIVE | INACTIVE]
            {BEFORE | AFTER}
            {DELETE | INSERT | UPDATE}
            [POSITION number]
            AS <trigger_body> terminator

<trigger_body> = [<variable_declaration_list>] <block>

<variable_declaration_list> =
   DECLARE VARIABLE variable <datatype>;
   [DECLARE VARIABLE variable <datatype>; ...]

<block> =
BEGIN
   <compound_statement>
   [<compound_statement> ...]
END

 <datatype> = SMALLINT
 | INTEGER
 | FLOAT
 | DOUBLE PRECISION
 | {DECIMAL | NUMERIC} [(precision [, scale])]
 | {DATE | TIME | TIMESTAMP)
 | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
 [(int)] [CHARACTER SET charname]
   | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]

<compound_statement> = <block> | statement;
```

Argument	Description
name	Name of the trigger; must be unique in the database
table	Name of the table or view that causes the trigger to fire when the specified operation occurs on the table or view
ACTIVE|INACTIVE	Optional. Specifies trigger action at transaction end: • ACTIVE: [Default] Trigger takes effect • INACTIVE: Trigger does not take effect
BEFORE|AFTER	Required. Specifies whether the trigger fires: • BEFORE: Before associated operation • AFTER: After associated operation Associated operations are DELETE, INSERT, or UPDATE
DELETE|INSERT |UPDATE	Specifies the table operation that causes the trigger to fire
POSITION *number*	Specifies firing order for triggers before the same action or after the same action; *number* must be an integer between 0 and 32,767, inclusive. • Lower-number triggers fire first • Default: 0 = first trigger to fire • Triggers for a table need not be consecutive; triggers on the same action with the same position number will fire in random order.

Argument	Description
DECLARE VARIABLE *var datatype*	Declares local variables used only in the trigger. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;). • *var*: Local variable name, unique in the trigger • *datatype*: The datatype of the local variable
statement	Any single statement in Firebird procedure and trigger language; each statement except BEGIN and END must be followed by a semicolon (;)
terminator	Terminator defined by the SET TERM statement; signifies the end of the trigger body. Used in ISQL only.

Description CREATE TRIGGER defines a new trigger to a database. A trigger is a self-contained program associated with a table or view that automatically performs an action when a row in the table or view is inserted, updated, or deleted.

A trigger is never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation automatically execute, or *fire*. Triggers defined for UPDATE on non-updatable views fire even if no update occurs.

A trigger is composed of a *header* and a *body*.

The trigger header contains:

- A *trigger name*, unique within the database, that distinguishes the trigger from all others.
- A *table name*, identifying the table with which to associate the trigger.
- *Statements* that determine when the trigger fires.

The trigger body contains:

- An optional list of *local variables* and their datatypes.
- A *block* of statements in Firebird procedure and trigger language, bracketed by BEGIN and END. These statements are performed when the trigger fires. A block can itself include other blocks, so that there may be many levels of nesting.

Important Because each statement in the trigger body must be terminated by a semicolon, you must define a different symbol to terminate the trigger body itself. In ISQL, include a SET TERM statement before CREATE TRIGGER to specify a terminator other than a semicolon. After the body of the trigger, include another SET TERM to change the terminator back to a semicolon.

A trigger is associated with a table. The table owner and any user granted privileges to the table automatically have rights to execute associated triggers.

Triggers can be granted privileges on tables, just as users or procedures can be granted privileges. Use the GRANT statement, but instead of using TO *username*, use TO TRIGGER *trigger_name*. Triggers' privileges can be revoked similarly using REVOKE.

When a user performs an action that fires a trigger, the trigger will have privileges to perform its actions if one of the following conditions is true:

- The trigger has privileges for the action.
- The user has privileges for the action.

Firebird procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- SQL operators and expressions, including generators and UDFs that are linked with the calling application.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for triggers. For a complete description of each statement, see chapter 3, PSQL-Firebird Procedural Language (p. 222).

For discussion of programming triggers, see Triggers, Coding the body of the code module and Implementing stored procedures and triggers in *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).

TABLE 2–6 Language extensions for triggers

Statement	Description
BEGIN … END	Defines a block of statements that executes as one • The BEGIN reserved word starts the block; the END reserved word terminates it • Neither should be followed by a semicolon
variable = *expression*	Assignment statement that assigns the value of *expression* to *variable*, a local variable, input parameter, or output parameter
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text
EXCEPTION *exception_name*	Raises the named exception; an exception is a user-defined error that returns an error message to the calling application unless handled by a WHEN statement
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name,* with the listed input arguments • Returns values in the listed output arguments following RETURNING_VALUES • Input and output arguments must be local variables.
FOR *select_statement* DO *compound_statement*	Repeats the statement or block following DO for every qualifying row retrieved by *select_statement*
select_statement	A normal SELECT statement
compound_statement	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END

TABLE 2–6 Language extensions for triggers (*continued*)

Statement	Description
IF (*condition*) THEN *compound_statement* [ELSE *compound_statement*]	Tests *condition*, and if it is TRUE, performs the statement or block following THEN; otherwise, performs the statement or block following ELSE, if present
condition	A Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an event name
WHILE (*condition*) DO *compound_statement*	While *condition* is TRUE, keep performing *compound_statement* • *Tests condition*, andperforms *compound_statement* if *condition* is TRUE • Repeats this sequence until *condition* is no longer TRUE
WHEN {*error* [, *error* …] | ANY} DO *compound_statement*	Error-handling statement. When one of the specified errors occurs, performs *compound_statement*. WHEN statements, if present, must come at the end of a block, just before END • ANY: Handles any errors
error	EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number*

The stored procedure and trigger language does not include many of the statement types available in DSQL or ESQL. The following statement types are not supported in triggers or stored procedures:

- Data definition language statements: CREATE, ALTER, DROP, DECLARE EXTERNAL FUNCTION, and DECLARE FILTER
- Transaction control statements: SET TRANSACTION, COMMIT, ROLLBACK
- Dynamic SQL statements: PREPARE, DESCRIBE, EXECUTE
- CONNECT/DISCONNECT, and sending SQL statements to another database
- GRANT/REVOKE
- SET GENERATOR
- EVENT INIT/EVENT WAIT
- BEGIN DECLARATION/END DECLARE SECTION
- BASED ON
- WHENEVER
- DECLARE CURSOR
- OPEN
- FETCH

Examples The following trigger, SAVE_SALARY_CHANGE, makes correlated updates to the SALARY_HISTORY table when a change is made to an employee's salary in the EMPLOYEE table:

```
  SET TERM !! ;
  CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
  AFTER UPDATE AS
  BEGIN
  IF (OLD.SALARY <> NEW.SALARY) THEN
  INSERT INTO SALARY_HISTORY
  (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
  VALUES (OLD.EMP_NO, 'now', USER, OLD.SALARY,
  (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
  END !!
SET TERM ; !!
```

The following trigger, SET_CUST_NO, uses a generator to create unique customer numbers when a new customer record is inserted in the CUSTOMER table:

```
  SET TERM !! ;
  CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
  BEFORE INSERT AS
  BEGIN
  NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
  END !!
SET TERM ; !!
```

The following trigger, POST_NEW_ORDER, posts an event named “new_order” whenever a new record is inserted in the SALES table:

```
  SET TERM !! ;
  CREATE TRIGGER POST_NEW_ORDER FOR SALES
  AFTER INSERT AS
  BEGIN
  POST_EVENT 'new_order';
  END !!
SET TERM ; !!
```

The following four fragments of trigger headers demonstrate how the POSITION option determines trigger firing order:

```
CREATE TRIGGER A FOR accounts
   BEFORE UPDATE
   POSITION 5 ... /*Trigger body follows*/

CREATE TRIGGER B FOR accounts
   BEFORE UPDATE
   POSITION 0 ... /*Trigger body follows*/

CREATE TRIGGER C FOR accounts
   AFTER UPDATE
   POSITION 5 ... /*Trigger body follows*/

CREATE TRIGGER D FOR accounts
   AFTER UPDATE
   POSITION 3 ... /*Trigger body follows*/
```

When this update takes place:

```
UPDATE accounts SET account_status = 'on_hold'
   WHERE account_balance <0;
```

The triggers fire in this order:

1. Trigger B fires.
2. Trigger A fires.

3 The update occurs.

4 Trigger D fires.

5 Trigger C fires.

See Also **ALTER EXCEPTION**, **ALTER TRIGGER**, **CREATE EXCEPTION**, **CREATE PROCEDURE**, **DROP EXCEPTION**, **DROP TRIGGER**, **EXECUTE PROCEDURE**

For a complete description of each statement, see chapter 3, PSQL-Firebird Procedural Language (p. 222).

For discussion of programming triggers, see Triggers, Coding the body of the code module and Implementing stored procedures and triggers in *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).

CREATE VIEW

Creates a new view of data from one or more tables.

Availability DSQL ESQL ISQL ~~PSQL~~

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name for the view; must be unique among all view, table, and procedure names in the database
view_col	Names the columns for the view • Column names must be unique among all column names in the view • Required if the view includes columns based on expressions; otherwise optional • Default: Column name from the underlying table
select	Specifies the selection criteria for rows to be included in the view
WITH CHECK OPTION	Prevents INSERT or UPDATE operations on an updatable view if the INSERT or UPDATE violates the search condition specified in the WHERE clause of the view's SELECT clause

Description CREATE VIEW describes a view of data based on one or more underlying tables in the database. The rows to return are defined by a SELECT statement that lists columns from the source tables. Only the view definition is stored in the database; a view does not directly represent physically stored data. It is possible to perform select, project, join, and union operations on views as if they were tables.

The user who creates a view is its owner and has all privileges for it, including the ability to GRANT privileges to other users, roles, triggers, views, and stored procedures. A user may have privileges to a view without having access to its base tables. When creating views:

- A read-only view requires SELECT privileges for any underlying tables.
- An updatable view requires ALL privileges to the underlying tables.

The *view_col* option ensures that the view always contains the same columns and that the columns always have the same view-defined names.

View column names correspond in order and number to the columns listed in the SELECT clause, so specify *all* view column names or none.

A *view_col* definition can contain one or more columns based on an expression that combines the outcome of two columns. The expression must return a single value, and cannot return an array or array element. If the view includes an expression, the *view-column* option is required.

Note Any columns used in the value expression must exist before the expression can be defined.

A SELECT statement clause cannot include the ORDER BY clause.
When SELECT * is used rather than a column list, order of display is based on the order in which columns are stored in the base table.
WITH CHECK OPTION enables Firebird to verify that a row added to or updated in a view is able to be seen through the view before allowing the operation to succeed. Do not use WITH CHECK OPTION for read-only views.

Note Although it is possible to create a view based on the output of a selectable stored procedure, it adds an unnecessary layer of dependency to do so. Using the output set of a stored procedure joined to a table, another view or another stored procedure is also theoretically possible but, in practice, it causes more trouble than it saves. With such complex requirements, it is almost invariably best to define the entire output within a selectable stored procedure.

A view is updatable if:

- It is a subset of a single table or another updatable view.
- All base table columns excluded from the view definition allow NULL values.
- The view's SELECT statement does not contain subqueries, a DISTINCT predicate, a HAVING clause, aggregate functions, joined tables, user-defined functions, or stored procedures.

If the view definition does not meet these conditions, it is considered read-only.

Note Read-only views can be updated by using a combination of user-defined referential constraints, triggers, and unique indexes.

Examples The following ISQL statement creates an updatable view:

```
CREATE VIEW SNOW_LINE (CITY, STATE, SNOW_ALTITUDE) AS
SELECT CITY, STATE, ALTITUDE
FROM CITIES
     WHERE ALTITUDE > 5000;
```

The next ISQL statement uses a nested query to create a view:

```
CREATE VIEW RECENT_CITIES AS
SELECT STATE, CITY, POPULATION
FROM CITIES WHERE STATE IN
        (SELECT STATE FROM STATES WHERE STATEHOOD > '1-JAN-1850');
```

In an updatable view, the WITH CHECK OPTION prevents any inserts or updates through the view that do not satisfy the WHERE clause of the CREATE VIEW SELECT statement:

```
CREATE VIEW HALF_MILE_CITIES AS
SELECT CITY, STATE, ALTITUDE
FROM CITIES
WHERE ALTITUDE > 2500
     WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in the view would prevent the following insertion:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
  VALUES ('Chicago', 'Illinois', 250);
```

On the other hand, the following UPDATE would be permitted:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
  VALUES ('Truckee', 'California', 2736);
```

The WITH CHECK OPTION clause does not allow updates through the view which change the value of a row so that the view cannot retrieve it. For example, the WITH CHECK OPTION in the HALF_MILE_CITIES view prevents the following update:

```
UPDATE HALF_MILE_CITIES
SET ALTITUDE = 2000
  WHERE STATE = 'NY';
```

The next ISQL statement creates a view that joins two tables, and so is read-only:

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

See Also CREATE TABLE, DROP VIEW, GRANT, INSERT, REVOKE, SELECT, UPDATE

For a complete discussion, see *Using Firebird*— Views (ch. 19 p. 363).

DECLARE CURSOR

Defines a cursor for a table by associating a name with the set of rows specified in a SELECT statement.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax (ESQL only):

```
DECLARE cursor CURSOR FOR <select> [FOR UPDATE OF <col> [, <col>...]];
```

Blob form: See DECLARE CURSOR (BLOB)

Argument	Description
cursor	Name for the cursor
select	Determines which rows to retrieve. SQL only
FOR UPDATE OF *col* [, *col* …]	Enables UPDATE and DELETE of specified column for retrieved rows
statement_id	SQL statement name of a previously prepared statement, which in this case must be a SELECT statement. DSQL only

Description DECLARE CURSOR defines the set of rows that can be retrieved using the cursor it names. It is the first member of a group of table cursor statements that must be used in sequence.

<*select*> specifies a SELECT statement that determines which rows to retrieve. This SELECT statement cannot include INTO or ORDER BY clauses.

The FOR UPDATE OF clause is necessary for updating or deleting rows using the WHERE CURRENT OF clause with UPDATE and DELETE.

A *cursor* is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and releases system resources

Examples The following ESQL statement declares a cursor with a search condition:

```
EXEC SQL
DECLARE C CURSOR FOR
SELECT CUST_NO, ORDER_STATUS
FROM SALES
     WHERE ORDER_STATUS IN ('open', 'shipping');
```

The next DSQL statement declares a cursor for a previously prepared statement, QUERY1:

```
DECLARE Q CURSOR FOR QUERY1
```

See Also CLOSE, DECLARE CURSOR (BLOB), FETCH, OPEN, PREPARE, SELECT

DECLARE CURSOR (BLOB)

Declares a blob cursor for read or insert.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax
```
DECLARE cursor CURSOR FOR
            {READ BLOB column FROM table
            | INSERT BLOB column INTO table}
            [FILTER [FROM subtype] TO subtype]
            [MAXIMUM_SEGMENT length];
```

Argument	Description
cursor	Name for the blob cursor
column	Name of the blob column
table	Table name
READ BLOB	Declares a read operation on the blob
INSERT BLOB	Declares a write operation on the blob
[FILTER [FROM *subtype*] TO *subtype*]	Specifies optional blob filters used to translate a blob from one user-specified format to another; *subtype* determines which filters are used for translation
MAXIMUM_ SEGMENT *length*	Length of the local variable to receive the blob data after a FETCH operation

Description Declares a cursor for reading or inserting blob data. A blob cursor can be associated with only one blob column.

To read partial blob segments when a host-language variable is smaller than the segment length of a blob, declare the blob cursor with the MAXIMUM_SEGMENT clause. If *length* is less than the blob segment, FETCH returns *length* bytes. If the same or greater, it returns a full segment (the default).

Examples The following ESQL statement declares a READ BLOB cursor and uses the MAXIMUM_SEGMENT option:

```
EXEC SQL
DECLARE BC CURSOR FOR
  READ BLOB JOB_REQUIREMENT FROM JOB MAXIMUM_SEGMENT 40;
```

The next ESQL statement declares an INSERT BLOB cursor:

```
EXEC SQL
DECLARE BC CURSOR FOR
INSERT BLOB JOB_REQUIREMENt INTO JOB;
```

See Also CLOSE (BLOB), FETCH (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

DECLARE EXTERNAL FUNCTION

Declares an existing user-defined function (UDF) to a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
    [, datatype | CSTRING (int) ...]]
RETURNS {datatype [BY VALUE] | [BY DESCRIPTOR] | CSTRING (int)}
    | PARAMETER int_pos} [FREE_IT]
ENTRY_POINT 'entryname'
MODULE_NAME 'modulename';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Note Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the FREE_IT reserved word in order to free the allocated memory.

Argument	Description
name	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT reserved word
datatype	Datatype of an input or return parameter • All input parameters are passed to a UDF by reference • Return parameters can be passed by value • Cannot be an array element
CSTRING (*int*)	Specifies a UDF that returns a null-terminated string *int* bytes in length; CSTRING (*n*) allocates an array of *n* bytes and holds a string no longer than *n*-1 bytes
int_pos	Position of one of the arguments, starting with 1
RETURNS	Specifies the return value of a function
BY VALUE	Specifies that a return value should be passed by value rather than by reference
BY DESCRIPTOR	Specifies that input arguments and return values are passed as Firebird data type descriptors

Argument	Description
FREE_IT	Frees memory of the return value after the UDF finishes running • Use only if the memory is allocated dynamically in the UDF • See also *Language Reference*, Chapter 5
'*entryname*'	Quoted string specifying the name of the UDF in the source code as stored in the UDF library
'*modulename*'	Quoted file name identifying the library that contains the UDF; the placement of the library in the file system must meet one of the following criteria: • The library is in *ib_install_dir***/UDF** • The library in a directory other than *ib_install_dir***/UDF** and the complete pathname to the directory, including a drive letter in the case of a Windows server, is listed in the Firebird configuration file. • See the UDF chapter of the *Developer's Guide* for more about how Firebird finds the library

Description DECLARE EXTERNAL FUNCTION provides information about a UDF to a database: where to find it, its name, the input parameters it requires, and the single value it returns. Each UDF in a library must be declared once to each database where it will be used. As long as the entry point and module name do not change, there is no need to redelcare a UDF, even if the function itself is modified.

entryname is the actual name of the function as stored in the UDF library. It does not have to match the name of the UDF as stored in the database.

The module name does not need to include a path. However, the module must either be placed in **ib_install_dir/UDF** or must be listed in the Firebird configuration file.

To specify a location for UDF libraries in a configuration file, enter a line of the following form for Windows platforms:

```
EXTERNAL_FUNCTION_DIRECTORY D:\Mylibraries\Firebird
```

For Linux/UNIX, the path does not include a drive letter:

```
EXTERNAL_FUNCTION_DIRECTORY /usr/Mylibraries/Firebird
```

Note that beginning with Firebird 1, you *must* list the path in the Firebird configuration file if it is other than *ib_install_dir***/UDF**. A path name is no longer useful in the DECLARE EXTERNAL FUNCTION statement.

The Firebird configuration file is called **ibconfig** on Windows machines, **isc_config** on Linux/UNIX machines.

Examples The following ISQL statement declares the TOPS() UDF to a database:

```
DECLARE EXTERNAL FUNCTION TOPS
   CHAR(256), INTEGER, BLOB
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'te1' MODULE_NAME 'tm1.dll';
```

This example does not need the FREE_IT reserved word because only CSTRINGs, CHAR, and VARCHAR return types require memory allocation.

The next example declares the LOWER() UDF and frees the memory allocated for the return value:

```
DECLARE EXTERNAL FUNCTION LOWER VARCHAR(256)
RETURNS CSTRING(256) FREE_IT
  ENTRY POINT 'fn_lower' MODULE_NAME 'udflib.dll';
```

In the next example, an external function is declared for a function that passes and receives arguments BY DESCRIPTOR:

```
DECLARE EXTERNAL FUNCTION SNVL
VARCHAR(100) BY DESCRIPTOR, VARCHAR(100) BY DESCRIPTOR,
VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
ENTRY_POINT 'SNVL' MODULE_NAME 'FBUDF';
```

See Also DROP EXTERNAL FUNCTION

For more information about writing and using UDFs, see *Using Firebird*— Working with UDFs and Blob Filters (ch. 26 p. 572).

For declarations of the UDFs in the ib_udf and FBUDF libraries, see User-defined Functions on page 257 in chapter 6.

DECLARE FILTER

Declares an existing blob filter to a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax
```
DECLARE FILTER filter
        INPUT_TYPE subtype OUTPUT_TYPE subtype
        ENTRY_POINT 'entryname' MODULE_NAME 'modulename';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
filter	Name of the filter; must be unique among filter names in the database
INPUT_TYPE *subtype*	Specifies the blob subtype from which data is to be converted
OUTPUT_TYPE *subtype*	Specifies the blob subtype into which data is to be converted
'*entryname*'	Quoted string specifying the name of the blob filter as stored in a linked library
'*modulename*'	Quoted file specification identifying the object module in which the filter is stored

Description DECLARE FILTER provides information about an existing blob filter to the database: where to find it, its name, and the blob subtypes it works with. A blob filter is a user-written program that converts data stored in blob columns from one subtype to another.

INPUT_TYPE and OUTPUT_TYPE together determine the behavior of the blob filter. Each filter declared to the database should have a unique combination of INPUT_TYPE and OUTPUT_TYPE integer values. Firebird provides a built-in type of 1, for handling text. User-defined types must be expressed as negative values.

entryname is the name of the blob filter stored in the library. When an application uses a blob filter, it calls the filter function with this name.

Example The following ISQL statement declares a blob filter:

```
DECLARE FILTER DESC_FILTER
INPUT_TYPE 1
OUTPUT_TYPE -4
ENTRY_POINT 'desc_filter'
MODULE_NAME 'FILTERLIB';
```

See Also DROP FILTER

For more information about BLOB subtypes and instructions on writing blob filters, see *Using Firebird*— BLOB filters (ch. 26 p. 596) and associated topics in that section.

DECLARE STATEMENT

Identifies dynamic SQL statements before they are prepared and executed in an embedded program.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `DECLARE <statement> STATEMENT;`

Argument	Description
statement	Name of a SQL variable for a user-supplied SQL statement to prepare and execute at runtime

Description DECLARE STATEMENT names a SQL variable for a user-supplied SQL statement to prepare and execute at run time. DECLARE STATEMENT is not executed, so it does not produce run-time errors. The statement provides internal documentation.

Example The following ESQL statement declares Q1 to be the name of a string for preparation and execution.

```
EXEC SQL
DECLARE Q1 STATEMENT;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

DECLARE TABLE

Describes the structure of a table to the preprocessor, **gpre**, before it is created with CREATE TABLE.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax DECLARE *table* TABLE (<*table_def*>);

Argument	Description
table	Name of the table; table names must be unique within the database
table_def	Definition of the table; for complete table definition syntax, see CREATE TABLE

Description DECLARE TABLE causes **gpre** to store a table description. You must use it if you both create and populate a table with data in the same program. If the declared table already exists in the database or if the declaration contains syntax errors, **gpre** returns an error.

When a table is referenced at run time, the column descriptions and datatypes are checked against the description stored in the database. If the table description is not in the database and the table is not declared, or if column descriptions and datatypes do not match, the application returns an error.

DECLARE TABLE can include an existing domain in a column definition, but must give the complete column description if the domain is not defined at compile time.

DECLARE TABLE cannot include integrity constraints and column attributes, even if they are present in a subsequent CREATE TABLE statement.

Important DECLARE TABLE cannot appear in a program that accesses multiple databases.

Example The following ESQL statements declare and create a table:

```
EXEC SQL
DECLARE STOCK TABLE
(MODEL SMALLINT,
MODELNAME CHAR(10),
ITEMID INTEGER);
EXEC SQL
CREATE TABLE STOCK
(MODEL SMALLINT NOT NULL UNIQUE,
MODELNAME CHAR(10) NOT NULL,
ITEMID INTEGER NOT NULL,
CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

See Also CREATE DOMAIN, CREATE TABLE

DELETE

Removes rows in a table or in the active set of a cursor.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax **ESQL and DSQL form:**

Important Omit the terminating semicolon for DSQL.

```
DELETE [TRANSACTION transaction] FROM table
   {[WHERE <search_condition>] | WHERE CURRENT OF cursor};

<search_condition> = Search condition as specified in SELECT.
```

ISQL form:

```
DELETE FROM TABLE [WHERE <search_condition>];
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed; SQL only
table	Name of the table from which to delete rows
WHERE *search_condition*	Search condition that specifies the rows to delete; without this clause, DELETE affects all rows in the specified table or view
WHERE CURRENT OF *cursor*	Specifies that the current row in the active set of *cursor* is to be deleted

DELETE specifies one or more rows to delete from a table or updatable view. DELETE is one of the database privileges controlled by the GRANT and REVOKE statements.
The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the DELETE operation. The TRANSACTION clause is not available in DSQL or ISQL.
For searched deletions, the optional WHERE clause can be used to restrict deletions to a subset of rows in the table.

Important Without a WHERE clause, a searched delete removes all rows from a table.

When performing a positioned delete with a cursor, the WHERE CURRENT OF clause must be specified to delete one row at a time from the active set.

Examples The following ISQL statement deletes all rows in a table:

```
DELETE FROM EMPLOYEE_PROJECT;
```

The next ESQL statement is a searched delete in an embedded application. It deletes all rows where a host-language variable equals a column value.

```
EXEC SQL
DELETE FROM SALARY_HISTORY
  WHERE EMP_NO = :emp_num;
```

The following ESQL statements use a cursor and the WHERE CURRENT OF option to delete rows from CITIES with a population less than the host variable, *min_pop*. They declare and open a cursor that finds qualifying cities, fetch rows into the cursor, and delete the current row pointed to by the cursor.

```
EXEC SQL
DECLARE SMALL_CITIES CURSOR FOR
SELECT CITY, STATE
FROM CITIES
  WHERE POPULATION < :min_pop;

EXEC SQL
  OPEN SMALL_CITIES;

EXEC SQL
FETCH SMALL_CITIES INTO :cityname, :statecode;
WHILE (!SQLCODE)
{EXEC SQL
DELETE FROM CITIES
WHERE CURRENT OF SMALL_CITIES;
EXEC SQL
FETCH SMALL_CITIES INTO :cityname, :statecode;}
EXEC SQL
  CLOSE SMALL_CITIES;
```

See Also **DECLARE CURSOR**, **FETCH**, **GRANT**, **OPEN**, **REVOKE**, **SELECT**

For more information about using cursors, see the *Embedded SQL Guide* (EmbedSQL.pdf) of the InterBase® 6 documentation set, obtainable from Borland.

DESCRIBE

Provides information about columns that are retrieved by a dynamic SQL (DSQL) statement, or information about dynamic parameters that statement passes.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `DESCRIBE [OUTPUT | INPUT] statement {INTO | USING} SQL DESCRIPTOR xsqlda;`

Argument	Description
OUTPUT	[Default] Indicates that column information should be returned in the XSQLDA
INPUT	Indicates that dynamic parameter information should be stored in the XSQLDA
statement	A previously defined alias for the statement to DESCRIBE. • Use PREPARE to define aliases
{INTO | USING} SQL DESCRIPTOR *xsqlda*	Specifies the XSQLDA to use for the DESCRIBE statement

Description DESCRIBE has two uses:

- As a *describe output* statement, DESCRIBE stores into an XSQLDA a description of the columns that make up the select list of a previously prepared statement. If the PREPARE statement included an INTO clause, it is unnecessary to use DESCRIBE as an output statement.
- As a *describe input* statement, DESCRIBE stores into an XSQLDA a description of the dynamic parameters that are in a previously prepared statement.

DESCRIBE is one of a group of statements that process DSQL statements.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution

Statement	Purpose
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

Separate DESCRIBE statements must be issued for input and output operations. The INPUT reserved word must be used to store dynamic parameter information.

Important When using DESCRIBE for output, if the value returned in the *sqld* field in the XSQLDA is larger than the *sqln* field, you must:

- Allocate more storage space for XSQLVAR structures.
- Reissue the DESCRIBE statement.

Note The same XSQLDA structure can be used for input and output if desired.

Example The following ESQL statement retrieves information about the output of a SELECT statement:

```
EXEC SQL
  DESCRIBE Q INTO xsqlda
```

The next ESQL statement stores information about the dynamic parameters passed with a statement to be executed:

```
EXEC SQL
DESCRIBE INPUT Q2 USING SQL DESCRIPTOR xsqlda;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

For more information about ESQL programming and the XSQLDA descriptor, see the *Embedded SQL Guide* of the InterBase® 6 documentation set, available from Borland.

DISCONNECT

Detaches an application from a database.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

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

Argument	Description
ALL DEFAULT	Either reserved word detaches all open databases
dbhandle	Previously declared database handle specifying a database to detach

Description DISCONNECT closes a specific database identified by a database handle or all databases, releases resources used by the attached database, zeroes database handles, commits the default transaction if the **gpre -manual** option is not in effect, and returns an error if any non-default transaction is not committed.

Before using DISCONNECT, commit or roll back the transactions affecting the database to be detached. To reattach to a database closed with DISCONNECT, reopen it with a CONNECT statement.

Examples The following ESQL statements close all databases:

```
EXEC SQL
  DISCONNECT DEFAULT;

EXEC SQL
  DISCONNECT ALL;
```

The next ESQL statements close the databases identified by their handles:

```
EXEC SQL
  DISCONNECT DB1;
```

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

See Also COMMIT, CONNECT, ROLLBACK, SET DATABASE

DROP DATABASE

Deletes the currently attached database.

Availability ~~DSQL~~ ~~ESQL~~ ISQL ~~PSQL~~

Syntax `DROP DATABASE;`

Description DROP DATABASE deletes the currently attached database, including any associated secondary, shadow, and log files. Dropping a database deletes any data it contains.

A database can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Example The following ISQL statement deletes the current database:

```
DROP DATABASE;
```

See Also ALTER DATABASE, CREATE DATABASE

DROP DOMAIN

Deletes a domain from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP DOMAIN` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing domain

Description DROP DOMAIN removes an existing domain definition from a database.

If a domain is currently used in any column definition in the database, the DROP operation fails. To prevent failure, use ALTER TABLE to delete the columns based on the domain before executing DROP DOMAIN.
A domain may be dropped by its creator, the SYSDBA, and, on Linux./UNIX, the root user or another user with root privileges..

Example The following ISQL statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE DOMAIN

DROP EXCEPTION

Deletes an exception from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP EXCEPTION` *name*

Argument	Description
name	Name of an existing exception message

Description DROP EXCEPTION removes an exception from a database.

Exceptions used in existing procedures and triggers cannot be dropped.

Tip In ISQL, SHOW EXCEPTION displays a list of exceptions' *dependencies*, the procedures and triggers that use the exceptions.

An exception can be dropped by its creator, the SYSDBA user, and any user with operating system root privileges.

Example This ISQL statement drops an exception:

```
DROP EXCEPTION UNKNOWN_EMP_ID;
```

See Also ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER

DROP EXTERNAL FUNCTION

Removes a user-defined function (UDF) declaration from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP EXTERNAL FUNCTION` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing UDF

Description DROP EXTERNAL FUNCTION deletes a UDF declaration from a database. Dropping a UDF declaration from a database does *not* remove it from the corresponding UDF library, but it does make the UDF inaccessible from the database. Once the definition is dropped, any applications that depend on the UDF will return run-time errors.

A UDF can be dropped by its declarer, the SYSDBA user, and, on Linux./UNIX, the root user or another user with root privileges.

Example This ISQL statement drops a UDF:

```
DROP EXTERNAL FUNCTION TOPS;
```

See Also DECLARE EXTERNAL FUNCTION

DROP FILTER

Removes a blob filter declaration from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP FILTER name;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing blob filter

Description DROP FILTER removes a blob filter declaration from a database. Dropping a blob filter declaration from a database does *not* remove it from the corresponding blob filter library, but it does make the filter inaccessible from the database. Once the definition is dropped, any applications that depend on the filter will return run-time errors.

DROP FILTER fails and returns an error if any processes are using the filter.
A filter can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges..

Example This ISQL statement drops a blob filter:

```
DROP FILTER DESC_FILTER;
```

See Also DECLARE FILTER

DROP GENERATOR

Removes a generator from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP EXTERNAL FUNCTION name;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing generator

Description DROP GENERATOR deletes the named generator from a database, provided there are no procedures or triggers dependent upon it.

A generator can be dropped by its creator, the SYSDBA user or, on Linux./UNIX, the root user or another user with root privileges.

Example This ISQL statement drops a generator named GEN_pkPROJECT:

```
DROP GENERATOR GEN_pkPROJECT;
```

See Also CREATE GENERATOR

DROP INDEX

Removes an index from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP INDEX` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index

Description DROP INDEX removes a user-defined index from a database.

An index can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Important You cannot drop system-defined indexes, such as those for UNIQUE, PRIMARY KEY, and FOREIGN KEY.

An index in use is not dropped until it is no longer in use.

Example The following ISQL statement deletes an index:

```
DROP INDEX MINSALX;
```

See Also **ALTER INDEX**, **CREATE INDEX**

For more information about integrity constraints and system-defined indexes, see *Using Firebird*— Tables (ch. 17 p. 313). For a discussion of indexing and related issues, see Indexes in ch. 18 of the same volume.

DROP PROCEDURE

Deletes an existing stored procedure from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax DROP PROCEDURE *name*

Argument	Description
name	Name of an existing stored procedure

Description DROP PROCEDURE removes an existing stored procedure definition from a database.

Procedures used by other procedures, triggers, or views cannot be dropped. Procedures currently in use cannot be dropped.

Tip In ISQL, SHOW PROCEDURE displays a list of procedures' *dependencies*, the procedures, triggers, exceptions, and tables that use the procedures.

A procedure can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Example The following ISQL statement deletes a procedure:

```
DROP PROCEDURE GET_EMP_PROJ;
```

See Also ALTER PROCEDURE, CREATE PROCEDURE, EXECUTE PROCEDURE

DROP ROLE

Deletes a role from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP ROLE` *`rolename`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
rolename	Name of an existing role

Description DROP ROLE deletes a role that was previously created using CREATE ROLE. Any privileges that users acquired or granted through their membership in the role are revoked.

A role can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Example The following ISQL statement deletes a role from its database:

```
DROP ROLE administrator;
```

See Also CREATE ROLE, GRANT, REVOKE

DROP SHADOW

Deletes a shadow from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP SHADOW` *`set_num`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
set_num	Positive integer to identify an existing shadow set

Description DROP SHADOW deletes a shadow set and detaches from the shadowing process. The ISQL SHOW DATABASE command can be used to see shadow set numbers for a database.

A shadow can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Example The following ISQL statement deletes a shadow set from its database:

```
DROP SHADOW 1;
```

See Also CREATE SHADOW

DROP TABLE

Removes a table from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP TABLE` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing table

Description DROP TABLE removes a table's data, metadata, and indexes from a database. It also drops any triggers that reference the table.

A table referenced in a SQL expression, a view, integrity constraint, or stored procedure cannot be dropped. A table used by an active transaction is not dropped until it is free.

Note When used to drop an external table, DROP TABLE only removes the table definition from the database. The external file is not deleted.

A table can be dropped by its creator, the SYSDBA user and, on Linux./UNIX, the root user or another user with root privileges.

Example The following ESQL statement drops a table:

```
EXEC SQL
DROP TABLE COUNTRY;
```

See Also ALTER TABLE, CREATE TABLE

DROP TRIGGER

Deletes an existing user-defined trigger from a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP TRIGGER` *name*

Argument	Description
name	Name of an existing trigger

Description DROP TRIGGER removes a user-defined trigger definition from the database. System-defined triggers, such as those created for CHECK constraints, cannot be dropped. Use ALTER TABLE to drop the CHECK clause that defines the trigger.

Triggers used by an active transaction cannot be dropped until the transaction is terminated.
A trigger can be dropped by its creator, the SYSDBA user or, on Linux./UNIX, the root user or another user with root privileges.

Tip To inactivate a trigger temporarily, use ALTER TRIGGER and specify INACTIVE in the header.

Example The following ISQL statement drops a trigger:

```
DROP TRIGGER POST_NEW_ORDER;
```

See Also ALTER TRIGGER, CREATE TRIGGER

DROP VIEW

Removes a view definition from the database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `DROP VIEW name;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing view definition to drop

Description DROP VIEW enables a view's creator to remove a view definition from the database if the view is not used in another view, stored procedure, or CHECK constraint definition.

A view can be dropped by its creator, the SYSDBA user or, on Linux./UNIX, the root user or another user with root privileges.

Example The following ISQL statement removes a view definition:

```
DROP VIEW PHONE_LIST;
```

See Also CREATE VIEW

END DECLARE SECTION

Identifies the end of a host-language variable declaration section.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `END DECLARE SECTION;`

Description END DECLARE SECTION is used in ESQL applications to identify the end of host-language variable declarations for variables used in subsequent SQL statements.

Example The following ESQL statements declare a section, and single host-language variable:

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED_ON EMPLOYEE.SALARY salary;
EXEC SQL
END DECLARE SECTION;
```

See Also BASED ON, BEGIN DECLARE SECTION

EVENT INIT

Registers interest in one or more events with the Firebird event manager.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax
```
EVENT INIT request_name [dbhandle]
          ('string' | :variable [, 'string' | :variable ...]);
```

Argument	Description
request_name	Application event handle
dbhandle	Specifies the database to examine for occurrences of the events; if omitted, *dbhandle* defaults to the database named in the most recent SET DATABASE statement
'*string*'	Unique name identifying an event associated with *event_name*
:*variable*	Host-language character array containing a list of event names to associate with

Description EVENT INIT is the first step in the Firebird two-part synchronous event mechanism:

1. EVENT INIT registers an application's interest in an event.
2. EVENT WAIT causes the application to wait until notified of the event's occurrence.

EVENT INIT registers an application's interest in a list of events in parentheses. The list should correspond to events posted by stored procedures or triggers in the database. If an application registers interest in multiple events with a single EVENT INIT, then when one of those events occurs, the application must determine which event occurred.
Events are posted by a POST_EVENT call within a stored procedure or trigger.
The event manager keeps track of events of interest. At commit time, when an event occurs, the event manager notifies interested applications.

Example The following ESQL statement registers interest in an event:

```
EXEC SQL
EVENT INIT ORDER_WAIT EMPDB ('new_order');
```

See Also **CREATE PROCEDURE**, **CREATE TRIGGER**, **EVENT WAIT**, **SET DATABASE**

For more information about events, see How events work, Handling events on a client and related topics in *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).

EVENT WAIT

Causes an application to wait until notified of an event's occurrence.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `EVENT WAIT request_name;`

Argument	Description
request_name	Application event handle declared in a previous EVENT INIT statement

Description EVENT WAIT is the second step in the Firebird two-part synchronous event mechanism. After a program registers interest in an event, EVENT WAIT causes the process running the application to sleep until the event of interest occurs.

Examples The following ESQL statements register an application event name and indicate the program is ready to receive notification when the event occurs:

```
EXEC SQL
  EVENT INIT ORDER_WAIT EMPDB ('new_order');

EXEC SQL
  EVENT WAIT ORDER_WAIT;
```

See Also EVENT INIT

For more information about events, see How events work, Handling events on a client and related topics in *Using Firebird*— Programming on Firebird Server (ch. 25 p. 494).

EXECUTE

Executes a previously prepared dynamic SQL (DSQL) statement.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax `EXECUTE [TRANSACTION transaction] statement [USING SQL DESCRIPTOR xsqlda] [INTO SQL DESCRIPTOR xsqlda];`

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
statement	Alias of a previously prepared statement to execute
USING SQL DESCRIPTOR	Specifies that values corresponding to the prepared statement's parameters should be taken from the specified XSQLDA
INTO SQL DESCRIPTOR	Specifies that return values from the executed statement should be stored in the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description EXECUTE carries out a previously prepared DSQL statement. It is one of a group of statements that process DSQL statements.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

Before a statement can be executed, it must be prepared using the PREPARE statement. The statement can be any SQL data definition, manipulation, or transaction management statement. Once it is prepared, a statement can be executed any number of times.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE operation.

USING DESCRIPTOR enables EXECUTE to extract a statement's parameters from an XSQLDA structure previously loaded with values by the application. It need only be used for statements that have dynamic parameters.

INTO DESCRIPTOR enables EXECUTE to store return values from statement execution in a specified XSQLDA structure for application retrieval. It need only be used for DSQL statements that return values.

Note If an EXECUTE statement provides both a USING DESCRIPTOR clause and an INTO DESCRIPTOR clause, then two XSQLDA structures must be provided.

Example The following ESQL statement executes a previously prepared DSQL statement:

```
EXEC SQL
  EXECUTE DOUBLE_SMALL_BUDGET;
```

The next ESQL statement executes a previously prepared statement with parameters stored in an XSQLDA:

```
EXEC SQL
  EXECUTE Q USING DESCRIPTOR xsqlda;
```

The following ESQL statement executes a previously prepared statement with parameters in one XSQLDA, and produces results stored in a second XSQLDA:

```
EXEC SQL
  EXECUTE Q USING DESCRIPTOR xsqlda_1 INTO DESCRIPTOR xsqlda_2;
```

See Also **DESCRIBE**, **EXECUTE IMMEDIATE**, **PREPARE**

For more information about ESQL programming and the XSQLDA, see the *Embedded SQL Guide* (EmbedSQL.pdf) available from Borland.

EXECUTE IMMEDIATE

Prepares a dynamic SQL (DSQL) statement,
executes it once, and discards it.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax
```
EXECUTE IMMEDIATE [TRANSACTION transaction]
            {:variable | 'string'} [USING SQL DESCRIPTOR xsqlda];
```

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
:*variable*	Host variable containing the SQL statement to execute
'*string*'	A string literal containing the SQL statement to execute
USING SQL DESCRIPTOR	Specifies that values corresponding to the statement's parameters should be taken from the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description EXECUTE IMMEDIATE prepares a DSQL statement stored in a host-language variable or in a literal string, executes it once, and discards it. To prepare and execute a DSQL statement for repeated use, use PREPARE and EXECUTE instead of EXECUTE IMMEDIATE.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE IMMEDIATE operation.

The SQL statement to execute must be stored in a host variable or be a string literal. It can contain any SQL data definition statement or data manipulation statement that does not return output.

USING DESCRIPTOR enables EXECUTE IMMEDIATE to extract the values of a statement's parameters from an XSQLDA structure previously loaded with appropriate values.

Example The following ESQL statement prepares and executes a statement in a host variable:

```
EXEC SQL
EXECUTE IMMEDIATE :insert_date;
```

See Also **DESCRIBE**, **EXECUTE IMMEDIATE**, **PREPARE**

For more information about ESQL programming and the XSQLDA, see the *Embedded SQL Guide*.

EXECUTE PROCEDURE

Calls a stored procedure.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax ESQL form:

```
EXECUTE PROCEDURE [TRANSACTION transaction]
   name [:param [[INDICATOR]:indicator]]
      [, :param [[INDICATOR]:indicator] ...]
   [RETURNING_VALUES :param [[INDICATOR]:indicator]
      [, :param [[INDICATOR]:indicator] ...]];
```

DSQL form:

```
EXECUTE PROCEDURE TRANSACTION transaction
name [param [, param ...]]
[RETURNING_VALUES param [, param ...]]
```

ISQL form:

```
EXECUTE PROCEDURE name [param [, param ...]]
```

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs
name	Name of an existing stored procedure in the database
param	Input or output parameter; can be a host variable or a constant
RETURNING_VALUES: *param*	Host variable which takes the values of an output parameter
[INDICATOR] :*indicator*	Host variable for indicating NULL or unknown values

Description EXECUTE PROCEDURE calls the specified stored procedure. If the procedure requires input parameters, they are passed as host-language variables or as constants. If a procedure returns output parameters to a SQL program, host variables must be supplied in the RETURNING_VALUES clause to hold the values returned.

In ISQL, do not use the RETURN clause or specify output parameters. ISQL will automatically display return values.

Note In DSQL, an EXECUTE PROCEDURE statement requires an input descriptor area if it has input parameters and an output descriptor area if it has output parameters.

In ESQL, input parameters and return values may have associated indicator variables for tracking NULL values. Indicator variables are integer values that indicate unknown or NULL values of return values.
An indicator variable that is less than zero indicates that the parameter is unknown or NULL. An indicator variable that is zero or greater indicates that the associated parameter is known and not NULL.

Examples The following ESQL statement demonstrates how the executable procedure, DEPT_BUDGET, is called from ESQL with literal parameters:

```
EXEC SQL
  EXECUTE PROCEDURE DEPT_BUDGET 100 RETURNING_VALUES :sumb;
```

The next ESQL statement calls the same procedure using a host variable instead of a literal as the input parameter:

```
EXEC SQL
  EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNING_VALUES :sumb;
```

See Also ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE

For more information about indicator variables, see the *Embedded SQL Guide* (EmbedSQL.pdf) from the InterBase® 6 documentation set, available from Borland.

EXTRACT( )

Extracts date and time information from DATE, TIME, and TIMESTAMP values.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `EXTRACT (part FROM value)`

Argument	Description
part	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, WEEKDAY, or YEARDAY; see Table 2–7 for datatypes and ranges of values
value	DATE, TIME, or TIMESTAMP value

Description The *value* passed to the EXTRACT() expression must be a DATE, a TIME, or a TIMESTAMP. Extracting a part that does not exist in a datatype results in an error. For example, a statement such as `EXTRACT (YEAR from aTime);` would fail.

The datatype of *part* depends on which part is extracted.

TABLE 2–7 EXTRACT( ) date and time parts

Part extracted	Datatype	Range
YEAR	SMALLINT	1-9999
MONTH	SMALLINT	1-12
DAY	SMALLINT	1-31
HOUR	SMALLINT	1-23
MINUTE	SMALLINT	1-59
SECOND	DECIMAL(6,4)	0-59.9999
WEEKDAY	SMALLINT	0-6 (0 = Sunday, 1 = Monday, etc)
YEARDAY	SMALLINT	0-365 (where 0 is January 1 and 364 is December 31, except in a leap year)

Examples `EXTRACT(HOUR FROM StartTime);`

Use an explicit `CAST()` if *value* is not a DATE, TIME, or TIMESTAMP datatype. This statement results in an error:

```
SELECT EXTRACT(YEAR FROM '1999-1-1')
      FROM RDB$DATABASE;
```

This statement is valid:

```
SELECT EXTRACT(YEAR FROM CAST('1999-1-1' AS DATE))
      FROM RDB$DATABASE;
```

Error:

```
SELECT EXTRACT(YEAR FROM 'NOW')
      FROM RDB$DATABASE;
```

Valid:

```
SELECT EXTRACT(YEAR FROM CAST('NOW' AS DATE))
      FROM RDB$DATABASE;
```

Valid:

```
SELECT EXTRACT(YEAR FROM CURRENT_TIMESTAMP)
      FROM RDB$DATABASE;
```

FETCH

Retrieves the next available row from the active set of an opened cursor.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax **ESQL form:**

```
FETCH cursor
   [INTO :hostvar [[INDICATOR] :indvar]
   [, :hostvar [[INDICATOR] :indvar] ...]];
```

DSQL form:

```
FETCH cursor {INTO | USING} SQL DESCRIPTOR xsqlda
```

Blob form: See FETCH (BLOB).

Argument	Description
cursor	Name of the opened cursor from which to fetch rows
:*hostvar*	A host-language variable for holding values retrieved with the FETCH • Optional if FETCH gets rows for DELETE or UPDATE • Required if row is displayed before DELETE or UPDATE
:*indvar*	Indicator variable for reporting that a column contains an unknown or NULL value
[INTO|USING] SQL DESCRIPTOR	Specifies that values should be returned in the specified XSQLDA
xsqlda	XSQLDA host-language variable

Description FETCH retrieves one row at a time into a program from the active set of a cursor. The first FETCH operates on the first row of the active set. Subsequent FETCH statements advance the cursor sequentially through the active set one row at a time until no more rows are found and SQLCODE is set to 100.

A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declare the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieve the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieve the current row from the active set, starting with the first row; subsequent FETCH statements advance the cursor through the set
4	CLOSE	Close the cursor and release system resources

The number, size, datatype, and order of columns in a FETCH must be the same as those listed in the query expression of its matching DECLARE CURSOR statement. If they are not, the wrong values can be assigned.

Examples The following ESQL statement fetches a column from the active set of a cursor:

```
EXEC SQL
FETCH PROJ_CNT INTO :department, :hcnt;
```

See Also CLOSE, DECLARE CURSOR, DELETE, FETCH (BLOB), OPEN

For more information about cursors and XSQLDA, see the *Embedded SQL Guide* (EmbedSQL.pdf) from the InterBase® 6 documentation set, available from Borland.

FETCH (BLOB)

Retrieves the next available segment of a blob column and places it in the specified local buffer.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax

```
FETCH cursor INTO
          :<buffer> [[INDICATOR] :segment_length];
```

Argument	Description
cursor	Name of an open blob cursor from which to retrieve segments
:*buffer*	Host-language variable for holding segments fetched from the blob column; user must declare the buffer before fetching segments into it
INDICATOR	Optional reserved word indicating that a host-language variable for indicating the number of bytes returned by the FETCH follows
:*segment_length*	Host-language variable used to indicate he number of bytes returned by the FETCH

Description FETCH retrieves the next segment from a blob and places it into the specified buffer.

The host variable, *segment_length,* indicates the number of bytes fetched. This is useful when the number of bytes fetched is smaller than the host variable, for example, when fetching the last portion of a blob. FETCH can return two SQLCODE values:

- SQLCODE = 100 indicates that there are no more blob segments to retrieve.
- SQLCODE = 101 indicates that a partial segment was retrieved and placed in the local buffer variable.

Note To ensure that a host variable buffer is large enough to hold a blob segment buffer during FETCH operations, use the SEGMENT option of the BASED ON statement.

To ensure that a host variable buffer is large enough to hold a blob segment buffer during FETCH operations, use the SEGMENT option of the BASED ON statement.

Example The following code, from an ESQL application, performs a blob FETCH:

```
while (SQLCODE != 100)
{
EXEC SQL
OPEN BLOB_CUR USING :blob_id;
EXEC SQL
FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;
while (SQLCODE !=100 || SQLCODE == 101)
{
blob_segment{blob_seg_len + 1] = '\0';
printf("%*.*s",blob_seg_len,blob_seg_len,blob_segment);
blob_segment{blob_seg_len + 1] = ' ';
EXEC SQL
FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;
}
. . .
}
```

See Also BASED ON, CLOSE (BLOB), DECLARE CURSOR (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

FIRST(m) SKIP(n)

Optional sub-clauses to a SELECT statement—FIRST(m) produces the first m rows of an ordered output set, discarding the remainder, while SKIP(n) causes the first n rows of an ordered output set to be discarded and begins the output at row n+1. Both FIRST and SKIP are optional. If both are present, they interact in the output.

Example The following example shows a query which uses both clauses:

```
SELECT FIRST (30) SKIP (10) Field1, Field2, Field3
FROM ATable
ORDER BY Field1;
```

It generates a set of 30 rows, being the 11th to 40th rows in the ordered set.

GEN_ID( )

Produces a system-generated integer value.

Availability DSQL ESQL ISQL PSQL

Syntax `GEN_ID (generator, step)`

Argument	Description
generator	Name of an existing generator
step	Integer or expression specifying the increment for increasing or decreasing the current generator value. Values can range from $-(2^{63})$ to $2^{63} - 1$

Description The GEN_ID() function:

1. Increments the current value of the specified generator by *step*.
2. Returns the new value of the specified generator.

GEN_ID() is useful for automatically producing unique values that can be inserted into a UNIQUE or PRIMARY KEY column. To insert a generated number in a column, write a trigger, procedure, or SQL statement that calls GEN_ID().

Note A generator is initially created with CREATE GENERATOR. By default, the value of a generator begins at zero. It can be set to a different value with SET GENERATOR.

Examples The following ISQL trigger definition includes a call to GEN_ID():

```
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
BEFORE INSERT
POSITION 0
AS BEGIN
NEW.EMPNO = GEN_ID (EMPNO_GEN, 1);
END
SET TERM ; !!
```

The first time the trigger fires, NEW.EMPNO is set to 1. Each subsequent firing increments NEW.EMPNO by 1.

See Also CREATE GENERATOR, SET GENERATOR

GRANT

Assigns privileges to users for specified database objects.

Availability DSQL ESQL ISQL ~~PSQL~~

```
GRANT {<privileges> ON [TABLE] {tablename | viewname}
       TO {<object> | <userlist> | GROUP UNIX_group}
   | EXECUTE ON PROCEDURE procname TO {<object> | <userlist>}
   | <role_granted> TO {PUBLIC | <role_grantee_list>}};

<privileges> = ALL [PRIVILEGES] | <privilege_list>

 <privilege_list> = {
 SELECT
 | DELETE
```

```
  | INSERT
  | UPDATE [(col [, col ...])]
  | REFERENCES [(col [, col ...])]
  }
    [, <privilege_list> ...]

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    }
    [, <object> ...]

<userlist> = {
    [USER] username
    | rolename
    | UNIX_user
    }
    [, <userlist> ...]
    [WITH GRANT OPTION]

<role_granted> = rolename [, rolename ...]

 <role_grantee_list> = [USER] username [, [USER] username ...]
 [WITH ADMIN OPTION]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
privilege_list	Name of privilege to be granted; valid options are SELECT, DELETE, INSERT, UPDATE, and REFERENCES
col	Column to which the granted privileges apply
tablename	Name of an existing table for which granted privileges apply
viewname	Name of an existing view for which granted privileges apply
GROUP *unix_group*	On a UNIX system, the name of a group defined in **/etc/group**
object	Name of an existing procedure, trigger, or view; PUBLIC is also a permitted value
userlist	A user in **isc4.gdb** or a rolename created with CREATE ROLE
WITH GRANT OPTION	Passes GRANT authority for privileges listed in the GRANT statement to *userlist*
rolename	An existing role created with the CREATE ROLE statement
role_grantee_list	A list of users to whom *rolename* is granted; users must be in **isc4.gdb**
WITH ADMIN OPTION	Passes grant authority for roles listed to *role_grantee_list*

Description GRANT assigns privileges and roles for database objects to users, roles, or other database objects. When an object is first created, only its creator has privileges to it and only its creator can GRANT privileges for it to other users or objects.

- The following table summarizes available privileges:

Privilege	Enables users to …
ALL	Perform SELECT, DELETE, INSERT, UPDATE, and REFERENCES
SELECT	Retrieve rows from a table or view
DELETE	Remove rows from a table or view
INSERT	Store new rows in a table or view
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns
EXECUTE	Execute a stored procedure
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all

- To access a table or view, a user or object needs the appropriate SELECT, INSERT, UPDATE, DELETE, or REFERENCES privileges for that table or view. SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges can be assigned as a unit with ALL.
- A user or object must have EXECUTE privilege to call a stored procedure in an application.
- To grant privileges to a group of users, create a role using CREATE ROLE. Then use GRANT *privilege* TO *rolename* to assign the desired privileges to that role and use GRANT *rolename* TO *user* to assign that role to users. Users can be added or removed from a role on a case-by-case basis using GRANT and REVOKE. A user must specify the role at connection time to actually have those privileges.

- See *Using Firebird*— Connecting to a database (ch. 14 p. 282) and Database-level security in ch. 22 of the same volume for more information about invoking a role when connecting to a database.

- On Linux/UNIX systems, privileges can be granted to groups listed in **/etc/groups** and to any UNIX user listed in **/etc/passwd** on both the client and server, as well as to individual users and to roles.
- To allow another user to reference a columns from a foreign key, grant REFERENCES privileges on the primary key table or on the table's primary key columns to the owner of the foreign key table. You

must also grant REFERENCES or SELECT privileges on the primary key table to any user who needs to write to the foreign key table.

Tip Make it easy: if read security is not an issue, GRANT REFERENCES on the primary key table to PUBLIC.

- If you grant the REFERENCES privilege, it must, at a minimum, be granted to all columns of the primary key. When REFERENCES is granted to the entire table, columns that are not part of the primary key are not affected in any way.
- When a user defines a foreign key constraint on a table owned by someone else, Firebird checks that that user has REFERENCES privileges on the referenced table.
- The privilege is used at runtime to verify that a value entered in a foreign key field is contained in the primary key table.
- You can grant REFERENCES privileges to roles.
- To give users permission to grant privileges to other users, provide a *userlist* that includes the WITH GRANT OPTION. Users can grant to others only the privileges that they themselves possess.
- To grant privileges to all users, specify PUBLIC in place of a list of user names. Specifying PUBLIC grants privileges only to users, not to database objects.

Privileges can be removed only by the user who assigned them, using REVOKE. If ALL privileges are assigned, then ALL privileges must be revoked. If privileges are granted to PUBLIC, they can be removed only for PUBLIC.

Examples The following ISQL statement grants SELECT and DELETE privileges to a user. The WITH GRANT OPTION gives the user GRANT authority.

```
GRANT SELECT, DELETE ON COUNTRY TO CHLOE WITH GRANT OPTION;
```

The next ESQL statement, from an embedded program, grants SELECT and UPDATE privileges to a procedure for a table:

```
EXEC SQL
  GRANT SELECT, UPDATE ON JOB TO PROCEDURE GET_EMP_PROJ;
```

This ESQL statement grants EXECUTE privileges for a procedure to another procedure and to a user:

```
EXEC SQL
  GRANT EXECUTE ON PROCEDURE GET_EMP_PROJ
  TO PROCEDURE ADD_EMP_PROJ, LUIS;
```

The following example creates a role called “administrator”, grants UPDATE privileges on table1 to that role, and then grants the role to user1, user2, and user3. These users then have UPDATE and REFERENCES privileges on table1.

```
CREATE ROLE administrator;
GRANT UPDATE ON table1 TO administrator;
GRANT administrator TO user1, user2, user3;
```

See Also REVOKE

For more information about privileges, see *Using Firebird*— Database-level security (ch. 22 p. 429).

INSERT

Adds one or more new rows to a specified table.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

```
INSERT [TRANSACTION transaction] INTO <object> [(col [, col ...])]
    {VALUES (<val> [, <val> ...]) | <select_expr>};

<object> = tablename | viewname

<val> = {:variable | <constant> | <expr> | (<single_select_expr>)
| <function> | udf ([<val> [, <val> ...]])
| NULL | USER | RDB$DB_KEY | ?
  } [COLLATE collation]

<constant> = num | 'string' | _charsetname 'string'

<function> = CAST (<val> AS <datatype>)
| UPPER (<val>)
  | GEN_ID (generator, <val>)
```

Argument	Description
expr	A valid SQL expression that results in a single column value
select_expr	A SELECT that returns zero or more rows and where the number of columns in each row is the same as the number of items to be inserted
single_select_expr	A SELECT that returns a single column value, that can be generated by a summary SQL function, and that must be enclosed in parentheses

Notes on the INSERT statement

- In SQL and ISQL, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and ISQL, *val* cannot be a variable.
- When you need to qualify a constant string with a specific character set, prefix the character set name with an underscore, to indicate the name is not a regular SQL identifier. You must use, for example:

```
_WIN1252 'This is my string';
```

- You cannot specify a COLLATE clause for blob columns.

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
TRANSACTION *transaction*	Name of the transaction that controls the execution of the INSERT
INTO *object*	Name of an existing table or view into which to insert data

Argument	Description
col	Name of an existing column in a table or view into which to insert values
VALUES (*val* [, *val* …])	Lists values to insert into the table or view; values must be listed in the same order as the target columns
select_expr	Query that returns row values to insert into target columns

Description INSERT stores one or more new rows of data in an existing table or view. INSERT is one of the database privileges controlled by the GRANT and REVOKE statements.

Values are inserted into a row in column order unless an optional list of target columns is provided. If the target list of columns is a subset of available columns, default or NULL values are automatically stored in all unlisted columns.

If the optional list of target columns is omitted, the VALUES clause must provide values to insert into all columns in the table.

To insert a single row of data, the VALUES clause should include a specific list of values to insert.

To insert multiple rows of data, specify a *select_expr* that retrieves existing data from another table to insert into this one. The selected columns must correspond to the columns listed for insert.

Important It is valid to select from the same table into which insertions are made, but this practice is not advised because it may result in infinite row insertions.

The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the INSERT operation. The TRANSACTION clause is not available in DSQL or ISQL.

Examples The following statement, from an ESQL application, adds a row to a table, assigning values from host-language variables to two columns:

```
EXEC SQL
  INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
     VALUES (:emp_no, :proj_id);
```

The next ISQL statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
   SELECT * FROM NEW_PROJECTS
   WHERE NEW_PROJECTS.START_DATE > '6-JUN-1994';
```

See Also GRANT, REVOKE, SET TRANSACTION, UPDATE

INSERT CURSOR (BLOB)

Inserts data into a blob cursor
in units of a blob segment-length or less in size.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax

```
INSERT CURSOR cursor
             VALUES (:buffer [INDICATOR] :bufferlen);
```

Argument	Description
cursor	Name of the blob cursor
VALUES	Clause containing the name and length of the buffer variable to insert
:*buffer*	Name of host-variable buffer containing information to insert
INDICATOR	Indicates that the length of data placed in the buffer follows
:*bufferlen*	Length, in bytes, of the buffer to insert

Description INSERT CURSOR writes blob data into a column. Data is written in units equal to or less than the segment size for the blob. Before inserting data into a blob cursor:

- Declare a local variable, *buffer*, to contain the data to be inserted.
- Declare the length of the variable, *bufferlen*.
- Declare a blob cursor for INSERT and open it.

Each INSERT into the blob column inserts the current contents of *buffer*. Between statements fill *buffer* with new data. Repeat the INSERT until each existing *buffer* is inserted into the blob.

Important INSERT CURSOR requires the INSERT privilege, a table privilege controlled by the GRANT and REVOKE statements.

Example The following ESQL statement shows an insert into the blob cursor:

```
EXEC SQL
INSERT CURSOR BC VALUES (:line INDICATOR :len);
```

See Also CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB), OPEN (BLOB)

MAX( )

Retrieves the maximum value in a column.

Availability DSQL ESQL ISQL PSQL

Syntax `MAX ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Searches all values in a column
DISTINCT	Eliminates duplicate values before finding the largest
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF

Description MAX() is an aggregate function that returns the largest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MAX() returns a NULL value.

When MAX() is used on a CHAR, VARCHAR, or BLOB text column, the largest value returned varies depending on the character set and collation in use for the column. A default character set can be specified for an entire database with the DEFAULT CHARACTER SET clause in CREATE DATABASE, or specified at the column level with the COLLATE clause in CREATE TABLE.

Example The following ESQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), CREATE DATABASE, CREATE TABLE, MIN( ), SUM( )

MIN( )

Retrieves the minimum value in a column.

Availability DSQL ESQL ISQL PSQL

Syntax `MIN ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Searches all values in a column
DISTINCT	Eliminates duplicate values before finding the smallest
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF

Description MIN() is an aggregate function that returns the smallest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MIN() returns a NULL value.

When MIN() is used on a CHAR, VARCHAR, or blob text column, the smallest value returned varies depending on the character set and collation in use for the column. Use the DEFAULT CHARACTER SET clause in CREATE DATABASE to specify a default character set for an entire database, or the COLLATE clause in CREATE TABLE to specify a character set at the column level.

Example The following ESQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), CREATE DATABASE, CREATE TABLE, MAX( ), SUM( )

OPEN

Retrieve specified rows from a cursor declaration.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax **ESQL form:**

```
OPEN [TRANSACTION transaction] cursor;
```

DSQL form:

```
OPEN [TRANSACTION transaction] cursor [USING SQL DESCRIPTOR xsqlda]
```

Blob form: See OPEN (BLOB).

Argument	Description
TRANSACTION *transaction*	Name of the transaction that controls execution of OPEN
cursor	Name of a previously declared cursor to open
USING DESCRIPTOR *xsqlda*	Passes the values corresponding to the prepared statement's parameters through the extended descriptor area (XSQLDA)

Description OPEN evaluates the search condition specified in a cursor's DECLARE CURSOR statement. The selected rows become the *active set* for the cursor.

A cursor is a one-way pointer into the ordered set of rows retrieved by the SELECT in a DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declares the cursor; the SELECT statement determines rows retrieved for the cursor
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR; the resulting rows become the cursor's *active set*
3	FETCH	Retrieves the current row from the active set, starting with the first row • Subsequent FETCH statements advance the cursor through the set
4	CLOSE	Closes the cursor and release system resources

Examples The following ESQL statement opens a cursor:

```
EXEC SQL
OPEN C;
```

See Also CLOSE, DECLARE CURSOR, FETCH

OPEN (BLOB)

Opens a previously declared blob cursor and prepares it for read or insert.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax
```
OPEN [TRANSACTION name] cursor
     {INTO | USING} :blob_id;
```

Argument	Description
TRANSACTION *name*	Specifies the transaction under which the cursor is opened Default: The default transaction
cursor	Name of the blob cursor
INTO | USING	Depending on blob cursor type, use one of these: INTO: For INSERT BLOB USING: For READ BLOB
blob_id	Identifier for the blob column

Description OPEN prepares a previously declared cursor for reading or inserting blob data. Depending on whether the DECLARE CURSOR statement declares a READ or INSERT BLOB cursor, OPEN obtains the value for Blob ID differently:

- For a READ BLOB, the *blob_id* comes from the outer TABLE cursor.
- For an INSERT BLOB, the *blob_id* is returned by the system.

Examples The following ESQL statements declare and open a blob cursor:

```
EXEC SQL
DECLARE BC CURSOR FOR
  INSERT BLOB PROJ_DESC INTO PRJOECT;

EXEC SQL
OPEN BC INTO :blob_id;
```

See Also CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB),INSERT CURSOR (BLOB)

PREPARE

Prepares a statement for execution in embedded SQL.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax
```
PREPARE [TRANSACTION transaction] statement
        [INTO SQL DESCRIPTOR xsqlda] FROM {:variable | 'string'};
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed
statement	Establishes an alias for the prepared statement that can be used by subsequent DESCRIBE and EXCUTE statements
INTO *xsqlda*	Specifies an XSQLDA to be filled in with the description of the select-list columns in the prepared statement
:*variable* | '*string*'	DSQL statement to PREPARE; can be a host-language variable or a string literal

Description PREPARE is one of a group of statements that perform operations to prepare statements for execution in ESQL. It sets up a statement for repeated execution by:

- Checking the statement for syntax errors.
- Determining datatypes of optionally specified dynamic parameters.
- Optimizing statement execution.
- Compiling the statement for execution by EXECUTE.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution
DESCRIBE	Fills in the XSQLDA with information about the statement
EXECUTE	Executes a previously prepared statement
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it

After a statement is prepared, it is available for execution as many times as necessary during the current session. To prepare and execute a statement only once, use EXECUTE IMMEDIATE.

statement establishes a symbolic name for the actual DSQL statement to prepare. It is *not* declared as a host-language variable. Except for C programs, **gpre** does not distinguish between uppercase and lowercase in *statement*, treating "B" and "b" as the same character. For C programs, use the **gpre -either_case** switch to activate case sensitivity during preprocessing.

If the optional INTO clause is used, PREPARE also fills in the extended SQL descriptor area (XSQLDA) with information about the datatype, length, and name of select-list columns in the prepared statement. This clause is useful only when the statement to prepare is a SELECT.

Note The DESCRIBE statement can be used instead of the INTO clause to fill in the XSQLDA for a select list.

The FROM clause specifies the actual DSQL statement to PREPARE. It can be a host-language variable, or a quoted string literal. The DSQL statement to PREPARE can be any SQL data definition, data manipulation, or transaction-control statement.

Examples The following ESQL statement prepares a DSQL statement from a host-variable statement. Because it uses the optional INTO clause, the assumption is that the DSQL statement in the host variable is a SELECT.

```
EXEC SQL
PREPARE Q INTO xsqlda FROM :buf;
```

Note The previous statement could also be prepared and described in the following manner:

```
EXEC SQL
  PREPARE Q FROM :buf;
EXEC SQL
DESCRIBE Q INTO SQL DESCRIPTOR xsqlda;
```

See Also DESCRIBE, EXECUTE, EXECUTE IMMEDIATE

RECREATE PROCEDURE

RECREATE PROCEDURE redefines an existing stored procedure to a database.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax
```
RECREATE PROCEDURE name
            [(param <pdatatype> [, param <pdatatype> ...])]
            [RETURNS param <pdatatype> [, param <pdatatype> ...])]
            AS <procedure_body> [terminator]
```

Redefinition using RECREATE PROCEDURE is identical to CREATE PROCEDURE. Through this statement, it is possible to drop an existing procedure of the same name and recreate it using new specifications. All ownership and dependency restrictions applicable to DROP PROCEDURE and CREATE PROCEDURE apply.

See Also DROP PROCEDURE, CREATE PROCEDURE, ALTER PROCEDURE

RECREATE TABLE

RECREATE TABLE redefines an existing table to a database. Redefinition using RECREATE TABLE is identical to CREATE TABLE. Through this statement, it is possible to drop an existing table of the same name and recreate it using new specifications. All ownership and dependency restrictions applicable to DROP TABLE and CREATE TABLE apply.

Availability DSQL ESQL ISQL ~~PSQL~~

Important All triggers and co-created system objects are dropped by RECREATE TABLE. It is the developer's responsibility to save the source code for triggers.

Syntax RECREATE TABLE *table* [EXTERNAL [FILE] '*filespec*']
(<*col_def*> [, <*col_def*> | <*tconstraint*> ...]);

See Also DROP TABLE, CREATE TABLE, ALTER TABLE

REVOKE

Withdraws privileges from users for specified database objects.

Availability DSQL ESQL ISQL ~~PSQL~~

```
REVOKE [GRANT OPTION FOR] {<privileges> ON [TABLE]
          {tablename | viewname}
       FROM {<object> | <userlist> | <rolelist> | GROUP UNIX_group}
   | EXECUTE ON PROCEDURE procname
       FROM {<object> | <userlist>}
   | <role_granted> FROM {PUBLIC | <role_grantee_list>}};

<privileges> = ALL [PRIVILEGES] | <privilege_list>

 <privilege_list> = {
 SELECT
 | DELETE
 | INSERT
 | UPDATE [(col [, col ...])]
 | REFERENCES [(col [, col ...])]
 }
   [, <privilege_list> ...]

<object> = {
   PROCEDURE procname
   | TRIGGER trigname
   | VIEW viewname
   | PUBLIC
   }
```

[, <*object*> ...]

```
<userlist> = [USER] username [, [USER] username ...]

<rolelist> = rolename [, rolename]

<role_granted> = rolename [, rolename ...]

<role_grantee_list> = [USER] username [, [USER] username ...]
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
privilege_list	Name of privilege to be granted; valid options are SELECT, DELETE, INSERT, UPDATE, and REFERENCES
GRANT OPTION FOR	Removes grant authority for privileges listed in the REVOKE statement from *userlist*; cannot be used with *object*
col	Column for which the privilege is revoked
tablename	Name of an existing table for which privileges are revoked
viewname	Name of an existing view for which privileges are revoked
GROUP *unix_group*	On a UNIX system, the name of a group defined in **/etc/group**
object	Name of an existing database object from which privileges are to be revoked
userlist	A list of users from whom privileges are to be revoked
rolename	An existing role created with the CREATE ROLE statement
role_grantee_list	A list of users to whom *rolename* is granted; users must be in **isc4.gdb**

Description REVOKE removes privileges from users or other database objects. Privileges are operations for which a user has authority. The following table lists SQL privileges:

TABLE 2–8 SQL privileges

Privilege	Removes a user's privilege to …
ALL	Perform SELECT, DELETE, INSERT, UPDATE, REFERENCES, and EXECUTE
SELECT	Retrieve rows from a table or view
DELETE	Remove rows from a table or view
INSERT	Store new rows in a table or view
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all
EXECUTE	Execute a stored procedure

GRANT OPTION FOR revokes a user's right to GRANT privileges to other users.
The following limitations should be noted for REVOKE:

- Only the user who grants a privilege can revoke that privilege.
- A single user can be assigned the same privileges for a database object by any number of other users. A REVOKE issued by a user only removes privileges previously assigned by that particular user.
- Privileges granted to all users with PUBLIC can only be removed by revoking privileges from PUBLIC.
- When a role is revoked from a user, all privileges that granted by that user to others because of authority gained from membership in the role are also revoked.

Examples The following ISQL statement takes the SELECT privilege away from a user for a table:

```
REVOKE SELECT ON COUNTRY FROM MIREILLE;
```

The following ISQL statement withdraws EXECUTE privileges for a procedure from another procedure and a user:

```
REVOKE EXECUTE ON PROCEDURE GET_EMP_PROJ
  FROM PROCEDURE ADD_EMP_PROJ, LUIS;
```

See Also GRANT

ROLLBACK

Restores the database to its state prior to the start of the current transaction.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `ROLLBACK [TRANSACTION name] [WORK] [RELEASE];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
TRANSACTION *name*	Specifies the transaction to roll back in a multiple-transaction application [Default: roll back the default transaction]
WORK	Optional word allowed for compatibility
RELEASE	Detaches from all databases after ending the current transaction; ESQL only

Description ROLLBACK undoes changes made to a database by the current transaction, then ends the transaction. It breaks the program's connection to the database and frees system resources. Use RELEASE in the last ROLLBACK to close all open databases. Wait until a program no longer needs the database to release system resources.

The TRANSACTION clause can be used in multiple-transaction SQL applications to specify which transaction to roll back. If omitted, the default transaction is rolled back. The TRANSACTION clause is not available in DSQL or ISQL.

Note RELEASE, available only in SQL, detaches from all databases after ending the current transaction. In effect, this option ends database processing. RELEASE is supported for backward compatibility with old versions of InterBase. The preferred method of detaching is with DISCONNECT.

Examples The following ISQL statement rolls back the default transaction:

```
ROLLBACK;
```

The next ESQL statement rolls back a named transaction:

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

See Also **COMMIT**, **DISCONNECT**

For more information about controlling transactions, see *Using Firebird*— Transactions in Firebird (ch. 8 p. 90). .

SELECT

Retrieves data from one or more tables.

Availability DSQL ESQL ISQL PSQL*

* In PSQL, a variant syntax for SELECT is available. For details, refer to notes on SELECT and FOR SELECT…INTO…DO in the chapter PSQL-Firebird Procedural Language.

Syntax

```
SELECT [TRANSACTION transaction]
            [ {[FIRST int] [SKIP int]} ]
            [DISTINCT | ALL]
            {* | <val> [, <val> ...]}
            [INTO :var [, :var ...]]
            FROM <tableref> [, <tableref> ...]
            [WHERE <search_condition>]
            [GROUP BY col [COLLATE collation] [, col [COLLATE collation]
        ...]
```

```
            [HAVING <search_condition>]
            [UNION [ALL] <select_expr>]
            [PLAN <plan_expr>]
            [ORDER BY <order_list>]
            [FOR UPDATE [OF col [, col ...]]];

<val> = {
   col [<array_dim>] | :variable
   | <constant> | <expr> | <function>
   | udf ([<val> [, <val> ...]])
   | NULL | USER | RDB$DB_KEY | ?
   } [COLLATE collation] [AS alias]

<array_dim> = [[x:]y [, [x:]y ...]]

<constant> = num | 'string' | _charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
   | SUM ([ALL] <val> | DISTINCT <val>)
   | AVG ([ALL] <val> | DISTINCT <val>)
   | MAX ([ALL] <val> | DISTINCT <val>)
   | MIN ([ALL] <val> | DISTINCT <val>)
   | CAST (<val> AS <datatype>)
   | UPPER (<val>)
   | GEN_ID (generator, <val>)

<tableref> = <joined_table> | table | view | procedure
   [(<val> [, <val> ...])] [alias]

<joined_table> = <tableref> <join_type> JOIN <tableref>
   ON <search_condition> | (<joined_table>)

<join_type> = [INNER] JOIN
   | {LEFT | RIGHT | FULL } [OUTER]} JOIN
```

```
 <search_condition> = <val> <operator> {<val> | (<select_one>)}
 | <val> [NOT] BETWEEN <val> AND <val>
 | <val> [NOT] LIKE <val> [ESCAPE <val>]
 | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
 | <val> IS [NOT] NULL
 | <val> {>= | <=} <val>
 | <val> [NOT] {= | < | >} <val>
 | {ALL | SOME | ANY} (<select_list>)
 | EXISTS (<select_expr>)
 | SINGULAR (<select_expr>)
 | <val> [NOT] CONTAINING <val>
 | <val> [NOT] STARTING [WITH] <val>
 | (<search_condition>)
 | NOT <search_condition>
 | <search_condition> OR <search_condition>
   | <search_condition> AND <search_condition>

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<plan_expr> =
   [JOIN | [SORT] [MERGE]] ({<plan_item> | <plan_expr>}
   [, {<plan_item> | <plan_expr>} ...])

 <plan_item> = {table | alias}
   {NATURAL | INDEX (<index> [, <index> ...])| ORDER <index>}

<order_list> =
   {col | int} [COLLATE collation]
      [ASC[ENDING] | DESC[ENDING]]
      [, <order_list> ...]
```

Argument	Description
expr	A valid SQL expression that results in a single value
select_one	A SELECT on a single column that returns exactly one value
select_list	A SELECT on a single column that returns zero or more rows
select_expr	A SELECT on a list of values that returns zero or more rows

Notes on SELECT syntax

- It is possible to use expressions involving built-in functions and UDFs in the GROUP BY list. The expression must have a corresponding expression in the output set specifed in the SELECT clause. Example:

```
SELECT STRLEN(RTRIM(RDB$RELATION_NAME)), COUNT(*) FROM RDB$RELATIONS
GROUP BY STRLEN(RTRIM(RDB$RELATION_NAME))
ORDER BY 2
```

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array = varchar(6)[5,5]
```

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

```
my_array = integer[20:30]
```

- In ESQL and ISQL, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and ISQL, *val* cannot be a variable.
- You cannot specify a COLLATE clause for blob columns.

- When you need to qualify a constant string with a specific character set, prefix the character set name with an underscore, to indicate the name is not a regular SQL identifier. You must use, for example:

```
_WIN1252 'This is my string';
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed; ESQL only
SELECT [DISTINCT | ALL]	Specifies data to retrieve. DISTINCT prevents duplicate values from being returned. ALL, the default, retrieves every value
SELECT {[FIRST *m*] | [SKIP *n*]} ... ORDER BY ...	FIRST *m* returns an output set consisting of *m* rows, optionally SKIPping *n* rows and returning a set beginning (*n*+1) rows from the "top" of the set specified by the rest of the SELECT specification. If SKIP *n* is used and the [FIRST *m*] parameter is omitted, the output set returns all rows in the SELECT specification except the "top" *n* rows. These parameters generally make sense only if applied to a sorted set.
{*|*val* [, *val* …]}	The asterisk (*) retrieves all columns for the specified tables *val* [, *val* …] retrieves a list of specified columns, values, and expressions
INTO :*var* [, *var* …]	Singleton select in ESQL only; specifies a list of host-language variables into which to retrieve values
FROM *tableref* [, *tableref* …]	List of tables, views, and stored procedures from which to retrieve data; list can include joins and joins can be nested
table	Name of an existing table in a database

Argument	Description
view	Name of an existing view in a database
procedure	Name of an existing stored procedure that functions like a SELECT statement
alias	Brief, alternate name for a table, view, or column; after declaration in *tableref*, *alias* can stand in for subsequent references to a table or view
joined_table	A table reference consisting of a JOIN
join_type	Type of join to perform. Default: INNER
WHERE *search_condition*	Specifies a condition that limits rows retrieved to a subset of all available rows
GROUP BY *col* [, *col* …]	Partitions the results of a query, assembling the output into groups formed on the basis of common values in all of the output columns named in the grouping list. Precedence of grouping columns is left=high. Aggregations apply to the grouping column having the lowest precedence.
COLLATE *collation*	Specifies the collation order for the data retrieved by the query
HAVING *search_condition*	Used with GROUP BY; specifies a condition that limits grouped rows returned
UNION [ALL]	Combines two or more tables that are fully or partially identical in structure; the ALL option keeps identical rows separate instead of folding them together into one

Argument	Description
PLAN *plan_expr*	Specifies the access plan for the Firebird optimizer to use during retrieval
plan_item	Specifies a table and index method for a plan
ORDER BY *order_list*	Specifies columns to order, either by column name or ordinal number in the query, and the order (ASC or DESC) in which rows to return the rows

Description SELECT retrieves data from tables, views, or stored procedures. Variations of the SELECT statement make it possible to:

- Retrieve a single row, or part of a row, from a table. This operation is referred to as a *singleton select*.

In embedded applications, all SELECT statements that occur outside the context of a cursor must be singleton selects.

- Retrieve multiple rows, or parts of rows, from a table.

In embedded applications, multiple row retrieval is accomplished by embedding a SELECT within a DECLARE CURSOR statement.

In ISQL, SELECT can be used directly to retrieve multiple rows.

- Retrieve related rows, or parts of rows, from a join of two or more tables.
- Retrieve all rows, or parts of rows, from union of two or more tables.

All SELECT statements consist of two required clauses (SELECT, FROM), and possibly others (INTO, WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY). The following table explains the purpose of each clause, and when they are required:

TABLE 2–9 SELECT statement clauses

Clause	Purpose	Singleton SELECT	Multi-row SELECT
SELECT	Lists columns to retrieve	Required	Required
FIRST *m*	Specifies *m* rows to output from the "top" of an ordered set	Not allowed	Optional
SKIP *n*	Specifies the number of rows to skip before commencing output	Not allowed	Optional
INTO	Lists host variables for storing retrieved columns	Required	Not allowed
FROM	Identifies the tables to search for values	Required	Required
WHERE	Specifies the search conditions used to restrict retrieved rows to a subset of all available rows; a WHERE clause can contain its own SELECT statement, referred to as a *subquery*	Optional	Optional
GROUP BY	Groups related rows based on common column values; used in conjunction with HAVING	Optional	Optional
HAVING	Restricts rows generated by GROUP BY to a subset of those rows	Optional	Optional
UNION	Combines the results of two or more SELECT statements to produce a single, dynamic table without duplicate rows	Optional	Optional

TABLE 2–9 SELECT statement clauses (*continued*)

Clause	Purpose	Singleton SELECT	Multi-row SELECT
ORDER BY	Specifies which columns to order, either by column name or by ordinal number in the query, and the sort order of rows returned: ascending (ASC) [default] or descending (DESC)	Optional	Optional
PLAN	Specifies the query plan that should be used by the query optimizer instead of one it would normally choose	Optional	Optional
FOR UPDATE	Specifies columns listed after the SELECT clause of a DECLARE CURSOR statement that can be updated using a WHERE CURRENT OF clause	—	Optional

Because SELECT is such a ubiquitous and complex statement, a meaningful discussion lies outside the scope of this reference. To learn how to use SELECT in ISQL, see the *Operations Guide*. For a complete explanation of SELECT and its clauses, see the *Embedded SQL Guide*.

Examples The following ISQL statement selects columns from a table:

```
SELECT JOB_GRADE, JOB_CODE, JOB_COUNTRY, MAX_SALARY FROM PROJECT;
```

The next ISQL statement uses the * wildcard to select all columns and rows from a table:

```
SELECT * FROM COUNTRIES;
```

The following ESQL statement uses an aggregate function to count all rows in a table that satisfy a search condition specified in the WHERE clause:

```
EXEC SQL
    SELECT COUNT (*) INTO :cnt FROM COUNTRY
    WHERE POPULATION > 5000000;
```

The next ISQL statement establishes a table alias in the SELECT clause and uses it to identify a column in the WHERE clause:

```
SELECT C.CITY FROM CITIES C
  WHERE C.POPULATION < 1000000;
```

The following ISQL statement selects two columns and orders the rows retrieved by the second of those columns:

```
SELECT CITY, STATE FROM CITIES
  ORDER BY STATE;
```

The next ISQL statement performs a left join:

```
SELECT CITY, STATE_NAME FROM CITIES C
LEFT JOIN STATES S ON S.STATE = C.STATE
  WHERE C.CITY STARTING WITH 'San';
```

The following ISQL statement specifies a query optimization plan for ordered retrieval, utilizing an index for ordering:

```
SELECT * FROM CITIES
PLAN (CITIES ORDER CITIES_1);
  ORDER BY CITY
```

Th next ISQL statement requests the first 10 rows of an ordered set, after skipping the first 50 rows—the first row output would have been the 51st row in the unrestricted set:

```
SELECT FIRST 10 SKIP 50 * FROM CITIES
  ORDER BY CITY;
```

The next ISQL statement specifies a query optimization plan based on a three-way join with two indexed column equalities:

```
SELECT * FROM CITIES C, STATES S, MAYORS M
   WHERE C.CITY = M.CITY AND C.STATE = M.STATE
   PLAN JOIN (STATE NATURAL, CITIES INDEX DUPE_CITY,
   MAYORS INDEX MAYORS_1);
```

The next example queries two of the system tables, RDB$CHARACTER_SETS and RDB$COLLATIONS to display all the available character sets, their ID numbers, number of bytes per character, and collations. Note the use of ordinal column numbers in the ORDER BY clause.

```
SELECT RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_ID,
RDB$BYTES_PER_CHARACTER, RDB$COLLATION_NAME
FROM RDB$CHARACTER_SETS JOIN RDB$COLLATIONS
ON RDB$CHARACTER_SETS.RDB$CHARACTER_SET_ID =
RDB%COLLATIONS.RDB$CHARACTER_SET_ID
ORDER BY 1, 4;
```

See Also **DECLARE CURSOR**, **DELETE**, **INSERT**, **UPDATE**

For discussions of topics related to query specifications and SQL,see *Using Firebird*— Firebird SQL & Queries (ch. 9 p. 110).

For a full discussion of data retrieval in embedded programming using DECLARE CURSOR and SELECT, see the *Embedded SQL Guide* (EmbedSQL) of the InterBase® 6 documentation set, available from Borland.

SET DATABASE

Declares a database handle for database access.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

Syntax

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

Argument	Description
dbhandle	An alias for a specified database • Must be unique within the program • Used in subsequent SQL statements that support database handles
GLOBAL	[Default] Makes this database declaration available to all modules
STATIC	Limits scope of this database declaration to the current module
EXTERN	References a database declaration in another module, rather than actually declaring a new handle
COMPILETIME	Identifies the database used to look up column references during preprocessing • If only one database is specified in SET DATABASE, it is used both at runtime and compiletime
'*dbname*'	Location and path name of the database associated with *dbhandle*; platform-specific
RUNTIME	Specifies a database to use at runtime if different than the one specified for use during preprocessing
:*var*	Host-language variable containing a database specification, user name, or password
USER '*name*'	A valid user name on the server where the database resides • Used with PASSWORD to gain database access on the server • Required for PC client attachments, optional for all others

Argument	Description
PASSWORD '*string*'	A valid password on the server where the database resides • Used with USER to gain database access on the server • Required for PC client attachments, optional for all others.

Description SET DATABASE declares a database handle for a specified database and associates the handle with that database. It enables optional specification of different compile-time and run-time databases. Applications that access multiple databases simultaneously must use SET DATABASE statements to establish separate database handles for each database.

dbhandle is an application-defined name for the database handle. Usually handle names are abbreviations of the actual database name. Once declared, database handles can be used in subsequent CONNECT, COMMIT, and ROLLBACK statements. They can also be used within transactions to differentiate table names when two or more attached databases contain tables with the same names.

dbname is a platform-specific file specification for the database to associate with *dbhandle*. It should follow the file syntax conventions for the server where the database resides.

GLOBAL, STATIC, and EXTERN are optional parameters that determine the scope of a database declaration. The default scope, GLOBAL, means that a database handle is available to all code modules in an application. STATIC limits database handle availability to the code module where the handle is declared. EXTERN references a global database handle in another module.

The optional COMPILETIME and RUNTIME parameters enable a single database handle to refer to one database when an application is preprocessed, and to another database when an application is run by a user. If omitted, or if only a COMPILETIME database is specified, Firebird uses the same database during preprocessing and at run time.

The USER and PASSWORD parameters are required for all PC client applications, but are optional for all other remote attachments. The user name and password are verified by the server in the security database before permitting remote attachments to succeed.

Examples The following ESQL statement declares a handle for a database:

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

The next ESQL statement declares different databases at compile time and run time. It uses a host-language variable to specify the run-time database.

```
EXEC SQL
SET DATABASE EMDBP = 'employee.gdb' RUNTIME :db_name;
```

See Also COMMIT, CONNECT, ROLLBACK, SELECT

For more information on the security database, see*Using Firebird*— Managing Security (ch. 22 p. 414)..

SET GENERATOR

Sets a new value for an existing generator.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `SET GENERATOR` *`name`* `TO` *`int`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing generator
int	Value to which to set the generator, an integer from $-(2^{63})$ to $2^{63} - 1$

Description SET GENERATOR initializes a starting value for a newly created generator, or resets the value of an existing generator. A generator provides a unique, sequential numeric value through the GEN_ID() function. If a newly created generator is not initialized with SET GENERATOR, its starting value defaults to zero.

int is the new value for the generator. When the GEN_ID() function inserts or updates a value in a column, that value is *int* plus the increment specified in the GEN_ID() step parameter. Any value that can be stored in a DECIMAL(18,0) can be specified as the value in a SET GENERATOR statement.

Generators return a 64-bit value, and wrap around only after 2^{64} invocations (assuming an increment of 1). Use an ISC_INT64 variable to hold the value returned by a generator.

Tip To force a generator's first insertion value to 1, use SET GENERATOR to specify a starting value of 0, and set the step value of the GEN_ID() function to 1.

Important When resetting a generator that supplies values to a column defined with PRIMARY KEY or UNIQUE integrity constraints, be careful that the new value does not enable duplication of existing column values, or all subsequent insertions and updates will fail.

Example The following ISQL statement sets a generator value to 1,000:

```
SET GENERATOR CUST_NO_GEN TO 1000;
```

If GEN_ID() now calls this generator with a step value of 1, the first number it returns is 1,001.

See Also CREATE GENERATOR, CREATE PROCEDURE, CREATE TRIGGER, GEN_ID( )

SET NAMES

Specifies an active character set to use for subsequent database attachments.

Availability ~~DSQL~~ ESQL ISQL ~~PSQL~~

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
charset	Name of a character set that identifies the active character set for a given process; default: NONE
:*var*	Host variable containing string identifying a known character set name • Must be declared as a character set name • SQL only

Description SET NAMES specifies the character set to use for subsequent database attachments in an application. It enables the server to translate between the default character set for a database on the server and the character set used by an application on the client.

SET NAMES must appear before the SET DATABASE and CONNECT statements it is to affect.

Tip Use a host-language variable with SET NAMES in an embedded application to specify a character set interactively.

For a complete list of character sets recognized by Firebird, see chapter 4, Character Sets and Collation Orders (p. 249). Choice of character sets limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

Important If you do not specify a default character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration is performed between the source and destination character sets, so in most cases, errors occur during assignment.

Example The following statements demonstrate the use of SET NAMES in an ESQL application:

```
EXEC SQL
  SET NAMES ISO8859_1;

EXEC SQL
  SET DATABASE DB1 = 'employee.gdb';

EXEC SQL
  CONNECT;
```

The next statements demonstrate the use of SET NAMES in ISQL:

```
  SET NAMES LATIN1;
CONNECT 'employee.gdb';
```

See Also CONNECT, SET DATABASE

For more information about character sets and collation orders, see *Using Firebird*— Character Sets and Collation Orders (ch. 16 p. 301).

SET SQL DIALECT

Declares the SQL Dialect for database access.

Availability ~~DSQL~~ ESQL ISQL ~~PSQL~~

Syntax `SET SQL DIALECT` *`n`*`;`

Argument	Description
n	The SQL Dialect type, either 1, 2, or 3

Description SET SQL DIALECT declares the SQL Dialect for database access.

n is the SQL Dialect type, either 1, 2, or 3. If no dialect is specified, the default dialect is set to that of the specified compile-time database. If the default dialect is different than the one specified by the user, a warning is generated and the the default dialect is set to the user-specified value

TABLE 2–10 SQL Dialects

SQL Dialect	Used for
1	InterBase 5 and earlier compatibility
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3
3	Firebird 1: you can use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP datatypes

Examples The following ESQL statement sets the SQL Dialect to 3:

```
EXEC SQL
  SET SQL DIALECT 3;
```

See Also SHOW SQL DIALECT

SET STATISTICS

Recomputes the selectivity of a specified index.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax `SET STATISTICS INDEX name;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index for which to recompute selectivity

Description SET STATISTICS enables the selectivity of an index to be recomputed. Index selectivity is a calculation, based on the number of distinct rows in a table, that is made by the Firebird optimizer when a table is accessed. It is cached in memory, where the optimizer can access it to calculate the optimal retrieval plan for a given query. For tables where the number of duplicate values in indexed columns radically increases or decreases, periodically recomputing index selectivity can improve performance.

Only the creator of an index can use SET STATISTICS.

Note SET STATISTICS does not rebuild an index. To rebuild an index, use ALTER INDEX.

Example The following ESQL statement recomputes the selectivity for an index:

```
EXEC SQL
SET STATISTICS INDEX MINSALX;
```

See Also ALTER INDEX, CREATE INDEX, DROP INDEX

SET TRANSACTION

Starts a transaction and optionally specifies its behavior.

Availability DSQL ESQL ISQL ~~PSQL~~

Syntax

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

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in ISQL, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
NAME *transaction*	Specifies the name for this transaction • *transaction* is a previously declared and initialized host-language variable • SQL only
READ WRITE	[Default] Specifies that the transaction can read and write to tables
READ ONLY	Specifies that the transaction can only read tables
WAIT	[Default] Specifies that a transaction wait for access if it encounters a lock conflict with another transaction

Argument	Description
NO WAIT	Specifies that a transaction immediately return an error if it encounters a lock conflict
ISOLATION LEVEL	Specifies the isolation level for this transaction when attempting to access the same tables as other simultaneous transactions; default: SNAPSHOT
RESERVING *reserving_clause*	Reserves lock for tables at transaction start
USING *dbhandle* [, *dbhandle* …]	Limits database access to a subset of available databases; SQL only

Description SET TRANSACTION starts a transaction, and optionally specifies its database access, lock conflict behavior, and level of interaction with other concurrent transactions accessing the same data. It can also reserve locks for tables. As an alternative to reserving tables, multiple database SQL applications can restrict a transaction's access to a subset of connected databases.

Important Applications preprocessed with the **gpre -manual** switch must explicitly start each transaction with a SET TRANSACTION statement.

SET TRANSACTION affects the default transaction unless another transaction is specified in the optional NAME clause. Named transactions enable support for multiple, simultaneous transactions in a single application. All transaction names must be declared as host-language variables at compile time. In DSQL, this restriction prevents dynamic specification of transaction names.

By default a transaction has READ WRITE access to a database. If a transaction only needs to read data, specify the READ ONLY parameter.

When simultaneous transactions attempt to update the same data in tables, only the first update succeeds. No other transaction can update or delete that data until the controlling transaction is rolled back or committed. By default, transactions WAIT until the controlling transaction ends, then attempt their own operations. To force a transaction to return immediately and report a lock conflict error without waiting, specify the NO WAIT parameter.

ISOLATION LEVEL determines how a transaction interacts with other simultaneous transactions accessing the same tables. The default ISOLATION LEVEL is SNAPSHOT. It provides a repeatable-read view of the database at the moment the transaction starts. Changes made by other simultaneous transactions are not visible. SNAPSHOT TABLE STABILITY provides a repeatable read of the database by ensuring that transactions cannot write to tables, though they may still be able to read from them.

READ COMMITTED enables a transaction to see the most recently committed changes made by other simultaneous transactions. It can also update rows as long as no update conflict occurs. Uncommitted changes made by other transactions remain invisible until committed. READ COMMITTED also provides two optional parameters:

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

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. Reserving tables at transaction start can reduce the possibility of deadlocks.

The USING clause, available only in SQL, can be used to conserve system resources by limiting the number of databases a transaction can access.

Examples The following ESQL statement sets up the default transaction with an isolation level of READ COMMITTED. If the transaction encounters an update conflict, it waits to get control until the first (locking) transaction is committed or rolled back.

```
EXEC SQL
  SET TRANSACTION WAIT ISOLATION LEVEL READ COMMITTED;
```

The next ESQL statement starts a named transaction:

```
EXEC SQL
  SET TRANSACTION NAME T1 READ COMMITTED;
```

The following ESQL statement reserves three tables:

```
EXEC SQL
SET TRANSACTION NAME TR1
ISOLATION LEVEL READ COMMITTED
NO RECORD_VERSION WAIT
  RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
     TABLE3 FOR PROTECTED WRITE;
```

See Also COMMIT, ROLLBACK, SET NAMES

For more information about transactions, see *Using Firebird*— Transactions in Firebird (ch. 8 p. 90).

SHOW SQL DIALECT

Returns the current client SQL dialect setting
and the database SQL dialect value.

Availability ~~DSQL~~ ESQL ISQL ~~PSQL~~

Syntax `SHOW SQL DIALECT;`

Description Returns the current client SQL dialect setting and the database SQL dialect value, either 1, 2, or 3.

TABLE 2–11 SQL dialects

SQL dialect	Used for
1	InterBase 5 and earlier compatibility
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3
3	Firebird 1: you can use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP datatypes

Examples The following embedded SQL statement returns the SQL Dialect:

```
EXEC SQL
  SHOW SQL DIALECT;
```

See Also SET SQL DIALECT

SUBSTRING( )

Returns a string of specified length from within an input string, starting from a specified position in the input string.

Availability DSQL ESQL ISQL PSQL

Syntax `SUBSTRING(<input-string> <pos> [FOR <length>])`

Argument	Description
input-string	The string from which the result is to be extracted. Can be any legal CHAR or VARCHAR type or string expression.
pos	The starting position in the string: it must be an integer constant.
length	The maximum number of characters to be returned in the result: it must be an integer constant.

Description SUBSTRING is a string manipulation function that operates on a string as if it were a 1-based array of characters from the character set of the database—it considers characters, not bytes.

It will return a stream consisting of the byte at <*pos*> and all subsequent bytes up to the end of the string. If the option FOR <*length*> is specified, it will return the lesser of <*length*> bytes or the number of bytes up to the end of the input stream.

The first argument <*pos*>

- can be any expression, constant or identifier that evaluates to a string
- starts at 1 and must evaluate to an integer

Neither <*pos*> nor <*length*> can be query parameters.

Because <*pos*> and <*length*> are byte positions, the identifier can be a binary blob, or a SUB_TYPE 1 text blob with an underlying one-byte-per-character charset. The function currently does not handle text blobs with Chinese (2 byte/char maximum) or Unicode (3 byte/char maximum) character sets.

For a string argument (as opposed to a blob), the function will tackle ANY charset.

Example The following PSQL statement demonstrates a **failed attempt** to pass *pos* and *length* to SUBSTRING( ) in a stored procedure:

```
SET TERM ^;
CREATE PROCEDURE TEST_SUBSTRING(INPUTSTR VARCHAR(100),
STARTPOS INTEGER,
RESULTLENGTH INTEGER)
RETURNS (RESULT VARCHAR(100))
AS
BEGIN
  RESULT = SUBSTRING('LOW'||INPUTSTR FROM STARTPOS FOR RESULTLENGTH);
END^
SET TERM ;^
```

The procedure fails to compile because SUBSTRING( ) does not accept variable arguments into pos or length.

The following DSQL statement **will fail** with an 'unknown datatype' error because the data type of input-string cannot be determined:

```
SELECT SUBSTRING(:INPUTSTR FROM 2 FOR 4) AS RESULT
FROM RDB$DATABASE;
```

The following procedure will compile:

```
SET TERM ^;
CREATE PROCEDURE TEST_SUBSTRING(INPUTSTR VARCHAR(100))
RETURNS (RESULT VARCHAR(100))
AS
BEGIN
  RESULT = SUBSTRING('LOW'||INPUTSTR FROM 2 FOR 4);
END^
SET TERM ;^
```

The following succeeds because the input-string argument refers to a defined database column:

```
SELECT SUBSTRING(REGISTRATION FROM 2 FOR 4) AS RESULT
FROM AIRCRAFT WHERE...
...;
```

See Also The user-defined (external) functions **substr** and **substrlen**

SUM( )

Totals the numeric values in a specified column.

Availability DSQL ESQL ISQL PSQL

Syntax `SUM ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Totals all values in a column
DISTINCT	Eliminates duplicate values before calculating the total
val	A column, constant, host-language variable, expression, non-aggregate function, or UDF that evaluates to a numeric datatype

Description SUM() is an aggregate function that calculates the sum of numeric values for a column. If the number of qualifying rows is zero, SUM() returns a NULL value.

Example The following ESQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG( ), COUNT( ), MAX( ), MIN( )

UPDATE

Changes the data in all or part of an existing row in a table, view, or active set of a cursor.

Availability DSQL ESQL ISQL PSQL

Syntax **ESQL form:**

```
UPDATE [TRANSACTION transaction] {table | view}
   SET col = <val> [, col = <val> ...]
   [WHERE <search_condition> | WHERE CURRENT OF cursor];
```

DSQL and ISQL form:

```
UPDATE {table | view}
   SET col = <val> [, col = <val> ...]
   [WHERE <search_condition>

 <val> = {
 col [<array_dim>]
 | :variable
 | <constant>
 | <expr>
 | <function>
```

```
  | udf ([<val> [, <val> ...]])
  | NULL
  | USER
  | ?
  }
    [COLLATE collation]

<array_dim> = [[x:]y [, [x:]y ...]]

<constant> = num | 'string' | _charsetname 'string'

 <function> = CAST (<val> AS <datatype>)
  | UPPER (<val>)
    | GEN_ID (generator, <val>)
```

<*expr*> = A valid SQL expression that results in a single value.
<*search_condition*> = See CREATE TABLE for a full description.

Notes on the UPDATE statement

- In SQL and ISQL, you cannot use *val* as a parameter placeholder (like "?").
- In DSQL and ISQL, *val* cannot be a variable.
- When you need to qualify a constant string with a specific character set, prefix the character set name with an underscore, to indicate the name is not a regular SQL identifier. You must use, for example:

```
_WIN1252 'This is my string';
```

- You cannot specify a COLLATE clause for blob columns

.

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed (ESQL only)
table | *view*	Name of an existing table or view to update.
SET *col* = *val*	Specifies the columns to change and the values to assign to those columns
WHERE *search_condition*	Searched update only; specifies the conditions a row must meet to be modified
WHERE CURRENT OF *cursor*	Positioned update only; specifies that the current row of a cursor's active set is to be modified • Not available in DSQL and ISQL

Description UPDATE modifies one or more existing rows in a table or view. UPDATE is one of the database privileges controlled by GRANT and REVOKE.

For searched updates, the optional WHERE clause can be used to restrict updates to a subset of rows in the table. Searched updates cannot update array slices.

Important Without a WHERE clause, a searched update modifies all rows in a table.

When performing a positioned update with a cursor, the WHERE CURRENT OF clause must be specified to update one row at a time in the active set.

Note When updating a blob column, UPDATE replaces the entire blob with a new value.

Examples The following ISQL statement modifies a column for all rows in a table:

```
UPDATE CITIES
    SET POPULATION = POPULATION * 1.03;
```

The next ESQL statement uses a WHERE clause to restrict column modification to a subset of rows:

```
EXEC SQL
UPDATE PROJECT
SET PROJ_DESC = :blob_id
  WHERE PROJ_ID = :proj_id;
```

See Also DELETE, GRANT, INSERT, REVOKE, SELECT

UPPER( )

Converts a string to all uppercase.

Availability DSQL ESQL ISQL PSQL

Syntax UPPER (<*val*>)

Argument	Description
val	A column, constant, host-language variable, expression, function, or UDF that evaluates to a character datatype

Description UPPER() converts a specified string to all uppercase characters. If applied to character sets that have no case differentiation, UPPER() has no effect.

Examples The following ISQL statement changes the name, BMatthews, to BMATTHEWS:

```
UPDATE EMPLOYEE
    SET EMP_NAME = UPPER (BMatthews)
    WHERE EMP_NAME = 'BMatthews';
```

The next ISQL statement creates a domain called PROJNO with a CHECK constraint that requires the value of the column to be all uppercase:

```
CREATE DOMAIN PROJNO
AS CHAR(5)
  CHECK (VALUE = UPPER (VALUE));
```

See Also CAST( ) and the user-defined (external) function lower( )

WHENEVER

Traps SQLCODE errors and warnings.

Availability ~~DSQL~~ ESQL ~~ISQL~~ ~~PSQL~~

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

Argument	Description
NOT FOUND	Traps SQLCODE = 100, no qualifying rows found for the executed statement
SQLERROR	Traps SQLCODE < 0, failed statement
SQLWARNING	Traps SQLCODE > 0 AND < 100, system warning or informational message
GOTO *label*	Jumps to program location specified by *label* when a warning or error occurs
CONTINUE	Ignores the warning or error and attempts to continue processing

Description WHENEVER traps for SQLCODE errors and warnings. Every executable SQL statement returns a SQLCODE value to indicate its success or failure. If SQLCODE is zero, statement execution is successful. A non-zero value indicates an error, warning, or not found condition.

If the appropriate condition is trapped for, WHENEVER can:

- Use GOTO *label* to jump to an error-handling routine in an application.
- Use CONTINUE to ignore the condition.

WHENEVER can help limit the size of an application, because the application can use a single suite of routines for handling all errors and warnings.

WHENEVER statements should precede any SQL statement that can result in an error. Each condition to trap for requires a separate WHENEVER statement. If WHENEVER is omitted for a particular condition, it is not trapped.

Tip Precede error-handling routines with WHENEVER … CONTINUE statements to prevent the possibility of infinite looping in the error-handling routines.

Example In the following code from an ESQL application, three WHENEVER statements determine which label to branch to for error and warning handling:

```
EXEC SQL
WHENEVER SQLERROR GO TO Error; /* Trap all errors. */
EXEC SQL
  WHENEVER NOT FOUND GO TO AllDone; /* Trap SQLCODE = 100 */
EXEC SQL
  WHENEVER SQLWARNING CONTINUE; /* Ignore all warnings. */
```

For a complete discussion of error-handling methods and programming, see the *Embedded SQL Guide*, (EmbedSQL.pdf) of the InterBase® 6 documentation set, available from Borland.

CHAPTER 3

PSQL-Firebird Procedural Language

Nomenclature conventions

This chapter topic the following nomenclature:

- A *block* is one or more compound statements enclosed by BEGIN and END.
- A *compound statement* is either a block or a statement.
- A *statement* is a single statement in procedure and trigger language.

To illustrate in a syntax diagram:

```
<block> =
BEGIN
   <compound_statement>
   [<compound_statement> ...]
END

 <compound_statement> = <block> | statement;
```

Assignment statement

Assigns a value to an input or output parameter or local variable. Available in triggers and stored procedures.

Syntax `variable = <expression>;`

Argument	Description
variable	A local variable, input parameter, or output parameter
expression	Any valid combination of variables, SQL operators, and expressions, including user-defined functions (UDFs) and generators

Description An assignment statement sets the value of a local variable, input parameter, or output parameter. Variables must be declared before they can be used in assignment statements.

Example The first assignment statement below sets the value of *x* to 9. The second statement sets the value of *y* at twice the value of *x*. The third statement uses an arithmetic expression to assign *z* a value of 3.

```
DECLARE VARIABLE x INTEGER;
DECLARE VARIABLE y INTEGER;
DECLARE VARIABLE z INTEGER;
x = 9;
y = 2 * x;
z = 4 * x / (y - 6);
```

See Also DECLARE VARIABLE, Input parameters, Output parameters

BEGIN … END

Defines a block of statements executed as one.

Availability Procedures Triggers

Syntax

```
<block> =
BEGIN
    <compound_statement>
    [<compound_statement> ...]
END

<compound_statement> = {<block> | statement;}
```

Description Each block of statements in the procedure body starts with a BEGIN statement and ends with an END statement. As shown in the above syntax diagram, a block can itself contain other blocks, so there may be many levels of nesting.

BEGIN and END are not followed by a semicolon. In ISQL, the final END in the procedure body is followed by the terminator specified by SET TERM.

The final END statement in a trigger terminates the trigger. The final END statement in a stored procedure operates differently, depending on the type of procedure:

- In a select procedure, the final END statement returns control to the application and sets SQLCODE to 100, which indicates there are no more rows to retrieve.
- In an executable procedure, the final END statement returns control and current values of output parameters, if any, to the calling application.

Example The following ISQL fragment of the DELETE_EMPLOYEE procedure shows two examples of BEGIN … END blocks.

```
SET TERM !! ;
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
DECLARE VARIABLE ANY_SALES INTEGER;
BEGIN
ANY_SALES = 0;
. . .
IF (ANY_SALES > 0) THEN
BEGIN
EXCEPTION REASSIGN_SALES;
EXIT;
END
. . .
END !!
```

See Also EXIT, SUSPEND

Comment

Two comment types allow programmers to add documentation to procedure and trigger code that will be ignored by the compiler.

Availability Procedures Triggers

In-line comments

Syntax `/* comment_text */`

Argument	Description
comment_text	Any number of lines of comment text

Description Comments can be placed on the same line as code (in-line with a procedure or body statement), or on separate lines.

It is good programming practice to state the input and output parameters of a procedure in a comment preceding the procedure. It is also often useful to comment local variable declarations to indicate what each variable is used for.

Example The following ISQL procedure fragment illustrates some ways to use in-line comments:

```
/*
 *Procedure DELETE_EMPLOYEE : Delete an employee.
 *
 *Parameters:
 *employee number
 *Returns:
 *--
 */
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
DECLARE VARIABLE ANY_SALES INTEGER; /* Number of sales for emp. /*
BEGIN
. . .
```

One-line comments

Syntax `-- comment_text`

This style of comment can not be used in-line between the opening clause (CREATE.., ALTER.., RECREATE..) and the final END statement. It is for use outside these bounds to “comment out” a single statement in a script or t add “one-liners” such as signatures and dates.

Argument	Description
comment_text	A single line of comment text not located within a procedure or trigger declaration

Description The “double-hyphen” style of comments can be placed alone on as single line of a script but not within a procedure or trigger declaration. The pair of hyphens must be the first symbols on the line.

Example The following script fragment illustrates a use for one-line comments:

```
-- Amended Nov 11 by Joe Bloggs
-- ----------------------------------------
SET TERM ^;
RECREATE PROCEDURE MyProc....
...
END ^
SET TERM ;^
-- ----------------------------------------
```

DECLARE VARIABLE

Declares a local variable.

Availability Procedures Triggers

Syntax `DECLARE VARIABLE` *`var datatype`*`;`

Argument	Description
var	Name of the local variable, unique within the trigger or procedure
datatype	Datatype of the local variable; can be any Firebird datatype except arrays

Description Local variables are declared and used within a stored procedure. They have no effect outside the procedure.

Local variables must be declared at the beginning of a procedure body before they can be used. Each local variable requires a separate DECLARE VARIABLE statement, followed by a semicolon (;).

Example The following header declares the local variable, ANY_SALES:

```
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
DECLARE VARIABLE ANY_SALES INTEGER;
BEGIN
. . .
```

See Also Input parameters, Output parameters

EXCEPTION

Raises the specified exception.

Availability Procedures Triggers

Syntax `EXCEPTION` *`name`*`;`

Argument	Description
name	Name of the exception being raised

Description An exception is a user-defined error that has a name and an associated text message. When raised, an exception:

- Terminates the procedure or trigger in which it was raised and undoes any actions performed (directly or indirectly) by the procedure or trigger.
- Returns an error message to the calling application. In ISQL, the error message is displayed to the screen.

Exceptions can be handled with the WHEN statement. If an exception is handled, it will behave differently.

Example The following ISQL statement defines an exception named REASSIGN_SALES:

```
CREATE EXCEPTION REASSIGN_SALES
  'Reassign the sales records before deleting this employee.' !!
```

Then these statements from a procedure body raise the exception:

```
IF (ANY_SALES > 0) THEN
  EXCEPTION REASSIGN_SALES;
```

See Also WHEN … DO, CREATE EXCEPTION

For more information on handling exceptions, see *Using Firebird*— Error trapping and handling (ch. 25 p. 549) and the associated topics in that section.

EXECUTE PROCEDURE

Executes a stored procedure.

Availability Procedures Triggers

Syntax

```
EXECUTE PROCEDURE name [:param [, :param ...]]
    [RETURNING_VALUES :param [, :param ...]];
```

Argument	Description
name	Name of the procedure being executed. Must have been previously defined to the database with CREATE PROCEDURE
[*param* [, *param* …]]	List of input parameters, if the procedure requires them • Can be constants or variables • Precede variables with a colon, except NEW and OLD context variables
[RETURNING_VALUES *param* [, *param* …]]	List of output parameters, if the procedure returns values; precede each with a colon, except NEW and OLD context variables

Description A stored procedure can itself execute a stored procedure. Each time a stored procedure calls another procedure, the call is said to be *nested* because it occurs in the context of a previous and still active call to the first procedure. A stored procedure called by another stored procedure is known as a *nested procedure*.

If a procedure calls itself, it is *recursive*. Recursive procedures are useful for tasks that involve repetitive steps. Each invocation of a procedure is referred to as an *instance*, since each procedure call is a separate entity that performs as if called from an application, reserving memory and stack space as required to perform its tasks.

Note Stored procedures can be nested up to 1,000 levels deep. This limitation helps to prevent infinite loops that can occur when a recursive procedure provides no absolute terminating condition. Nested procedure calls may be restricted to fewer than 1,000 levels by memory and stack limitations of the server.

Example The following ISQL example illustrates a recursive procedure, FACTORIAL, which calculates factorials. The procedure calls itself recursively to calculate the factorial of NUM, the input parameter.

```
SET TERM !!;
CREATE PROCEDURE FACTORIAL (NUM INT)
RETURNS (N_FACTORIAL DOUBLE PRECISION)
AS
DECLARE VARIABLE NUM_LESS_ONE INT;
BEGIN
IF (NUM = 1) THEN
BEGIN /**** Base case: 1 factorial is 1 ****/
N_FACTORIAL = 1;
EXIT;
END
ELSE
BEGIN
/**** Recursion: num factorial = num * (num-1) factorial ****/
NUM_LESS_ONE = NUM - 1;
EXECUTE PROCEDURE FACTORIAL NUM_LESS_ONE
RETURNING_VALUES N_FACTORIAL;
N_FACTORIAL = N_FACTORIAL * NUM;
EXIT;
END
END!!
SET TERM ;!!
```

See Also CREATE PROCEDURE, Input parameters, Output parameters

For more information on executing procedures, see *Using Firebird*— Using executable procedures (ch. 25 p. 539).

EXIT

Jumps to the final END statement in the procedure.

Availability Procedures Triggers

Syntax `EXIT;`

Description In both select and executable procedures, EXIT jumps program control to the final END statement in the procedure.

What happens when a procedure reaches the final END statement depends on the type of procedure:

- In a select procedure, the final END statement returns control to the application and sets SQLCODE to 100, which indicates there are no more rows to retrieve.
- In an executable procedure, the final END statement returns control and values of output parameters, if any, to the calling application.
- In a trigger, terminates execution of the current trigger and passes control to the next trigger in the phase (BEFORE/AFTER), if any.

SUSPEND also returns values to the calling program. Each of these statements has specific behavior for executable and select procedures, as shown in the following table.

TABLE 3–1 SUSPEND, EXIT, and END

Procedure type	SUSPEND	EXIT	END
Select procedure	• Suspends execution of procedure until the current result set is read • Returns output values	Jumps to final END	• Returns control to application • Sets SQLCODE to 100 (end of record stream)
Executable procedure	• Jumps to final END • Not Recommended	Jumps to final END	• Returns values • Returns control to application

TABLE 3–1 SUSPEND, EXIT, and END

Procedure type	SUSPEND	EXIT	END
Trigger	• Not valid	Jumpd to final END	• Passes control to next trigger in the phase (BEFORE/AFTER), if any

Example Consider the following procedure from an ISQL script:

```
SET TERM !!;
CREATE PROCEDURE P RETURNS (r INTEGER)
AS
BEGIN
r = 0;
WHILE (r < 5) DO
BEGIN
r = r + 1;
SUSPEND;
IF (r = 3) THEN
EXIT;
END
END!!
SET TERM ;!!
```

If this procedure is used as a select procedure in ISQL, for example,

```
SELECT * FROM P;
```

then it returns values 1, 2, and 3 to the calling application, since the SUSPEND statement returns the current value of *r* to the calling application. The procedure terminates when it encounters EXIT.

If the procedure is used as an executable procedure in ISQL, for example,

```
EXECUTE PROCEDURE P;
```

it returns 1, since the SUSPEND statement will terminate the procedure and return the current value of *r* to the calling application. SUSPEND should not be used in an executable procedure, so EXIT would be used instead.

See Also BEGIN … END, SUSPEND

FOR SELECT…INTO…DO

Repeats a block or statement for each row retrieved by the SELECT statement.

Availability Procedures Triggers

Syntax
```
FOR <select_expr>
    DO <compound_statement>
```

Argument	Description
select_expr	SELECT statement that retrieves rows from the database; the INTO clause is required and must come last
compound_statement	Statement or block executed once for each row retrieved by the SELECT statement

Description FOR SELECT is a loop statement that retrieves the row specified in the *select_expr* and performs the statement or block following DO for each row retrieved.

<*select_expr*> is a normal SELECT, except the INTO clause is required and must be the last clause.

Example The following ISQL statement selects department numbers into the local variable, RDNO, which is then used as an input parameter to the DEPT_BUDGET procedure:

```
FOR SELECT DEPT_NO
FROM DEPARTMENT
WHERE HEAD_DEPT = :DNO
INTO :RDNO
DO
BEGIN
EXECUTE PROCEDURE DEPT_BUDGET :RDNO RETURNING_VALUES :SUMB;
TOT = TOT + SUMB;
END
```

See Also SELECT

IF…THEN … ELSE

Performs a block or statement in the THEN clause if the specified IF condition is TRUE, otherwise performs the block or statement in the optional ELSE clause.

Availability Procedures Triggers

Syntax

```
IF (<condition>)
    THEN <compound_statement>
    [ELSE <compound_statement>]
```

Argument	Description
condition	Boolean expression that evaluates to TRUE, FALSE, or UNKNOWN; must be enclosed in parentheses
THEN *compound_statement*	Statement or block executed if *condition* is TRUE
ELSE *compound_statement*	Optional statement or block executed if *condition* is not TRUE

Description The IF … THEN … ELSE statement selects alternative courses of action by testing a specified condition.

condition is an expression that must evaluate to TRUE to execute the statement or block following THEN. The optional ELSE clause specifies an alternative statement or block executed if *condition* is not TRUE.

Example The following lines of code illustrate the use of IF… THEN, assuming the variables LINE2, FIRST, and LAST have been previously declared:

```
. . .
IF (FIRST IS NOT NULL) THEN
LINE2 = FIRST || ' ' || LAST;
ELSE
LINE2 = LAST;
. . .
```

See Also WHILE … DO

Input parameters

Used to pass values from an application to a stored procedure.

Availability Procedures ~~Triggers~~

Syntax CREATE PROCEDURE *name*
[(*param datatype* [, *param datatype ...*])]

Description Input parameters are used to pass values from an application to a stored procedure. They are declared in a comma-delimited list in parentheses following the procedure name in the header of CREATE PROCEDURE. Once declared, they can be used in the procedure body anywhere a variable can appear.

Input parameters are passed *by value* from the calling program to a stored procedure. This means that if the procedure changes the value of an input variable, the change has effect only within the procedure. When control returns to the calling program, the input variable will still have its original value.
Input parameters can be of any Firebird datatype except arrays.

- Firebird will accept strings as input to text BLOB sub-types.

Example The following procedure header, from an ISQL script, declares two input parameters, EMP_NO and PROJ_ID:

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
. . .
```

See Also DECLARE VARIABLE

For more information on declaring input parameters in a procedure header, see see chapter 2, CREATE PROCEDURE (p. 93). For more details, see *Using Firebird*— Creating procedures (ch. 25 p. 521).

NEW context variables

Indicates a new column value in an INSERT or UPDATE operation.

Availability ~~Procedures~~ Triggers

Syntax NEW.*column*

Argument	Description
column	Name of a column in the affected row

Description Triggers support two context variables: OLD and NEW. A NEW context variable refers to the new value of a column in an INSERT or UPDATE operation.

Context variables are often used to compare the values of a column before and after it is modified. Context variables can be used anywhere a regular variable can be used.

New values for a row can only be altered *before* actions. A trigger that fires after INSERT and tries to assign a value to NEW.*column* will have no effect. However, the actual column values are not altered until after the action, so triggers that reference values from their target tables will not see a newly inserted or updated value unless they fire after UPDATE or INSERT.

Example The following ISQL script is a trigger that fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
IF (OLD.SALARY <> NEW.SALARY) THEN
 INSERT INTO SALARY_HISTORY
 (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY,
PERCENT_CHANGE)
VALUES (OLD.EMP_NO, 'NOW', USER, OLD.SALARY,
(NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END !!
SET TERM ; !!
```

See Also OLD context variables

For more information on creating triggers, see see chapter 2, CREATE TRIGGER (p. 116). For more details, see *Using Firebird*— Creating triggers (ch. 25 p. 532).

OLD context variables

Indicates a current column value in an UPDATE or DELETE operation.

Availability ~~Procedures~~ Triggers

Syntax `OLD.`*`column`*

Argument	Description
column	Name of a column in the affected row

Description Triggers support two context variables: OLD and NEW. An OLD context variable refers to the current or previous value of a column in an INSERT or UPDATE operation.

Context variables are often used to compare the values of a column before and after it is modified. Context variables can be used anywhere a regular variable can be used.

Example The following ISQL script is a trigger that fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
IF (OLD.SALARY <> NEW.SALARY) THEN
 INSERT INTO SALARY_HISTORY
 (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
 VALUES (OLD.EMP_NO, 'NOW', USER, OLD.SALARY,
(NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END !!
SET TERM ; !!
```

See Also NEW context variables

For more information about creating triggers, see CREATE TRIGGER on p. 116 of chapter 2.

Output parameters

Used to return values from a stored procedure to the calling application.

Availability Procedures ~~Triggers~~

Syntax
```
CREATE PROCEDURE name
            [(param datatype [, param datatype ...])]
            [RETURNS (param datatype [, param datatype ...])]
```

Description Output parameters are used to return values from a procedure to the calling application. They are declared in a comma-delimited list in parentheses following the RETURNS reserved word in the header of CREATE PROCEDURE. Once declared, they can be used in the procedure body anywhere a variable can appear. They can be of any Firebird datatype except BLOB. Arrays of datatypes are also unsupported.

If output parameters are declared in a procedure's header, the procedure must assign them values to return to the calling application. Values can be derived from any valid expression in the procedure.
A procedure returns output parameter values to the calling application with a SUSPEND statement. An application receives values of output parameters from a select procedure by using the INTO clause of the SELECT statement. An application receives values of output parameters from an executable procedure by using the RETURNING_VALUES clause.
In a SELECT statement that retrieves values from a procedure, the column names must match the names and datatypes of the procedure's output parameters. In an EXECUTE PROCEDURE statement, the output parameters need not match the names of the procedure's output parameters, but the datatypes must match.

Example The following ISQL script is a procedure header declares five output parameters, HEAD_DEPT, DEPARTMENT, MNGR_NAME, TITLE, and EMP_CNT:

```
CREATE PROCEDURE ORG_CHART RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT
CHAR(25), MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
```

See Also For more information on declaring output parameters in a procedure, see see CREATE PROCEDURE on p. 93 of chapter 2.

POST_EVENT

Posts an event.

Availability Procedures Triggers

Syntax `POST_EVENT 'event_name' | col | variable;`

Argument	Description
'event_name'	Name of the event being posted; must be enclosed in single quotes
col	Name of a column whose value the posting will be based on
variable	Name of a string variable in the stored procedure or trigger

Description POST_EVENT posts an event to the event manager. When an event occurs, this statement will notify the event manager, which alerts applications waiting for the named event.

Example The following statement posts an event named “new_order”:

```
POST_EVENT 'new_order';
```

The next statement posts an event based on the current value of a column:

```
POST_EVENT NEW.COMPANY;
```

The next example posts an event based on a string variable previously declared:

```
myval = 'new_order:' || NEW.COMPANY;
POST_EVENT myval;
```

See Also **EVENT INIT**, **EVENT WAIT**

For more information on events, see *Using Firebird*— Event alerters (ch. 25 p. 512) and Handling events on a client in the same chapter.

SELECT

Retrieves a single row that satisfies the requirements of the search condition; the same as standard singleton SELECT, with some differences in syntax.

Availability Procedures Triggers

```
<select_expr> = <select_clause> <from_clause>
[<where_clause>] [<group_by_clause>]
[<having_clause>]
[<union_expression>] [<plan_clause>]
```

```
[<ordering_clause>]
<into_clause>;
```

Description In a stored procedure, use the SELECT statement with an INTO clause to retrieve a single row value from the database and assign it to a host variable. The SELECT statement must return at most one row from the database, like a standard singleton SELECT. The INTO clause is required and must be the last clause in the statement.

The INTO clause comes at the end of the SELECT statement to allow the use of UNION operators when forming multi-row output sets for selectable stored procedures. UNION is not valid in singleton SELECT statements in PSQL.

Using the SELECT statement in a procedure:

```
SELECT SUM(BUDGET), AVG(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :HEAD_DEPT
  INTO :TOT_BUDGET, :AVG_BUDGET;
```

Note In ESQL, the INTO clause follows immediately after the specification of the SELECT <column-list>—and example of the small variations that sometimes exist between the subsets of SQL.

See Also FOR SELECT…INTO…DO

For a complete explanation of the standard SELECT syntax, see SELECT on page 191 in chapter 2.

SUSPEND

Suspends execution of a select procedure until
the current result set is read; returns values to the calling application.

Availability Procedures ~~Triggers~~

Syntax SUSPEND;

Description The SUSPEND statement:

- Suspends execution of a stored procedure until the application issues the next fetch.

- Returns values of output parameters, if any.

A procedure should ensure that all output parameters are assigned values before a SUSPEND. SUSPEND should not be used in an executable procedure. Use EXIT instead to indicate to the reader explicitly that the statement terminates the procedure.

The following table summarizes the behavior of SUSPEND, EXIT, and END.

TABLE 3–2 SUSPEND, EXIT, and END

Procedure type	SUSPEND	EXIT	END
Select procedure	• Suspends execution of procedure until current result set is read • Returns output values	Jumps to final END	• Returns control to application • Sets SQLCODE to 100 (end of record stream)
Executable procedure	• Jumps to final END • Not recommended	Jumps to final END	• Returns values • Returns control to application

Note If a SELECT procedure has executable statements following the last SUSPEND in the procedure, all of those statements are executed, even though no more rows are returned to the calling program. The procedure terminates with the final END statement, which sets SQLCODE to 100.

The SUSPEND statement also delimits atomic statement blocks in select procedures. If an error occurs in a select procedure—either a SQLCODE error, GDSCODE error, or exception—the statements executed since the last SUSPEND are undone. Statements before the last SUSPEND are never undone, unless the transaction comprising the procedure is rolled back.

Example The following procedure, from an ISQL script, illustrates the use of SUSPEND and EXIT:

```
SET TERM !!;
CREATE PROCEDURE P RETURNS (R INTEGER)
AS
BEGIN
R = 0;
WHILE (R < 5) DO
BEGIN
R = R + 1;
SUSPEND;
IF (R = 3) THEN
EXIT;
END
END;
SET TERM ;!!
```

If this procedure is used as a select procedure in ISQL, for example,

```
SELECT * FROM P;
```

then it will return values 1, 2, and 3 to the calling application, since the SUSPEND statement returns the current value of *r* to the calling application until *r* = 3, when the procedure performs an EXIT and terminates.
If the procedure is used as an executable procedure in ISQL, for example,

```
EXECUTE PROCEDURE P;
```

then it will return 1, since the SUSPEND statement will terminate the procedure and return the current value of *r* to the calling application. Since SUSPEND should not be used in executable procedures, EXIT would be used instead, indicating that when the statement is encountered, the procedure is exited.

See Also EXIT, BEGIN … END

WHEN … DO

Performs the statements following DO when the specified error occurs.

Availability Procedures Triggers

Syntax

```
WHEN {<error> [, <error> ...] | ANY}
     DO <compound_statement>

<error>=
{EXCEPTION exception_name | SQLCODE number | GDSCODE errcode}
```

Argument	Description
EXCEPTION *exception_name*	The name of a user-defined exception already in the database
SQLCODE *number*	A SQLCODE error code number—see SQLCODE codes and messages on page 291 in chapter 8.
GDSCODE *errcode*	A Firebird status error code—see Firebird status array error codes on page 315 in chapter 8.
ANY	Reserved word that handles any of the above types of errors
compound_statement	Statement or block executed when any of the specified errors occur.

Important If used, WHEN must be the last statement in a BEGIN…END block. It should come after SUSPEND, if present.

Description Procedures can handle three kinds of errors with a WHEN statement:

- Exceptions raised by EXCEPTION statements in the current procedure, in a nested procedure, or in a trigger fired as a result of actions by such a procedure.
- SQL errors reported in SQLCODE.
- Firebird error codes.

The WHEN ANY statement handles any of the three types.

Handling exceptions

Instead of terminating when an exception occurs, a procedure can respond to and perhaps correct the error condition by handling the exception. When an exception is raised, it:

- Terminates execution of the BEGIN … END block containing the exception and undoes any actions performed in the block.
- Backs out one level to the next BEGIN … END block and seeks an exception-handling (WHEN) statement, and continues backing out levels until one is found. If no WHEN statement is found, the procedure is terminated and all its actions are undone.
- Performs the ensuing statement or block of statements specified after WHEN, if found.
- Returns program control to the block or statement in the procedure following the WHEN statement.

Note An exception that is handled with WHEN does not return an error message.

Handling SQL errors

Procedures can also handle error numbers returned in SQLCODE. After each SQL statement executes, SQLCODE contains a status code indicating the success or failure of the statement. It can also contain a warning status, such as when there are no more rows to retrieve in a FOR SELECT loop.

Handling Firebird error codes

Procedures can also handle Firebird error codes. For example, suppose a statement in a procedure attempts to update a row already updated by another transaction, but not yet committed. In this case, the procedure might receive a Firebird error code, *lock_conflict*. Perhaps if the procedure retries its update, the other transaction may have rolled back its changes and released its locks. By using a WHEN GDSCODE statement, the procedure can handle lock conflict errors and retry its operation.

Example For example, if a procedure attempts to insert a duplicate value into a column defined as a PRIMARY KEY, Firebird returns SQLCODE -803. This error can be handled in a procedure with the following statement:

```
WHEN SQLCODE -803
DO
BEGIN
  . . .
```

For example, the following procedure, from an ISQL script, includes a WHEN statement to handle errors that may occur as the procedure runs. If an error occurs and SQLCODE is as expected, the procedure continues with the new value of B. If not, the procedure cannot handle the error, and rolls back all actions of the procedure, returning the active SQLCODE.

```
SET TERM !!;
CREATE PROCEDURE NUMBERPROC (A INTEGER) RETURNS (B INTEGER) AS
BEGIN
B = 0;
BEGIN
UPDATE R SET F1 = F1 + :A;
UPDATE R SET F2 = F2 * F2;
UPDATE R SET F1 = F1 + :A;
WHEN SQLCODE -803 DO
B = 1;
END
EXIT;
END!!
SET TERM; !!
```

See Also EXCEPTION

More information:

- SQLCODE codes and messages
- Firebird status array error codes

WHILE … DO

Performs the statement or block following DO as long as the specified condition is TRUE.

Availability Procedures Triggers

Syntax `WHILE (<condition>) DO <compound_statement>`

Argument	Description
condition	Boolean expression tested before each execution of the statement or block following DO
compound_statement	Statement or block executed as long as *condition* is TRUE

Description WHILE … DO is a looping statement that repeats a statement or block of statements as long as a condition is true. The condition is tested at the start of each loop.

Example The following procedure, from an ISQL script, uses a WHILE … DO loop to compute the sum of all integers from one up to the input parameter:

```
SET TERM !!;
CREATE PROCEDURE SUM_INT (I INTEGER) RETURNS (S INTEGER) AS
BEGIN
S = 0;
WHILE (I > 0) DO
BEGIN
S = S + I;
I = I - 1;
END
END!!
SET TERM ; !!
```

If this procedure is called from ISQL with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results is:

```
S
==========
10
```

See Also IF…THEN … ELSE, FOR SELECT…INTO…DO

CHAPTER 4

Character Sets and Collation Orders

Character sets and collations available in Firebird

Table 4–1 lists each character set that can be used in Firebird, including the minimum and maximum number of bytes used to store each character and all collation orders supported for that character set.
The first collation order for a given character set is that set's default collation, which is used if no COLLATE clause specifies an alternative collation.

TABLE 4–1 Character sets and collation orders

Character Set		Size in Bytes		
Name	**ID**	**Maximum**	**Minimum**	**Collation orders**
ASCII	2	1	1	ASCII
BIG_5	56	2	1	BIG_5
CYRL	50	1	1	CYRL DB_RUS PDOX_CYRL
DOS437	10	1	1	DOS437 DB_DEU437 DB_ESP437 DB_FIN437 DB_FRA437 DB_ITA437 DB_NLD437

cont.

TABLE 4–1 Character sets and collation orders (*continued*)

Character Set		Size in Bytes		
Name	**ID**	**Maximum**	**Minimum**	**Collation orders**
DOS437 (cont.)	10	1	1	DB_SVE437 DB_UK437 DB_US437 PDOX_ASCII PDOX_INTL PDOX_SWEDFIN
DOS850	11	1	1	DOS850 DB_DEU850 DB_ESP850 DB_FRA850 DB_FRC850 DB_ITA850 DB_NLD850 DB_PTB850 DB_SVE850 DB_UK850 DB_US850
DOS852	45	1	1	DOS852 DB_CSY DB_PLK DB_SLO PDOX_CSY PDOX_HUN

cont.

TABLE 4–1 Character sets and collation orders (*continued*)

Character Set		Size in Bytes		
Name	**ID**	**Maximum**	**Minimum**	**Collation orders**
DOS852 (cont.)	45	1	1	PDOX_PLK PDOX_SLO
DOS857	46	1	1	DOS857 DB_TRK
DOS860	13	1	1	DOS860 DB_PTG860
DOS861	47	1	1	DOS861 PDOX_ISL
DOS863	14	1	1	DOS863 DB_FRC863
DOS865	12	1	1	DOS865 DB_DAN865 DB_NOR865 PDOX_NORDAN4
EUCJ_0208	6	2	1	EUJC_0208
GB_2312	57	2	1	GB_2312
ISO8859_1	21	1	1	ISO8859_1 DA_DA DE_DE DU_NL EN_UK *cont.*

TABLE 4–1 Character sets and collation orders (*continued*)

Character Set		Size in Bytes		
Name	**ID**	**Maximum**	**Minimum**	**Collation orders**
ISO8859_1 (cont.)	21	1	1	EN_US ES_ES FI_FI FR_CA FR_FR IS_IS IT_IT NO_NO PT_PT SV_SV
ISO8859_2	22	1	1	CS_CZ
KSC_5601	44	2	1	KSC_5601 KSC_DICTIONARY
NEXT	19	1	1	NEXT NXT_DEU NXT_FRA NXT_ITA NXT_US
NONE	0	1	1	NONE
OCTETS	1	1	1	OCTETS
SJIS_0208	5	2	1	SJIS_0208

TABLE 4–1 Character sets and collation orders (*continued*)

Character Set		Size in Bytes		
Name	**ID**	**Maximum**	**Minimum**	**Collation orders**
UNICODE_FSS	3	3	1	UNICODE_FSS
WIN1250	51	1	1	WIN1250 PXW_CSY PXW_HUNDC PXW_PLK PXW_SLO
WIN1251	52	1	1	WIN1251 PXW_CYRL
WIN1252	53	1	1	WIN1252 PXW_INTL PXW_INTL850 PXW_NORDAN4 PXW_SPAN PXW_SWEDFIN
WIN1253	54	1	1	WIN1253 PXW_GREEK
WIN1254	55	1	1	WIN1254 PXW_TURK

Supported Datatypes

TABLE 1 Datatypes supported by Firebird

Name	Size	Range/Precision	Description
BLOB	Variable	• None • Blob segment size is limited to 64K	• Dynamically sizable datatype for storing large data such as graphics, text, and digitized voice • Basic structural unit is the segment • Blob subtype describes blob contents • Blob IDs can be used as input, output, and variable declarations in stored procedures. • You cannot edit, modify, create, or examine the contents of a blob.
CHAR(*n*) CHARACTER(*n*)	*n* characters	• 1 to 32,767 bytes • Character set character size determines the maximum number of characters that can fit in 32K	• Fixed length CHAR or text string type • Alternate reserved word: CHARACTER
DATE	32 bits, signed[a]	1 Jan 0001 to 31 Dec 9999	ISC_DATE

TABLE 1 Datatypes supported by Firebird (*continued*)

Name	Size	Range/Precision	Description
DECIMAL (*precision, scale*)	Variable (16, 32, or 64 bits)	• *precision* = 1 to 18; specifies the minimum number of digits of precision to store • *scale* = 1 to 18; specifies number of decimal places for storage • *scale* must be less than or equal to *precision*	• Number with a decimal point *scale* digits from the right • Example: DECIMAL(10, 3) holds numbers accurately in the following format: ppppppp.sss
DOUBLE PRECISION	64 bits[b]	2.225×10^{-308} to 1.797×10^{308}	IEEE double precision: 15 digits
FLOAT	32 bits	1.175×10^{-38} to 3.402×10^{38}	IEEE single precision: 7 digits
INT64	64 bits	from -10^{63} to $10^{63}-1$	Signed 64-bit integer, same as NUMERIC(18,0). Implemented from Firebird 1.5 onward.
INTEGER	32 bits	–2,147,483,648 to 2,147,483,647	Signed long (longword)
NCHAR(*n*) NATIONAL CHAR(*n*) NATIONAL CHARACTER(*n*)	*n* characters	1 to 32,767 bytes	Same as CHAR(*n*), except that NCHAR uses the ISO8859_1 character set by definition.

TABLE 1 Datatypes supported by Firebird (*continued*)

Name	Size	Range/Precision	Description
NUMERIC (*precision*, *scale*)	Variable (16, 32, or 64 bits)	• *precision* = 1 to 18; specifies the exact number of digits of precision to store • *scale* = 1 to 18; specifies number of decimal places for storage • *scale* must be less than or equal to *precision*	• Number with a decimal point *scale* digits from the right • Example: NUMERIC(10,3) holds numbers accurately in the following format: ppppppp.sss
SMALLINT	16 bits	–32,768 to 32,767	Signed short (word)
TIME	32 bits, unsigned	0:00:00.0000 to 23:59:59.9999	Unsigned integer of Firebird typedef ISC_TIME: time of day, in units of 0.0001 seconds since midnight
TIMESTAMP	64 bits	1 Jan 0001 at 0:00:00.0000 to 31 Dec 9999 at 23:59:59.9999	Typedef ISC_TIMESTAMP; combines DATE and TIME information
VARCHAR (*n*)	*n* characters	• 1 to 32,765 bytes • Character set character size determines the maximum number of characters that can fit in 32K	• Variable length CHAR or text string type • Alternate reserved words: CHAR VARYING, CHARACTER VARYING

a. InterBase 5 included a DATE datatype that was 64 bits long and included the date and time. Firebird recognizes that type if you have specified dialect 1. In dialect 3, that type is called TIMESTAMP.

b. Actual size of DOUBLE PRECISION is platform dependent. Most platforms support the 64-bit size.

CHAPTER 6

User-defined Functions

FBUDF library

The following functions pass arguments by descriptor. Initially this library is available as a beta for Windows platforms, Linux to follow. Bug reports, comments and offers to convert or enhance are welcome in the Firebird developers' forum, firebird-devel@lists.sourceforge.net.

In declaring the functions in SQL, notice the differences between the multiple declarations that map to the same function. See, for example, that INULLIF() is mapped to by both INULLIF() and I64NULLIF().

The source and the DDL for declarations are in the Firebird CVS tree. To find them, select 'Browse the CVS tree' from http://sourceforge.net/projects/firebird, click on 'Browse CVS Repository' and then select Developers | Latest sources | interbase | extlib | fbudf.*.

iNVL( ) and sNVL( )

NVL() functions for both exact precision ('invl') and string ('snvl') parameters. These functions attempt to mimic the NVL function of Oracle, to output an actual value when the column has a NULL value. They take two arguments, the first being the expression being tested for NULL, the second the value to output if the first argument is NULL. NVL will simply *return* the first argument if it is not null; otherwise it will return the second argument. If both are null, you get null.

The pair of parameters should be compatible, either two numeric values (smallint, int, int64) or two string values (CHAR, VARCHAR, CSTRING). The engine does not honor the parameter types when using the technique exercised by FBUDF, so mixing a numeric and a string as arguments will yield wrong results.

```
DECLARE EXTERNAL FUNCTION INVL
INT BY DESCRIPTOR, INT BY DESCRIPTOR
RETURNS INT BY DESCRIPTOR
ENTRY_POINT 'INVL' MODULE_NAME 'FBUDF';

DECLARE EXTERNAL FUNCTION SNVL
VARCHAR(100) BY DESCRIPTOR, VARCHAR(100) BY DESCRIPTOR,
VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
ENTRY_POINT 'SNVL' MODULE_NAME 'FBUDF';
```

sNullif( ), iNullif( ) and i64Nullif( )

NULLIF() for string ('snullif'), integer ('inullif') and INT64 ('i64nullif') parameters. NULLIF should take two arguments, returning NULL if they are equivalent, or the result of the first expression if they are not equivalent.

Because of a shortcoming in the engine which prevents NULL being returned from a UDF, each of these three functions returns a zero-equivalent. This non-standard behaviour makes it not useful for "casting" certain values as NULL in order to have aggregate functions ignore nulls.

Note that the function call to both the integer and int64 functions is the same ('iNullif').

```
DECLARE EXTERNAL FUNCTION INULLIF
INT BY DESCRIPTOR, INT BY DESCRIPTOR
RETURNS INT BY DESCRIPTOR
ENTRY_POINT 'INULLIF' MODULE_NAME 'FBUDF';

DECLARE EXTERNAL FUNCTION I64NULLIF
NUMERIC(18,4) BY DESCRIPTOR, NUMERIC(18,4) BY DESCRIPTOR
RETURNS NUMERIC(18,4) BY DESCRIPTOR
ENTRY_POINT 'INULLIF' MODULE_NAME 'FBUDF';
```

```
DECLARE EXTERNAL FUNCTION SNULLIF
VARCHAR(100) BY DESCRIPTOR, VARCHAR(100) BY DESCRIPTOR,
VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
ENTRY_POINT 'SNULLIF' MODULE_NAME 'FBUDF';
```

DOW( )

Returns the full name of the day of the week from a timestamp input. The return string can be localized.

```
DECLARE EXTERNAL FUNCTION DOW
TIMESTAMP,
VARCHAR(15) RETURNS PARAMETER 2
ENTRY_POINT 'DOW' MODULE_NAME 'FBUDF';
```

SDOW( )

Returns a short string abbreviating the day of the week from a timestamp input. The return string can be localized.

```
DECLARE EXTERNAL FUNCTION SDOW
TIMESTAMP,
VARCHAR(5) RETURNS PARAMETER 2
ENTRY_POINT 'SDOW' MODULE_NAME 'FBUDF';
```

Timestamp arithmetic functions

Several functions to add segments of time to a timestamp - 'addDay', 'AddWeek', etc.

addDay( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* days earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDDAY
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDDAY' MODULE_NAME 'FBUDF';
```

addWeek( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* weeks earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDWEEK
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDWEEK' MODULE_NAME 'FBUDF';
```

addMonth( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* months earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDMONTH
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDMONTH' MODULE_NAME 'FBUDF';
```

addYear( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* years earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDYEAR
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDYEAR' MODULE_NAME 'FBUDF';
```

addMillisecond( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* milliseconds earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDMILLISECOND
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDMILLISECOND' MODULE_NAME 'FBUDF';
```

addSecond( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* seconds earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDSECOND
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDSECOND' MODULE_NAME 'FBUDF';
```

addMinute( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* minutes earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDMINUTE
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDMINUTE' MODULE_NAME 'FBUDF';
```

addHour( )

Takes a Timestamp and an integer n, returns a Timestamp which is *n* hours earlier or later depending on the sign of *n*.

```
DECLARE EXTERNAL FUNCTION ADDHOUR
TIMESTAMP, INT
RETURNS TIMESTAMP
ENTRY_POINT 'ADDHOUR' MODULE_NAME 'FBUDF';
```

right( )

Returns the rightmost *n* characters from an input string.

```
DECLARE EXTERNAL FUNCTION SRIGHT
VARCHAR(100) BY DESCRIPTOR, SMALLINT,
VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
ENTRY_POINT 'RIGHT' MODULE_NAME 'FBUDF';
```

GetExactTimestamp( )

Returns the system timestamp with milliseconds precision, a workaround (for Windows 32 only) for the prevailing deficiency of the CURRENT_TIMESTAMP context variable, which always returns '.0000' for the part-seconds portion.

```
DECLARE EXTERNAL FUNCTION GETEXACTTIMESTAMP
TIMESTAMP RETURNS PARAMETER 1
ENTRY_POINT 'GETEXACTTIMESTAMP' MODULE_NAME 'FBUDF';
```

Truncate( )

Two declarations mapping to a single function call, for 32-bit and 64-bit integers respectively, taking scaled (exact-precision) numerics of any range (up to 9 in Dialect 1 or up to 19 in Dialect 3) and returning the whole-number portion. **They do not work with float or double types**.

```
DECLARE EXTERNAL FUNCTION TRUNCATE
INT BY DESCRIPTOR
RETURNS INT BY DESCRIPTOR
ENTRY_POINT 'TRUNCATE' MODULE_NAME 'FBUDF';

DECLARE EXTERNAL FUNCTION I64TRUNCATE
NUMERIC(18) BY DESCRIPTOR
RETURNS NUMERIC(18) BY DESCRIPTOR
ENTRY_POINT 'TRUNCATE' MODULE_NAME 'FBUDF';
```

Example `truncate(14.76)` returns 14

`truncate(14.22)` returns 14

Round( )

Two declarations mapping to a single function call, for 32-bit and 64-bit integers respectively, accepting scaled (exact-precision) numerics of any range (up to 9 in Dialect 1 or up to 19 in Dialect 3) and returning the nearest whole number. You cannot specify the number of decimal places.

```
DECLARE EXTERNAL FUNCTION ROUND
INT BY DESCRIPTOR
RETURNS INT BY DESCRIPTOR
ENTRY_POINT 'ROUND' MODULE_NAME 'FBUDF';

DECLARE EXTERNAL FUNCTION I64ROUND
NUMERIC(18, 4) BY DESCRIPTOR
RETURNS NUMERIC(18,4) BY DESCRIPTOR
ENTRY_POINT 'ROUND' MODULE_NAME 'FBUDF';
```

Example `round(14.76)` returns 15

`round(14.22)` returns 14

Important These functions are compiled to use math rounding, i.e., always towards higher values starting at n.5. This differs from functions found in Borland tools that round to the ABSOLUTE higher value (they are symmetric).

For example, symmetric rounding gives

```
round(1.5) => 2

round(-1.5) => -1
```

String2blob( )

Converts a CHAR or VARCHAR type to a blob. It is like the function that exists in FreeUDFLib, but is much simpler internally.

```
DECLARE EXTERNAL FUNCTION STRING2BLOB
VARCHAR(300) BY DESCRIPTOR,
BLOB RETURNS PARAMETER 2
ENTRY_POINT 'STRING2BLOB' MODULE_NAME 'FBUDF';
```

ib_udf library

Important Several of these UDFs must be called using the FREE_IT reserved word if—and only if—they are written in thread-safe form, using *malloc* to allocate dynamic memory.

Note When trigonometric functions are passed inputs that are out of bounds, they return zero rather than NaN.

abs

Returns the absolute value of a number.

```
DECLARE EXTERNAL FUNCTION ABS
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_abs' MODULE_NAME 'ib_udf';
```

acos

Returns the arccosine of a number between –1 and 1; if the number is out of bounds it returns zero.

```
DECLARE EXTERNAL FUNCTION ACOS
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_acos' MODULE_NAME 'ib_udf';
```

ascii_char

Returns the ASCII character corresponding to the value passed in.

```
DECLARE EXTERNAL FUNCTION ASCII_CHAR
INTEGER
RETURNS CSTRING(1) FREE_IT
ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

Important Ensure that you use the version of ib_udf that is distributed with Firebird 1 or later. The InterBase® versions incorrectly attempt to return a CHAR(1), resulting in a server crash.

ascii_val

Returns the ASCII value of the character passed in.

```
DECLARE EXTERNAL FUNCTION ASCII_VAL
CHAR(1)
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_ascii_val' MODULE_NAME 'ib_udf';
```

asin

Returns the arcsin of a number between –1 and 1; returns zero if the number is out of range.

```
DECLARE EXTERNAL FUNCTION ASIN
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_asin' MODULE_NAME 'ib_udf';
```

atan

Returns the arctangent of the input value.

```
DECLARE EXTERNAL FUNCTION ATAN
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_atan' MODULE_NAME 'ib_udf';
```

atan2

Returns the arctangent of the first parameter divided by the second parameter.

```
DECLARE EXTERNAL FUNCTION ATAN2
DOUBLE PRECISION, DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_atan2' MODULE_NAME 'ib_udf';
```

bin_and

Returns the result of a binary AND operation performed on the two input values.

```
DECLARE EXTERNAL FUNCTION BIN_AND
INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_bin_and' MODULE_NAME 'ib_udf';
```

bin_or

Returns the result of a binary OR operation performed on the two input values.

```
DECLARE EXTERNAL FUNCTION BIN_OR
INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_bin_or' MODULE_NAME 'ib_udf';
```

bin_xor

Returns the result of a binary XOR operation performed on the two input values.

```
DECLARE EXTERNAL FUNCTION BIN_XOR
INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_bin_xor' MODULE_NAME 'ib_udf';
```

ceiling

Returns a double value representing the smallest integer that is greater than or equal to the input value.

```
DECLARE EXTERNAL FUNCTION CEILING
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_ceiling' MODULE_NAME 'ib_udf';
```

cos

Returns the cosine of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION COS
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_cos' MODULE_NAME 'ib_udf';
```

cosh

Returns the hyperbolic cosine of x. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION COSH
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_cosh' MODULE_NAME 'ib_udf';
```

cot

Returns 1 over the tangent of the input value.

```
DECLARE EXTERNAL FUNCTION COT
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_cot' MODULE_NAME 'ib_udf';
```

div

Divides the two inputs and returns the quotient.

```
DECLARE EXTERNAL FUNCTION DIV
INTEGER, INTEGER
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_div' MODULE_NAME 'ib_udf';
```

floor

Returns a floating-point value representing the largest integer that is less than or equal to *x*.

```
DECLARE EXTERNAL FUNCTION FLOOR
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_floor' MODULE_NAME 'ib_udf';
```

ln

Returns the natural log of a number.

```
DECLARE EXTERNAL FUNCTION LN
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_ln' MODULE_NAME 'ib_udf';
```

log

LOG(*x*,*y*) returns the logarithm base *x* of *y*.

```
DECLARE EXTERNAL FUNCTION LOG
DOUBLE PRECISION, DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_log' MODULE_NAME 'ib_udf';
```

log10

Returns the logarithm base 10 of the input value.

```
DECLARE EXTERNAL FUNCTION LOG10
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_log10' MODULE_NAME 'ib_udf';
```

lower

Returns the input string as lowercase characters. *This function works only with ASCII characters.*

Note This function can receive and return up to 32,767 characters, the limit on a Firebird character string.

```
DECLARE EXTERNAL FUNCTION lower
CSTRING(80)
RETURNS CSTRING(80) FREE_IT
ENTRY_POINT 'IB_UDF_lower' MODULE_NAME 'ib_udf';
```

ltrim

Removes leading spaces from the input string.

Note This function can receive and return up to 32,767 characters, the limit on a Firebird character string.

```
DECLARE EXTERNAL FUNCTION LTRIM
CSTRING(80)
RETURNS CSTRING(80) FREE_IT
ENTRY_POINT 'IB_UDF_ltrim' MODULE_NAME 'ib_udf';
```

mod

Divides the two input parameters and returns the remainder.

```
DECLARE EXTERNAL FUNCTION MOD
INTEGER, INTEGER
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_mod' MODULE_NAME 'ib_udf';
```

pi

Returns the value of pi = 3.14159...

```
DECLARE EXTERNAL FUNCTION PI
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_pi' MODULE_NAME 'ib_udf';
```

rand

Returns a random number between 0 and 1. The current time is used to seed the random number generator.

```
DECLARE EXTERNAL FUNCTION rand
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_rand' MODULE_NAME 'ib_udf';
```

rtrim

Removes trailing spaces from the input string.

Note This function can receive and return up to 32,767 characters, the limit on a Firebird character string.

```
DECLARE EXTERNAL FUNCTION RTRIM
CSTRING(80)
RETURNS CSTRING(80) FREE_IT
ENTRY_POINT 'IB_UDF_rtrim' MODULE_NAME 'ib_udf';
```

sign

Returns 1, 0, or –1 depending on whether the input value is positive, zero or negative, respectively.

```
DECLARE EXTERNAL FUNCTION SIGN
DOUBLE PRECISION
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_sign' MODULE_NAME 'ib_udf';
```

sin

Returns the sine of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION SIN
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_sin' MODULE_NAME 'ib_udf';
```

sinh

Returns the hyperbolic sine of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION SINH
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_sinh' MODULE_NAME 'ib_udf';
```

sqrt

Returns the square root of a number.

```
DECLARE EXTERNAL FUNCTION SQRT
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf';
```

strlen

Returns the length of a the input string.

```
DECLARE EXTERNAL FUNCTION STRLEN
CSTRING(32767)
RETURNS INTEGER BY VALUE
ENTRY_POINT 'IB_UDF_strlen' MODULE_NAME 'ib_udf';
```

substr

SUBSTR(<*string expr*>, <*pos1*>, <*pos2*>) returns the substring of <*string expr*> starting at position <*pos1*> and ending at position <*pos2*>. If <*pos2*> is past the end of the string, the function will return all characters from <*pos1*> to the end of the string. This function can receive and return up to 32,767 characters in <*string expr*>, the limit on a Firebird character string.

Note that this behavior (from Firebird 1 on) is different from that of SUBSTR in Borland and previous Firebird versions of **ib_udf**, which returns an empty string when <*pos2*> is past the end of the string.

```
DECLARE EXTERNAL FUNCTION SUBSTR
CSTRING(80), SMALLINT, SMALLINT
RETURNS CSTRING(80) FREE_IT
ENTRY_POINT 'IB_UDF_substr' MODULE_NAME 'ib_udf';
```

Example

```
UPDATE ATABLE
   SET COLUMNB = SUBSTR(COLUMNB, 4, 32765)
   WHERE...
```

substrlen

SUBSTRLEN(<*string expr*>, <*pos*>, <*length*>) returns a string of size <*length*> starting at <*pos*>. The length of the string will be the lesser of <*length*> or the number of characters from <*pos*> to the end of the input string. New in Firebird 1.

Example

```
UPDATE ATABLE
   SET COLUMNB = SUBSTRLEN(COLUMNB, 4, 99)
   WHERE...
```

tan

Returns the tangent of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION TAN
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_tan' MODULE_NAME 'ib_udf';
```

tanh

Returns the tangent of *x*. If *x* is greater than or equal to 263, or less than or equal to –263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a zero.

```
DECLARE EXTERNAL FUNCTION TANH
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_tanh' MODULE_NAME 'ib_udf';
```

CHAPTER 7

Reserved Words

This topic lists words reserved from use in Firebird SQL programs. The list includes ESQL, DSQL, ISQL, and PSQL reserved words.
Reserved words are defined for special purposes. A reserved word cannot occur in a user-declared identifier, or as the name of a table, column, index, trigger, or constraint, unless it is enclosed in double quotes.
Reserved words can be:

- Part of statements
- Used as statements
- Names of standard data structures or datatypes

Reserved in Firebird. The terms listed in the first two columns of Table 7–1 are reserved in at least one "flavor" of InterBase SQL. Some terms are exclusive to InterBase, and are not reserved words in SQL 92.
Reserved in SQL 92. The third column lists terms that are reserved words according to the SQL 92 standard. As of this writing, these terms are not reserved in Firebird; however, any or all of them could become Firebird reserved words in the future. Therefore you should avoid using these terms in any context where a Firebird reserved word cannot be used.

TABLE 7–1 Reserved words in Firebird and SQL 92

Firebird and SQL 92	Firebird only	SQL 92 only
		ABSOLUTE
ACTION		
	ACTIVE	
ADD		
	ADMIN	
	AFTER	
ALL		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
		ALLOCATE
ALTER		
AND		
ANY		
ARE		
AS		
ASC		
	ASCENDING	
		ASSERTION
AT		
		AUTHORIZATION
	AUTO	
	AUTODDL	
AVG		
	BASED	
	BASENAME	
	BASE_NAME	
	BEFORE	
BEGIN		
BETWEEN		
		BIT
		BIT_LENGTH
	BLOB	
	BLOBEDIT	
		BOTH
	BREAK	

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
	BUFFER	
BY		
	CACHE	
CASCADE		
		CASCADED
		CASE
CAST		
		CATALOG
CHAR		
CHARACTER		
CHAR_LENGTH		
CHARACTER_LENGTH		
CHECK		
	CHECK_POINT_LEN	
	CHECK_POINT_LENGTH	
CLOSE		
		COALESCE
COLLATE		
COLLATION		
COLUMN		
COMMIT		
	COMMITTED	
	COMPILETIME	
	COMPUTED	
	CONDITIONAL	
CONNECT		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
		CONNECTION
CONSTRAINT		
		CONSTRAINTS
	CONTAINING	
CONTINUE		
		CONVERT
		CORRESPONDING
COUNT		
CREATE		
		CROSS
	CSTRING	
CURRENT		
CURRENT_DATE		
	CURRENT_ROLE	
CURRENT_TIME		
CURRENT_TIMESTAMP		
CURRENT_USER		
	DATABASE	
DATE		
DAY		
	DB_KEY	
		DEALLOCATE
	DEBUG	
DEC		
DECIMAL		
DECLARE		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
DEFAULT		
		DEFERRABLE
		DEFERRED
DELETE		
DESC		
	DESCENDING	
DESCRIBE		
DESCRIPTOR		
		DIAGNOSTICS
DISCONNECT		
	DISPLAY	
DISTINCT		
	DO	
DOMAIN		
DOUBLE		
DROP		
	ECHO	
	EDIT	
ELSE		
END		
		END-EXEC
	ENTRY_POINT	
ESCAPE		
	EVENT	
		EXCEPT
EXCEPTION		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
		EXEC
EXECUTE		
EXISTS		
	EXIT	
	EXTERN	
EXTERNAL		
EXTRACT		
		FALSE
FETCH		
	FILE	
	FILTER	
	FIRST	
FLOAT		
FOR		
FOREIGN		
FOUND		
	FREE_IT	
FROM		
FULL		
	FUNCTION	
	GDSCODE	
	GENERATOR	
	GEN_ID	
		GET
GLOBAL		
		GO

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
GOTO		
GRANT		
GROUP		
	GROUP_COMMIT_WAIT	
	GROUP_COMMIT_WAIT_TIME	
HAVING		
	HELP	
HOUR		
		IDENTITY
	IF	
IMMEDIATE		
IN		
	INACTIVE	
	INDEX	
INDICATOR		
	INIT	
		INITIALLY
INNER		
INPUT		
	INPUT_TYPE	
		INSENSITIVE
INSERT		
INT		
INTEGER		
		INTERSECT
		INTERVAL

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
INTO		
IS		
ISOLATION		
	ISQL	
JOIN		
KEY		
		LANGUAGE
		LAST
	LC_MESSAGES	
	LC_TYPE	
		LEADING
LEFT		
	LENGTH	
	LEV	
LEVEL		
LIKE		
		LOCAL
	LOGFILE	
	LOG_BUFFER_SIZE	
	LOG_BUF_SIZE	
	LONG	
		LOWER
	MANUAL	
		MATCH
MAX		
	MAXIMUM	

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
	MAXIMUM_SEGMENT	
	MAX_SEGMENT	
	MERGE	
	MESSAGE	
MIN		
	MINIMUM	
MINUTE		
		MODULE
	MODULE_NAME	
MONTH		
NAMES		
NATIONAL		
NATURAL		
NCHAR		
		NEXT
NO		
	NOAUTO	
NOT		
NULL		
		NULLIF
	NUM_LOG_BUFS	
	NUM_LOG_BUFFERS	
NUMERIC		
OCTET_LENGTH		
OF		
ON		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
ONLY		
OPEN		
OPTION		
OR		
ORDER		
OUTER		
OUTPUT		
	OUTPUT_TYPE	
	OVERFLOW	
		OVERLAPS
		PAD
	PAGE	
	PAGELENGTH	
	PAGES	
	PAGE_SIZE	
	PARAMETER	
		PARTIAL
	PASSWORD	
	PLAN	
POSITION		
	POST_EVENT	
PRECISION		
PREPARE		
		PRESERVE
PRIMARY		
		PRIOR

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
PRIVILEGES		
PROCEDURE		
PUBLIC		
	QUIT	
	RAW_PARTITIONS	
	RDB$DB_KEY	
READ		
REAL		
	RECORD_VERSION	
	RECREATE	
REFERENCES		
		RELATIVE
	RELEASE	
	RESERV	
	RESERVING	
RESTRICT		
	RETAIN	
	RETURN	
	RETURNING_VALUES	
	RETURNS	
REVOKE		
RIGHT		
	ROLE	
ROLLBACK		
		ROWS
	RUNTIME	

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
SCHEMA		
		SCROLL
	SECOND	
		SECTION
SELECT		
		SESSION
		SESSION_USER
SET		
	SHADOW	
	SHARED	
	SHELL	
	SHOW	
	SINGULAR	
SIZE		
	SKIP	
SMALLINT		
	SNAPSHOT	
SOME		
	SORT	
		SPACE
SQL		
SQLCODE		
SQLERROR		
		SQLSTATE
SQLWARNING		
	STABILITY	

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
	STARTING	
	STARTS	
	STATEMENT	
	STATIC	
	STATISTICS	
	SUB_TYPE	
	SUBSTRING	
SUM		
	SUSPEND	
		SYSTEM_USER
TABLE		
		TEMPORARY
	TERMINATOR	
THEN		
TIME		
TIMESTAMP		
		TIMEZONE_HOUR
		TIMEZONE_MINUTE
TO		
		TRAILING
TRANSACTION		
TRANSLATE		
TRANSLATION		
	TRIGGER	
TRIM		
		TRUE

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
	TYPE	
	UNCOMMITTED	
UNION		
UNIQUE		
		UNKNOWN
UPDATE		
UPPER		
		USAGE
USER		
USING		
VALUE		
VALUES		
VARCHAR		
	VARIABLE	
VARYING		
	VERSION	
VIEW		
	WAIT	
	WEEKDAY	
WHEN		
WHENEVER		
WHERE		
	WHILE	
WITH		
WORK		
WRITE		

TABLE 7–1 Reserved words in Firebird and SQL 92 (*continued*)

Firebird and SQL 92	Firebird only	SQL 92 only
YEAR		
	YEARDAY	
		ZONE

New keywords added to InterBase®

The following new keywords were added to InterBase® 6.5 and should be treated as reserved for the sake of compatibility:

PERCENT	ROWS	TIES

Keywords you should reserve for Firebird

The following words should be regarded as keywords which will be reserved in forthcoming releases of Firebird:

ABS	BOTH	CASE	CHAR_LENGTH
CHARACTER_LENGTH	COALESCE	IIF	LEADING
NULLIF	OCTET_LENGTH	TRIM	TRAILING

CHAPTER 8

Error Codes and Messages

SQLCODE codes and messages

The following table lists SQLCODEs and associated messages for SQL and DSQL. Some SQLCODE values have more than one text message associated with them. In these cases, Firebird returns the most relevant string message for the error that occurred.

When code messages include the name of a database object or object type, the name is represented by a code in the SQLCODE Text column:

- <*string*>: String value, such as the name of a database object or object type.
- <*long*>: Long integer value, such as the identification number or code of a database object or object type.
- <*digit*>: Integer value, such as the identification number or code of a database object or object type.
- The Firebird number in the right-hand column is the actual error number returned in the error status vector. You can use Firebird error-handling functions to report messages based on these numbers instead of SQL code, but doing so results in non-portable SQL programs.

TABLE 8–1 SQLCODE codes and messages

SQLCODE	SQLCODE text	Firebird number
101	Segment buffer length shorter than expected	335544366L
100	No match for first value expression	335544338L
100	Invalid database key	335544354L
100	Attempted retrieval of more segments than exist	335544367L
100	Attempt to fetch past the last record in a record stream	335544374L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-84	Table/procedure has non-SQL security class defined	335544554L
-84	Column has non-SQL security class defined	335544555L
-84	Procedure <*string*> does not return any values	335544668L
-103	Datatype for constant unknown	335544571L
-104	Invalid request BLR at offset <*long*>	335544343L
-104	BLR syntax error: expected <*string*> at offset <*long*>, encountered <*long*>	335544390L
-104	Context already in use (BLR error)	335544425L
-104	Context not defined (BLR error)	335544426L
-104	Bad parameter number	335544429L
-104		335544440L
-104	Invalid slice description language at offset <*long*>	335544456L
-104	Invalid command	335544570L
-104	Internal error	335544579L
-104	Option specified more than once	335544590L
-104	Unknown transaction option	335544591L
-104	Invalid array reference	335544592L
-104	Token unknown—line <*long*>, char <*long*>	335544634L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-104	Unexpected end of command	335544608L
-104	Token unknown	335544612L
-104	Feature not supported	335544763L
-105	Specified EXTRACT part does not exist in input datatype	335544789L
-150	Attempted update of read-only table	335544360L
-150	Cannot update read-only view <*string*>	335544362L
-150	Not updatable	335544446L
-150	Cannot define constraints on views	335544546L
-151	Attempted update of read-only column	335544359L
-155	<*string*> is not a valid base table of the specified view	335544658L
-157	Must specify column name for view select expression	335544598L
-158	Number of columns does not match select list	335544599L
-162	Dbkey not available for multi-table views	335544685L
-170	Parameter mismatch for procedure <*string*>	335544512L
-170	External functions cannot have more than10 parameters	335544619L
-171	Function <*string*> could not be matched	335544439L
-171	Column not array or invalid dimensions (expected <*long*>, encountered <*long*>)	335544458L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-171	Return mode by value not allowed for this datatype	335544618L
-172	Function <*string*> is not defined	335544438L
-204	Generator <*string*> is not defined	335544463L
-204	Reference to invalid stream number	335544502L
-204	CHARACTER SET <*string*> is not defined	335544509L
-204	Procedure <*string*> is not defined	335544511L
-204	Status code <*string*> unknown	335544515L
-204	Exception <*string*> not defined	335544516L
-204	Name of Referential Constraint not defined in constraints table.	335544532L
-204	Could not find table/procedure for GRANT	335544551L
-204	Implementation of text subtype <*digit*> not located.	335544568L
-204	Datatype unknown	335544573L
-204	Table unknown	335544580L
-204	Procedure unknown	335544581L
-204	COLLATION <*string*> is not defined	335544588L
-204	COLLATION <*string*> is not valid for specified CHARACTER SET	335544589L
-204	Trigger unknown	335544595L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-204	Alias <*string*> conflicts with an alias in the same statement	335544620L
-204	Alias <*string*> conflicts with a procedure in the same statement	335544621L
-204	Alias <*string*> conflicts with a table in the same statement	335544622L
-204	There is no alias or table named <*string*> at this scope level	335544635L
-204	There is no index <*string*> for table <*string*>	335544636L
-204	Invalid use of CHARACTER SET or COLLATE	335544640L
-204	BLOB SUB_TYPE <*string*> is not defined	335544662L
-205	Column <*string*> is not defined in table <*string*>	335544396L
-205	Could not find column for GRANT	335544552L
-206	Column unknown	335544578L
-206	Column is not a blob	335544587L
-206	Subselect illegal in this context	335544596L
-208	Invalid ORDER BY clause	335544617L
-219	Table <*string*> is not defined	335544395L
-239	Cache length too small	335544691L
-260	Cache redefined	335544690L
-281	Table <*string*> is not referenced in plan	335544637L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-282	Table <*string*> is referenced more than once in plan; use aliases to distinguish	335544638L
-282	The table <*string*> is referenced twice; use aliases to differentiate	335544643L
-282	Table <*string*> is referenced twice in view; use an alias to distinguish	335544659L
-282	View <*string*> has more than one base table; use aliases to distinguish	335544660L
-283	Table <*string*> is referenced in the plan but not the from list	335544639L
-284	Index <*string*> cannot be used in the specified plan	335544642L
-291	Column used in a PRIMARY/UNIQUE constraint must be NOT NULL.	335544531L
-292	Cannot update constraints (RDB$REF_CONSTRAINTS).	335544534L
-293	Cannot update constraints (RDB$CHECK_CONSTRAINTS).	335544535L
-294	Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)	335544536L
-295	Cannot update constraints (RDB$RELATION_CONSTRAINTS).	335544545L
-296	Internal isc software consistency check (invalid RDB$CONSTRAINT_TYPE)	335544547L
-297	Operation violates CHECK constraint <*string*> on view or table	335544558L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-313	Count of column list and variable list do not match	335544669L
-314	Cannot transliterate character between character sets	335544565L
-401	Invalid comparison operator for find operation	335544647L
-402	Attempted invalid operation on a blob	335544368L
-402	Blob and array datatypes are not supported for <*string*> operation	335544414L
-402	Data operation not supported	335544427L
-406	Subscript out of bounds	335544457L
-407	Null segment of UNIQUE KEY	335544435L
-413	Conversion error from string "<*string*>"	335544334L
-413	Filter not found to convert type <*long*> to type <*long*>	335544454L
-501	Invalid request handle	335544327L
-501	Attempt to reclose a closed cursor	335544577L
-502	Declared cursor already exists	335544574L
-502	Attempt to reopen an open cursor	335544576L
-504	Cursor unknown	335544572L
-508	No current record for fetch operation	335544348L
-510	Cursor not updatable	335544575L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-518	Request unknown	335544582L
-519	The PREPARE statement identifies a prepare statement with an open cursor	335544688L
-530	Violation of FOREIGN KEY constraint: "<*string*>"	335544466L
-530	Cannot prepare a CREATE DATABASE/SCHEMA statement	335544597L
-532	Transaction marked invalid by I/O error	335544469L
-551	No permission for <*string*> access to <*string*> <*string*>	335544352L
-552	Only the owner of a table can reassign ownership	335544550L
-552	User does not have GRANT privileges for operation	335544553L
-553	Cannot modify an existing user privilege	335544529L
-595	The current position is on a crack	335544645L
-596	Illegal operation when at beginning of stream	335544644L
-597	Preceding file did not specify length, so <*string*> must include starting page number	335544632L
-598	Shadow number must be a positive integer	335544633L
-599	Gen.c: node not supported	335544607L
-600	A node name is not permitted in a secondary, shadow, cache or log file name	335544625L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-600	Sort error: corruption in data structure	335544680L
-601	Database or file exists	335544646L
-604	Array declared with too many dimensions	335544593L
-604	Illegal array dimension range	335544594L
-604	New size specified for column %s must be greater than %d characters	336068816L
-604	Cannot change datatype for column %s. Conversion from base type%s to base type %s is not permitted	336068817L
-605	Inappropriate self-reference of column	335544682L
-607	Unsuccessful metadata update	335544351L
-607	Cannot modify or erase a system trigger	335544549L
-607	Array/BLOB/DATE/TIME/TIMESTAMP datatypes not allowed in arithmetic	335544657L
-612	Cannot rename domain %s to %s. A domain with that name already exists	336068812L
-612	Cannot rename column %s to %s. A column with that name already exists	336068813L
-615	Lock on table <*string*> conflicts with existing lock	335544475L
-615	Requested record lock conflicts with existing lock	335544476L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-615	Refresh range number <*long*> already in use	335544507L
-616	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.	335544530L
-616	Cannot delete index used by an integrity constraint	335544539L
-616	Cannot modify index used by an integrity constraint	335544540L
-616	Cannot delete trigger used by a CHECK Constraint	335544541L
-616	Cannot delete column being used in an integrity constraint.	335544543L
-616	There are <*long*> dependencies	335544630L
-616	Last column in a table cannot be deleted	335544674L
-616	Column %s from table %s is referenced in %s	336068814L
-617	Cannot update trigger used by a CHECK Constraint	335544542L
-617	Cannot rename column being used in an integrity constraint.	335544544L
-618	Cannot delete index segment used by an integrity constraint	335544537L
-618	Cannot update index segment used by an integrity constraint	335544538L
-625	Validation error for column <*string*>, value “<*string*>”	335544347L
-637	Duplicate specification of <*string*> not supported	335544664L
-660	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY	335544533L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-660	Cannot create index <*string*>	335544628L
-663	Segment count of 0 defined for index <*string*>	335544624L
-663	Too many keys defined for index <*string*>	335544631L
-663	Too few key columns found for index <*string*> (incorrect column name?)	335544672L
-664	key size exceeds implementation restriction for index "<*string*>"	335544434L
-677	<*string*> extension error	335544445L
-685	Invalid blob type for operation	335544465L
-685	Attempt to index blob column in index <*string*>	335544670L
-685	Attempt to index array column in index <*string*>	335544671L
-688	Cannot change datatype for column %s. Changing datatype is not supported for BLOB or ARRAY columns	336068815L
-688	Cannot change datatype for column %s from a character type to a non-character type	336068818L
-689	Page <*long*> is of wrong type (expected <*long*>, found <*long*>)	335544403L
-689	Wrong page type	335544650L
-690	Segments not allowed in expression index <*string*>	335544679L
-691	New record size of <*long*> bytes is too big	335544681L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-692	Maximum indexes per table (<*digit*>) exceeded	335544477L
-693	Too many concurrent executions of the same request	335544663L
-694	Cannot access column <*string*> in view <*string*>	335544684L
-802	Arithmetic exception, numeric overflow, or string truncation	335544321L
-803	Attempt to store duplicate value (visible to active transactions) in unique index "<*string*>"	335544349L
-803	Violation of PRIMARY or UNIQUE KEY constraint: "<*string*>"	335544665L
-804	Wrong number of arguments on call	335544380L
-804	SQLDA missing or incorrect version, or incorrect number/type of variables	335544583L
-804	Count of columns not equal count of values	335544584L
-804	Function unknown	335544586L
-804	Client SQL dialect %d does not support reference to %s datatype	335544796L
-806	Only simple column names permitted for VIEW WITH CHECK OPTION	335544600L
-807	No where clause for VIEW WITH CHECK OPTION	335544601L
-808	Only one table allowed for VIEW WITH CHECK OPTION	335544602L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-809	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION	335544603L
-810	No subqueries permitted for VIEW WITH CHECK OPTION	335544605L
-811	Multiple rows in singleton select	335544652L
-816	I/O error for file <string> -Error while trying to write to file -Cannot insert because the file is read-only or is on a read only medium	335544651L
-817	Attempted update during read-only transaction	335544361L
-817	Attempted write to read-only blob	335544371L
-817	Operation not supported	335544444L
-817	Attempted to update read-only database	335544445L
-817	Metadata update statement is not supported by the current database SQL dialect %d	335544793L
-820	Metadata is obsolete	335544356L
-820	Unsupported on-disk structure for file <*string*>; found <*long*>, support <*long*>	335544379L
-820	Wrong DYN version	335544437L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-820	Minor version too high found <*long*> expected <*long*>	335544467L
-823	Invalid bookmark handle	335544473L
-824	Invalid lock level <*digit*>	335544474L
-825	Invalid lock handle	335544519L
-826	Invalid statement handle	335544585L
-827	Invalid direction for find operation	335544655L
-828	Invalid key position	335544678L
-829	Invalid column reference	335544616L
-830	Column used with aggregate	335544615L
-831	Attempt to define a second PRIMARY KEY for the same table	335544548L
-832	FOREIGN KEY column count does not match PRIMARY KEY	335544604L
-833	Expression evaluation not supported	335544606L
-834	Refresh range number <*long*> not found	335544508L
-835	Bad checksum	335544649L
-836	Exception <*digit*>	335544517L
-837	Restart shared cache manager	335544518L
-838	Database <*string*> shutdown in <*digit*> seconds	335544560L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-839	journal file wrong format	335544686L
-840	Intermediate journal file full	335544687L
-841	Too many versions	335544677L
-842	Precision should be greater than 0	335544697L
-842	Scale cannot be greater than precision	335544698L
-842	Short integer expected	335544699L
-842	Long integer expected	335544700L
-842	Unsigned short integer expected	335544701L
-901	-mode “read_only” or “read_write”	335544109L
-901	“read_only” or “read_write” required	335544110L
-901	-mode access “read_only” or “read_write”	335544278L
-901	“read_only” or “read_write” required	335544279L
-901	Invalid database key	335544322L
-901	Unrecognized database parameter block	335544326L
-901	Invalid blob handle	335544328L
-901	Invalid Blob ID	335544329L
-901	Invalid parameter in transaction parameter block	335544330L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-901	Invalid format for transaction parameter block	335544331L
-901	Invalid transaction handle (expecting explicit transaction start)	335544332L
-901	Attempt to start more than <*long*> transactions	335544337L
-901	Information type inappropriate for object specified	335544339L
-901	No information of this type available for object specified	335544340L
-901	Unknown information item	335544341L
-901	Action cancelled by trigger (<*long*>) to preserve data integrity	335544342L
-901	Lock conflict on no wait transaction	335544345L
-901	Program attempted to exit without finishing database	335544350L
-901	Transaction is not in limbo	335544353L
-901	Blob was not closed	335544355L
-901	Cannot disconnect database with open transactions (<*long*> active)	335544357L
-901	Message length error (encountered <*long*>, expected <*long*>)	335544358L
-901	No transaction for request	335544363L
-901	Request synchronization error	335544364L
-901	Request referenced an unavailable database	335544365L
-901	Attempted read of a new, open blob	335544369L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-901	Attempted action on blob outside transaction	335544370L
-901	Attempted reference to blob in unavailable database	335544372L
-901	Table <*string*> was omitted from the transaction reserving list	335544376L
-901	Request includes a DSRI extension not supported in this implementation	335544377L
-901	Feature is not supported	335544378L
-901	<string>	335544382L
-901	Unrecoverable conflict with limbo transaction <*long*>	335544383L
-901	Internal error	335544392L
-901	Database handle not zero	335544407L
-901	Transaction handle not zero	335544408L
-901	Transaction in limbo	335544418L
-901	Transaction not in limbo	335544419L
-901	Transaction outstanding	335544420L
-901	Undefined message number	335544428L
-901	Blocking signal has been received	335544431L
-901	Database system cannot read argument <*long*>	335544442L
-901	Database system cannot write argument <*long*>	335544443L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-901	<string>	335544450L
-901	Transaction <*long*> is <*string*>	335544468L
-901	Invalid statement handle	335544485L
-901	Lock time-out on wait transaction	335544510L
-901	Invalid service handle	335544559L
-901	Wrong version of service parameter block	335544561L
-901	Unrecognized service parameter block	335544562L
-901	Service <*string*> is not defined	335544563L
-901	INDEX <*string*>	335544609L
-901	EXCEPTION <*string*>	335544610L
-901	Column <*string*>	335544611L
-901	Union not supported	335544613L
-901	Unsupported DSQL construct	335544614L
-901	Illegal use of keyword VALUE	335544623L
-901	Table <*string*>	335544626L
-901	Procedure <*string*>	335544627L
-901	Specified domain or source column does not exist	335544641L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-901	Variable <*string*> conflicts with parameter in same procedure	335544656L
-901	Server version too old to support all CREATE DATABASE options	335544666L
-901	Cannot delete	335544673L
-901	Sort error	335544675L
-901	Cannot delete rows from external files.	335544786L
-901	Cannot update rows in external files.	335544787L
-901	Datatype not supported for arithmetic	335544801L
-901	size specification either missing or incorrect for file <string>	336330754L
-901	file <string> out of sequence	336331015L
-901	can't join -- one of the files missing	336331016L
-901	standard input is not supported when using join operation	336331017L
-901	standard output is not supported when using split operation	336331018L
-902	Internal isc software consistency check (<*string*>)	335544333L
-902	Database file appears corrupt (<*string*>)	335544335L
-902	I/O error during “<*string*>” operation for file “<*string*>”	335544344L
-902	Corrupt system table	335544346L
-902	Operating system directive <*string*> failed	335544373L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-902	Internal error	335544384L
-902	Internal error	335544385L
-902	Internal error	335544387L
-902	Block size exceeds implementation restriction	335544388L
-902	Incompatible version of on-disk structure	335544394L
-902	Internal error	335544397L
-902	Internal error	335544398L
-902	Internal error	335544399L
-902	Internal error	335544400L
-902	Internal error	335544401L
-902	Internal error	335544402L
-902	Database corrupted	335544404L
-902	Checksum error on database page <*long*>	335544405L
-902	Index is broken	335544406L
-902	Transaction--request mismatch (synchronization error)	335544409L
-902	Bad handle count	335544410L
-902	Wrong version of transaction parameter block	335544411L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-902	Unsupported BLR version (expected <*long*>, encountered <*long*>)	335544412L
-902	Wrong version of database parameter block	335544413L
-902	Database corrupted	335544415L
-902	Internal error	335544416L
-902	Internal error	335544417L
-902	Internal error	335544422L
-902	Internal error	335544423L
-902	Lock manager error	335544432L
-902	SQL error code = <*long*>	335544436L
-902		335544448L
-902		335544449L
-902	Cache buffer for page <*long*> invalid	335544470L
-902	There is no index in table <*string*> with id <*digit*>	335544471L
-902	Your user name and password are not defined. Ask your database administrator to set up a Firebird login.	335544472L
-902	Enable journal for database before starting online dump	335544478L
-902	Online dump failure. Retry dump	335544479L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-902	An online dump is already in progress	335544480L
-902	No more disk/tape space. Cannot continue online dump	335544481L
-902	Maximum number of online dump files that can be specified is 16	335544483L
-902	Database <*string*> shutdown in progress	335544506L
-902	Long-term journaling already enabled	335544520L
-902	Database <*string*> shutdown	335544528L
-902	Database shutdown unsuccessful	335544557L
-902	Cannot attach to password database	335544653L
-902	Cannot start transaction for password database	335544654L
-902	Long-term journaling not enabled	335544564L
-902	Dynamic SQL Error	335544569L
-902	Expression evaluation not supported/old	335544606L
-904	Invalid database handle (no active connection)	335544324L
-904	Unavailable database	335544375L
-904	Implementation limit exceeded	335544381L
-904	Too many requests	335544386L
-904	Buffer exhausted	335544389L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-904	Buffer in use	335544391L
-904	Request in use	335544393L
-904	No lock manager available	335544424L
-904	Unable to allocate memory from operating system	335544430L
-904	Update conflicts with concurrent update	335544451L
-904	Object <*string*> is in use	335544453L
-904	Cannot attach active shadow file	335544455L
-904	A file in manual shadow <*long*> is unavailable	335544460L
-904	Cannot add index, index root page is full.	335544661L
-904	Sort error: not enough memory	335544676L
-904	Request depth exceeded. (Recursive definition?)	335544683L
-906	Product <*string*> is not licensed	335544452L
-909	Drop database completed with errors	335544667L
-911	Record from transaction <*long*> is stuck in limbo	335544459L
-913	Deadlock	335544336L
-922	File <*string*> is not a valid database	335544323L
-923	Connection rejected by remote interface	335544421L

TABLE 8–1 SQLCODE codes and messages (*continued*)

SQLCODE	SQLCODE text	Firebird number
-923	Secondary server attachments cannot validate databases	335544461L
-923	Secondary server attachments cannot start journaling	335544462L
-924	Bad parameters on attach or create database	335544325L
-924	Database detach completed with errors	335544441L
-924	Connection lost to pipe server	335544648L
-926	No rollback performed	335544447L
-999	Firebird error	335544689L

Firebird status array error codes

This section lists Firebird error codes and associated messages returned in the status array in the following tables. When code messages include the name of a database object or object type, the name is represented by a code in the Message column:

- <*string*>: String value, such as the name of a database object or object type.
- <*digit*>: Integer value, such as the identification number or code of a database object or object type.
- <*long*>: Long integer value, such as the identification number or code of a database object or object type.

Table 9 lists SQL Status Array codes for ESQL, DSQL, and ISQL.

Note License error codes should not occur in Firebird 1. However, these error messages can still appear when a Firebird 1 client accesses an earlier version remote Firebird server.

TABLE 9 Firebird status array error codes

Error code	GDSCODE	Number	Message
isc_arith_except	*arith_except*	335544321L	arithmetic exception, numeric overflow, or string truncation
isc_bad_dbkey	*bad_dbkey*	335544322L	invalid database key
isc_bad_db_format	NOT APPLICABLE	335544323L	file <*string*> is not a valid database
isc_bad_db_handle	NOT APPLICABLE	335544324L	invalid database handle (no active connection)
isc_bad_dpb_content	NOT APPLICABLE	335544325L	bad parameters on attach or create database
isc_bad_dpb_form	NOT APPLICABLE	335544326L	unrecognized database parameter block
isc_bad_req_handle	*bad_req_handle*	335544327L	invalid request handle

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_bad_segstr_handle	*bad_segstr_handle*	335544328L	invalid blob handle
isc_bad_segstr_id	*bad_segstr_id*	335544329L	invalid Blob ID
isc_bad_tpb_content	NOT APPLICABLE	335544330L	invalid parameter in transaction parameter block
isc_bad_tpb_form	NOT APPLICABLE	335544331L	invalid format for transaction parameter block
isc_bad_trans_handle	NOT APPLICABLE	335544332L	invalid transaction handle (expecting explicit transaction start)
isc_bug_check	*bug_check*	335544333L	internal isc software consistency check (<*string*>)
isc_convert_error	*convert_error*	335544334L	conversion error from string "<*string*>"
isc_db_corrupt	*db_corrupt*	335544335L	database file appears corrupt (<*string*>)
isc_deadlock	*deadlock*	335544336L	deadlock
isc_excess_trans	NOT APPLICABLE	335544337L	attempt to start more than <*long*> transactions
isc_from_no_match	*from_no_match*	335544338L	no match for first value expression
isc_infinap	*infinap*	335544339L	information type inappropriate for object specified

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_infona	*infona*	335544340L	no information of this type available for object specified
isc_infunk	*infunk*	335544341L	unknown information item
isc_integ_fail	*integ_fail*	335544342L	action cancelled by trigger (<*long*>) to preserve data integrity
isc_invalid_blr	*invalid_blr*	335544343L	invalid request BLR at offset <*long*>
isc_io_error	*io_error*	335544344L	I/O error during "<*string*>" operation for file "<*string*>"
isc_lock_conflict	*lock_conflict*	335544345L	lock conflict on no wait transaction
isc_metadata_corrupt	*metadata_corrupt*	335544346L	corrupt system table
isc_not_valid	*not_valid*	335544347L	validation error for column <*string*>, value "<*string*>"
isc_no_cur_rec	*no_cur_rec*	335544348L	no current record for fetch operation
isc_no_dup	*no_dup*	335544349L	attempt to store duplicate value (visible to active transactions) in unique index "<*string*>"
isc_no_finish	NOT APPLICABLE	335544350L	program attempted to exit without finishing database
isc_no_meta_update	NOT APPLICABLE	335544351L	unsuccessful metadata update
isc_no_priv	*no_priv*	335544352L	no permission for <*string*> access to <*string*> <*string*>

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_no_recon	NOT APPLICABLE	335544353L	transaction is not in limbo
isc_no_record	*no_record*	335544354L	invalid database key
isc_no_segstr_close	*no_segstr_close*	335544355L	Blob was not closed
isc_obsolete_metadata	*obsolete_metadata*	335544356L	metadata is obsolete
isc_open_trans	NOT APPLICABLE	335544357L	cannot disconnect database with open transactions (<*long*> active)
isc_port_len	*port_len*	335544358L	message length error (encountered <*long*>, expected <*long*>)
isc_read_only_field	*read_only_field*	335544359L	attempted update of read-only column
isc_read_only_rel	*read_only_rel*	335544360L	attempted update of read-only table
isc_read_only_trans	*read_only_trans*	335544361L	attempted update during read-only transaction
isc_read_only_view	*read_only_view*	335544362L	cannot update read-only view <*string*>
isc_req_no_trans	NOT APPLICABLE	335544363L	no transaction for request
isc_req_sync	*req_sync*	335544364L	request synchronization error
isc_req_wrong_db	NOT APPLICABLE	335544365L	request referenced an unavailable database
isc_segment	*segment*	335544366L	segment buffer length shorter than expected

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_segstr_eof	*segstr_eof*	335544367L	attempted retrieval of more segments than exist
isc_segstr_no_op	*segstr_no_op*	335544368L	attempted invalid operation on a blob
isc_segstr_no_read	*segstr_no_read*	335544369L	attempted read of a new, open blob
isc_segstr_no_trans	NOT APPLICABLE	335544370L	attempted action on blob outside transaction
isc_segstr_no_write	*segstr_no_write*	335544371L	attempted write to read-only blob
isc_segstr_wrong_db	NOT APPLICABLE	335544372L	attempted reference to blob in unavailable database
isc_sys_request	*sys_request*	335544373L	operating system directive <*string*> failed
isc_stream_eof	*stream_eof*	335544374L	attempt to fetch past the last record in a record stream
isc_unavailable	NOT APPLICABLE	335544375L	unavailable database
isc_unres_rel	NOT APPLICABLE	335544376L	Table <*string*> was omitted from the transaction reserving list
isc_uns_ext	NOT APPLICABLE	335544377L	request includes a DSRI extension not supported in this implementation
isc_wish_list	NOT APPLICABLE	335544378L	feature is not supported

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_wrong_ods	NOT APPLICABLE	335544379L	unsupported on-disk structure for file <*string*>; found <*long*>, support <*long*>
isc_wronumarg	*wronumarg*	335544380L	wrong number of arguments on call
isc_imp_exc	*imp_exc*	335544381L	Implementation limit exceeded
isc_random	*random*	335544382L	<*string*>
isc_fatal_conflict	*fatal_conflict*	335544383L	unrecoverable conflict with limbo transaction <*long*>
isc_badblk	*badblk*	335544384L	internal error
isc_invpoolcl	*invpoolcl*	335544385L	internal error
isc_nopoolids	*nopoolids*	335544386L	too many requests
isc_relbadblk	*relbadblk*	335544387L	internal error
isc_blktoobig	*blktoobig*	335544388L	block size exceeds implementation restriction
isc_bufexh	*bufexh*	335544389L	buffer exhausted
isc_syntaxerr	*syntaxerr*	335544390L	BLR syntax error: expected <*string*> at offset <*long*>, encountered <*long*>
isc_bufinuse	*bufinuse*	335544391L	buffer in use
isc_bdbincon	*bdbincon*	335544392L	internal error
isc_reqinuse	*reqinuse*	335544393L	request in use

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_badodsver	NOT APPLICABLE	335544394L	incompatible version of on-disk structure
isc_relnotdef	NOT APPLICABLE	335544395L	table <*string*> is not defined
isc_fldnotdef	NOT APPLICABLE	335544396L	column <*string*> is not defined in table <*string*>
isc_dirtypage	*dirtypage*	335544397L	internal error
isc_waifortra	*waifortra*	335544398L	internal error
isc_doubleloc	*doubleloc*	335544399L	internal error
isc_nodnotfnd	*nodnotfnd*	335544400L	internal error
isc_dupnodfnd	*dupnodfnd*	335544401L	internal error
isc_locnotmar	*locnotmar*	335544402L	internal error
isc_badpagtyp	*badpagtyp*	335544403L	page <*long*> is of wrong type (expected <*long*>, found <*long*>)
isc_corrupt	*corrupt*	335544404L	database corrupted
isc_badpage	NOT APPLICABLE	335544405L	checksum error on database page <*long*>
isc_badindex	*badindex*	335544406L	index is broken
isc_dbbnotzer	NOT APPLICABLE	335544407L	database handle not zero
isc_tranotzer	NOT APPLICABLE	335544408L	transaction handle not zero

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_trareqmis	NOT APPLICABLE	335544409L	transaction—request mismatch (synchronization error)
isc_badhndcnt	NOT APPLICABLE	335544410L	bad handle count
isc_wrotpbver	NOT APPLICABLE	335544411L	wrong version of transaction parameter block
isc_wroblrver	*wroblrver*	335544412L	unsupported BLR version (expected <*long*>, encountered <*long*>)
isc_wrodpbver	NOT APPLICABLE	335544413L	wrong version of database parameter block
isc_blobnotsup	NOT APPLICABLE	335544414L	Blob and array datatypes are not supported for <*string*> operation
isc_badrelation	NOT APPLICABLE	335544415L	database corrupted
isc_nodetach	*nodetach*	335544416L	internal error
isc_notremote	*notremote*	335544417L	internal error
isc_trainlim	NOT APPLICABLE	335544418L	transaction in limbo
isc_notinlim	NOT APPLICABLE	335544419L	transaction not in limbo
isc_traoutsta	NOT APPLICABLE	335544420L	transaction outstanding
isc_connect_reject	NOT APPLICABLE	335544421L	connection rejected by remote interface
isc_dbfile	*dbfile*	335544422L	internal error

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_orphan	*orphan*	335544423L	internal error
isc_no_lock_mgr	*no_lock_mgr*	335544424L	no lock manager available
isc_ctxinuse	*ctxinuse*	335544425L	context already in use (BLR error)
isc_ctxnotdef	*ctxnotdef*	335544426L	context not defined (BLR error)
isc_datnotsup	NOT APPLICABLE	335544427L	data operation not supported
isc_badmsgnum	*badmsgnum*	335544428L	undefined message number
isc_badparnum	NOT APPLICABLE	335544429L	bad parameter number
isc_virmemexh	*virmemexh*	335544430L	unable to allocate memory from operating system
isc_blocking_signal	*blocking_signal*	335544431L	blocking signal has been received
isc_lockmanerr	*lockmanerr*	335544432L	lock manager error
isc_journerr	NOT APPLICABLE	335544433L	communication error with journal "<*string*>"
isc_keytoobig	NOT APPLICABLE	335544434L	key size exceeds implementation restriction for index "<*string*>"
isc_nullsegkey	*nullsegkey*	335544435L	null segment of UNIQUE KEY
isc_sqlerr	*sqlerr*	335544436L	SQL error code = <*long*>
isc_wrodynver	NOT APPLICABLE	335544437L	wrong DYN version
isc_funnotdef	*funnotdef*	335544438L	function <*string*> is not defined

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_funmismat	*funmismat*	335544439L	function <*string*> could not be matched
isc_bad_msg_vec	*bad_msg_vec*	335544440L	
isc_bad_detach	NOT APPLICABLE	335544441L	database detach completed with errors
isc_noargacc_read	*noargacc_read*	335544442L	database system cannot read argument <*long*>
isc_noargacc_write	*noargacc_write*	335544443L	database system cannot write argument <*long*>
isc_read_only	NOT APPLICABLE	335544444L	operation not supported
isc_ext_err	*ext_err*	335544445L	<*string*> extension error
isc_non_updatable	*non_updatable*	335544446L	not updatable
isc_no_rollback	NOT APPLICABLE	335544447L	no rollback performed
isc_bad_sec_info	*bad_sec_info*	335544448L	
isc_invalid_sec_info	*invalid_sec_info*	335544449L	
isc_misc_interpreted	*minterpreted*	335544450L	<string>
isc_update_conflict	*update_conflict*	335544451L	update conflicts with concurrent update
isc_unlicensed	NOT APPLICABLE	335544452L	product <*string*> is not licensed
isc_obj_in_use	*obj_in_use*	335544453L	object <*string*> is in use

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_nofilter	*nofilter*	335544454L	filter not found to convert type <*long*> to type <*long*>
isc_shadow_accessed	NOT APPLICABLE	335544455L	cannot attach active shadow file
isc_invalid_sdl	*invalid_sdl*	335544456L	invalid slice description language at offset <*long*>
isc_out_of_bounds	*out_of_bounds*	335544457L	subscript out of bounds
isc_invalid_dimension	*invalid_dimension*	335544458L	column not array or invalid dimensions (expected <*long*>, encountered <*long*>)
isc_rec_in_limbo	*rec_in_limbo*	335544459L	record from transaction <*long*> is stuck in limbo
isc_shadow_missing	NOT APPLICABLE	335544460L	a file in manual shadow <*long*> is unavailable
isc_cant_validate	NOT APPLICABLE	335544461L	secondary server attachments cannot validate databases
isc_cant_start_journal	NOT APPLICABLE	335544462L	secondary server attachments cannot start journaling
isc_gennotdef	*gennotdef*	335544463L	generator <*string*> is not defined
isc_cant_start_logging	NOT APPLICABLE	335544464L	secondary server attachments cannot start logging
isc_bad_segstr_type	*bad_segstr_type*	335544465L	invalid blob type for operation

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_foreign_key	*foreign_key*	335544466L	violation of FOREIGN KEY constraint: "<*string*>"
isc_high_minor	NOT APPLICABLE	335544467L	minor version too high found <*long*> expected <*long*>
isc_tra_state	*tra_state*	335544468L	transaction <*long*> is <*string*>
isc_trans_invalid	*trans_invalid*	335544469L	transaction marked invalid by I/O error
isc_buf_invalid	*buf_invalid*	335544470L	cache buffer for page <*long*> invalid
isc_indexnotdefined	NOT APPLICABLE	335544471L	there is no index in table <*string*> with id <*digit*>
isc_login	NOT APPLICABLE	335544472L	Your user name and password are not defined. Ask your database administrator to set up a Firebird login.
isc_invalid_bookmark	*invalid_bookmark*	335544473L	invalid bookmark handle
isc_bad_lock_level	*bad_lock_level*	335544474L	invalid lock level <*digit*>
isc_relation_lock	*relation_lock*	335544475L	lock on table <*string*> conflicts with existing lock
isc_record_lock	*record_lock*	335544476L	requested record lock conflicts with existing lock
isc_max_idx	NOT APPLICABLE	335544477L	maximum indexes per table (<*digit*>) exceeded

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_jrn_enable	NOT APPLICABLE	335544478L	enable journal for database before starting online dump
isc_old_failure	NOT APPLICABLE	335544479L	online dump failure. Retry dump
isc_old_in_progress	NOT APPLICABLE	335544480L	an online dump is already in progress
isc_old_no_space	NOT APPLICABLE	335544481L	no more disk/tape space. Cannot continue online dump
isc_num_old_files	NOT APPLICABLE	335544483L	maximum number of online dump files that can be specified is 16
isc_bad_stmt_handle	*bad_stmt_handle*	335544485L	invalid statement handle
isc_stream_not_defined	*stream_not_defined*	335544502L	reference to invalid stream number
isc_shutinprog	*shutinprog*	335544506L	database <*string*> shutdown in progress
isc_range_in_use	*range_in_use*	335544507L	refresh range number <*long*> already in use
isc_range_not_found	*range_not_found*	335544508L	refresh range number <*long*> not found
isc_charset_not_found	*charset_not_found*	335544509L	character set <*string*> is not defined
isc_lock_timeout	*lock_timeout*	335544510L	lock time-out on wait transaction
isc_prcnotdef	*prcnotdef*	335544511L	procedure <*string*> is not defined

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_prcmismat	*prcmismat*	335544512L	parameter mismatch for procedure <*string*>
isc_codnotdef	*codnotdef*	335544515L	status code <*string*> unknown
isc_xcpnotdef	*xcpnotdef*	335544516L	exception <*string*> not defined
isc_except	*except*	335544517L	exception <*digit*>
isc_cache_restart	NOT APPLICABLE	335544518L	restart shared cache manager
isc_bad_lock_handle	*bad_lock_handle*	335544519L	invalid lock handle
isc_shutdown	*shutdown*	335544528L	database <*string*> shutdown
isc_existing_priv_mod	NOT APPLICABLE	335544529L	cannot modify an existing user privilege
isc_primary_key_ref	NOT APPLICABLE	335544530L	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.
isc_primary_key_notnull	*primary_key_notnull*	335544531L	Column used in a PRIMARY/UNIQUE constraint must be NOT NULL.
isc_ref_cnstrnt_notfound	NOT APPLICABLE	335544532L	Name of Referential Constraint not defined in constraints table.
isc_foreign_key_notfound	*foreign_key_notfound*	335544533L	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.
isc_ref_cnstrnt_update	NOT APPLICABLE	335544534L	Cannot update constraints (RDB$REF_CONSTRAINTS).

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_check_cnstrnt_update	NOT APPLICABLE	335544535L	Cannot update constraints (RDB$CHECK_CONSTRAINTS).
isc_check_cnstrnt_del	NOT APPLICABLE	335544536L	Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)
isc_integ_index_seg_del	*integ_index_seg_del*	335544537L	Cannot delete index segment used by an Integrity Constraint
isc_integ_index_seg_mod	*integ_index_seg_mod*	335544538L	Cannot update index segment used by an Integrity Constraint
isc_integ_index_del	*integ_index_del*	335544539L	Cannot delete index used by an Integrity Constraint
isc_integ_index_mod	*integ_index_mod*	335544540L	Cannot modify index used by an Integrity Constraint
isc_check_trig_del	NOT APPLICABLE	335544541L	Cannot delete trigger used by a CHECK Constraint
isc_check_trig_update	NOT APPLICABLE	335544542L	Cannot update trigger used by a CHECK Constraint
isc_cnstrnt_fld_del	NOT APPLICABLE	335544543L	Cannot delete column being used in an Integrity Constraint.
isc_cnstrnt_fld_rename	NOT APPLICABLE	335544544L	Cannot rename column being used in an Integrity Constraint.
isc_rel_cnstrnt_update	NOT APPLICABLE	335544545L	Cannot update constraints (RDB$RELATION_CONSTRAINTS).

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_constaint_on_view	NOT APPLICABLE	335544546L	Cannot define constraints on views
isc_invld_cnstrnt_type	*invld_cnstrnt_type*	335544547L	internal isc software consistency check (invalid RDB$CONSTRAINT_TYPE)
isc_primary_key_exists	NOT APPLICABLE	335544548L	Attempt to define a second PRIMARY KEY for the same table
isc_systrig_update	NOT APPLICABLE	335544549L	cannot modify or erase a system trigger
isc_not_rel_owner	NOT APPLICABLE	335544550L	only the owner of a table may reassign ownership
isc_grant_obj_notfound	NOT APPLICABLE	335544551L	could not find table/procedure for GRANT
isc_grant_fld_notfound	NOT APPLICABLE	335544552L	could not find column for GRANT
isc_grant_nopriv	*grant_nopriv*	335544553L	user does not have GRANT privileges for operation
isc_nonsql_security_rel	*nonsql_security_rel*	335544554L	table/procedure has non-SQL security class defined
isc_nonsql_security_fld	*nonsql_security_fld*	335544555L	column has non-SQL security class defined
isc_shutfail	*shutfail*	335544557L	database shutdown unsuccessful

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_check_constraint	*check_constraint*	335544558L	Operation violates CHECK constraint <*string*> on view or table
isc_bad_svc_handle	NOT APPLICABLE	335544559L	invalid service handle
isc_shutwarn	*shutwarn*	335544560L	database <*string*> shutdown in <*digit*> seconds
isc_wrospbver	NOT APPLICABLE	335544561L	wrong version of service parameter block
isc_bad_spb_form	NOT APPLICABLE	335544562L	unrecognized service parameter block
isc_svcnotdef	NOT APPLICABLE	335544563L	service <*string*> is not defined
isc_no_jrn	NOT APPLICABLE	335544564L	long-term journaling not enabled
isc_transliteration_failed	*transliteration_failed*	335544565L	Cannot transliterate character between character sets
isc_text_subtype	*text_subtype*	335544568L	Implementation of text subtype <*digit*> not located.
isc_dsql_error	*dsql_error*	335544569L	Dynamic SQL Error
isc_dsql_command_err	NOT APPLICABLE	335544570L	Invalid command
isc_dsql_constant_err	*dsql_constant_err*	335544571L	Datatype for constant unknown
isc_dsql_cursor_err	*dsql_cursor_err*	335544572L	Cursor unknown
isc_dsql_datatype_err	*dsql_datatype_err*	335544573L	Datatype unknown
isc_dsql_decl_err	*dsql_decl_err*	335544574L	Declared cursor already exists

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_dsql_cursor_update_err	*dsql_cursor_update_err*	335544575L	Cursor not updatable
isc_dsql_cursor_open_err	*dsql_cursor_open_err*	335544576L	Attempt to reopen an open cursor
isc_dsql_cursor_close_err	*dsql_cursor_close_err*	335544577L	Attempt to reclose a closed cursor
isc_dsql_field_err	NOT APPLICABLE	335544578L	Column unknown
isc_dsql_internal_err	*dsql_internal_err*	335544579L	Internal error
isc_dsql_relation_err	NOT APPLICABLE	335544580L	Table unknown
isc_dsql_procedure_err	NOT APPLICABLE	335544581L	Procedure unknown
isc_dsql_request_err	NOT APPLICABLE	335544582L	Request unknown
isc_dsql_sqlda_err	NOT APPLICABLE	335544583L	SQLDA missing or incorrect version, or incorrect number/type of variables
isc_dsql_var_count_err	NOT APPLICABLE	335544584L	Count of columns not equal count of values
isc_dsql_stmt_handle	*dsql_stmt_handle*	335544585L	Invalid statement handle
isc_dsql_function_err	*dsql_function_err*	335544586L	Function unknown
isc_dsql_blob_err	NOT APPLICABLE	335544587L	Column is not a blob
isc_collation_not_found	*collation_not_found*	335544588L	COLLATION <*string*> is not defined
isc_collation_not_for_charset	*collation_not_for_charset*	335544589L	COLLATION <*string*> is not valid for specified CHARACTER SET
isc_dsql_dup_option	NOT APPLICABLE	335544590L	Option specified more than once

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_dsql_tran_err	NOT APPLICABLE	335544591L	Unknown transaction option
isc_dsql_invalid_array	*dsql_invalid_array*	335544592L	Invalid array reference
isc_dsql_max_arr_dim_exceeded	*dsql_max_arr_dim_exceeded*	335544593L	Array declared with too many dimensions
isc_dsql_arr_range_error	*dsql_arr_range_error*	335544594L	Illegal array dimension range
isc_dsql_trigger_err	NOT APPLICABLE	335544595L	Trigger unknown
isc_dsql_subselect_err	*dsql_subselect_err*	335544596L	Subselect illegal in this context
isc_dsql_crdb_prepare_err	NOT APPLICABLE	335544597L	Cannot prepare a CREATE DATABASE/SCHEMA statement
isc_specify_field_err	NOT APPLICABLE	335544598L	must specify column name for view select expression
isc_num_field_err	NOT APPLICABLE	335544599L	number of columns does not match select list
isc_col_name_err	NOT APPLICABLE	335544600L	Only simple column names permitted for VIEW WITH CHECK OPTION
isc_where_err	NOT APPLICABLE	335544601L	No WHERE clause for VIEW WITH CHECK OPTION
isc_table_view_err	NOT APPLICABLE	335544602L	Only one table allowed for VIEW WITH CHECK OPTION

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_distinct_err	NOT APPLICABLE	335544603L	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION
isc_key_field_count_err	NOT APPLICABLE	335544604L	FOREIGN KEY column count does not match PRIMARY KEY
isc_subquery_err	NOT APPLICABLE	335544605L	No subqueries permitted for VIEW WITH CHECK OPTION
isc_expression_eval_err	NOT APPLICABLE	335544606L	expression evaluation not supported
isc_node_err	NOT APPLICABLE	335544607L	gen.c: node not supported
isc_command_end_err	NOT APPLICABLE	335544608L	Unexpected end of command
isc_index_name	*index_name*	335544609L	INDEX <*string*>
isc_exception_name	*Handled by* WHEN EXCEPTION name DO	335544610L	EXCEPTION <*string*>
isc_field_name	*field_name*	335544611L	COLUMN <*string*>
isc_token_err	NOT APPLICABLE	335544612L	Token unknown
isc_union_err	NOT APPLICABLE	335544613L	union not supported
isc_dsql_construct_err	NOT APPLICABLE	335544614L	Unsupported DSQL construct
isc_field_aggregate_err	*field_aggregate_err*	335544615L	column used with aggregate
isc_field_ref_err	NOT APPLICABLE	335544616L	invalid column reference

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_order_by_err	NOT APPLICABLE	335544617L	invalid ORDER BY clause
isc_return_mode_err	*return_mode_err*	335544618L	Return mode by value not allowed for this datatype
isc_extern_func_err	NOT APPLICABLE	335544619L	External functions cannot have more than 10 parameters
isc_alias_conflict_err	NOT APPLICABLE	335544620L	alias <*string*> conflicts with an alias in the same statement
isc_procedure_conflict_error	NOT APPLICABLE	335544621L	alias <*string*> conflicts with a procedure in the same statement
isc_relation_conflict_err	NOT APPLICABLE	335544622L	alias <*string*> conflicts with a table in the same statement
isc_dsql_domain_err	NOT APPLICABLE	335544623L	Illegal use of keyword VALUE
isc_idx_seg_err	NOT APPLICABLE	335544624L	segment count of 0 defined for index <*string*>
isc_node_name_err	NOT APPLICABLE	335544625L	A node name is not permitted in a secondary, shadow, cache or log file name
isc_table_name	*table_name*	335544626L	TABLE <*string*>
isc_proc_name	*proc_name*	335544627L	PROCEDURE <*string*>
isc_idx_create_err	NOT APPLICABLE	335544628L	cannot create index <*string*>
isc_dependency	NOT APPLICABLE	335544630L	there are <*long*> dependencies

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_idx_key_err	*idx_key_err*	335544631L	too many keys defined for index <*string*>
isc_dsql_file_length_err	NOT APPLICABLE	335544632L	Preceding file did not specify length, so <*string*> must include starting page number
isc_dsql_shadow_number_err	NOT APPLICABLE	335544633L	Shadow number must be a positive integer
isc_dsql_token_unk_err	NOT APPLICABLE	335544634L	Token unknown - line <*long*>, char <*long*>
isc_dsql_no_relation_alias	NOT APPLICABLE	335544635L	there is no alias or table named <*string*> at this scope level
isc_indexname	NOT APPLICABLE	335544636L	there is no index <*string*> for table <*string*>
isc_no_stream_plan	NOT APPLICABLE	335544637L	table <*string*> is not referenced in plan
isc_stream_twice	NOT APPLICABLE	335544638L	table <*string*> is referenced more than once in plan; use aliases to distinguish
isc_stream_not_found	NOT APPLICABLE	335544639L	table <*string*> is referenced in the plan but not the from list
isc_collation_requires_text	NOT APPLICABLE	335544640L	Invalid use of CHARACTER SET or COLLATE
isc_dsql_domain_not_found	NOT APPLICABLE	335544641L	Specified domain or source column does not exist

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_index_unused	NOT APPLICABLE	335544642L	index <*string*> cannot be used in the specified plan
isc_dsql_self_join	NOT APPLICABLE	335544643L	the table <*string*> is referenced twice; use aliases to differentiate
isc_stream_bof	*stream_bof*	335544644L	illegal operation when at beginning of stream
isc_stream_crack	*stream_crack*	335544645L	the current position is on a crack
isc_db_or_file_exists	NOT APPLICABLE	335544646L	database or file exists
isc_invalid_operator	NOT APPLICABLE	335544647L	invalid comparison operator for find operation
isc_conn_lost	*conn_lost*	335544648L	Connection lost to pipe server
isc_bad_checksum	*bad_checksum*	335544649L	bad checksum
isc_page_type_err	*page_type_err*	335544650L	wrong page type
isc_ext_readonly_err	*ext_readonly_err*	335544651L	external file could not be opened for output
isc_sing_select_err	*sing_select_err*	335544652L	multiple rows in singleton select
isc_psw_attach	NOT APPLICABLE	335544653L	cannot attach to password database
isc_psw_start_trans	NOT APPLICABLE	335544654L	cannot start transaction for password database
isc_invalid_direction	NOT APPLICABLE	335544655L	invalid direction for find operation

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_dsql_var_conflict	NOT APPLICABLE	335544656L	variable <*string*> conflicts with parameter in same procedure
isc_dsql_no_blob_array	NOT APPLICABLE	335544657L	Array/Blob/DATE /TIME/TIMESTAMP datatypes not allowed in arithmetic
isc_dsql_base_table	NOT APPLICABLE	335544658L	<*string*> is not a valid base table of the specified view
isc_duplicate_base_table	NOT APPLICABLE	335544659L	table <*string*> is referenced twice in view; use an alias to distinguish
isc_view_alias	NOT APPLICABLE	335544660L	view <*string*> has more than one base table; use aliases to distinguish
isc_index_root_page_full	NOT APPLICABLE	335544661L	cannot add index, index root page is full.
isc_dsql_blob_type_unknown	*dsql_blob_type_unknown*	335544662L	BLOB SUB_TYPE <*string*> is not defined
isc_req_max_clones_exceeded	*req_max_clones_exceeded*	335544663L	Too many concurrent executions of the same request
isc_dsql_duplicate_spec	NOT APPLICABLE	335544664L	duplicate specification of <*string*> - not supported
isc_unique_key_violation	*unique_key_violation*	335544665L	violation of PRIMARY or UNIQUE KEY constraint: "<*string*>"
isc_srvr_version_too_old	NOT APPLICABLE	335544666L	server version too old to support all CREATE DATABASE options

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_drdb_completed_with_errs	NOT APPLICABLE	335544667L	drop database completed with errors
isc_dsql_procedure_use_err	*dsql_procedure_use_err*	335544668L	procedure <*string*> does not return any values
isc_dsql_count_mismatch	NOT APPLICABLE	335544669L	count of column list and variable list do not match
isc_blob_idx_err	NOT APPLICABLE	335544670L	attempt to index blob column in index <*string*>
isc_array_idx_err	NOT APPLICABLE*r*	335544671L	attempt to index array column in index <*string*>
isc_key_field_err	NOT APPLICABLE	335544672L	too few key columns found for index <*string*> (incorrect column name?)
isc_no_delete	*no_delete*	335544673L	cannot delete
isc_del_last_field	NOT APPLICABLE	335544674L	last column in a table cannot be deleted
isc_sort_err	*sort_err*	335544675L	sort error
isc_sort_mem_err	*sort_mem_err*	335544676L	sort error: not enough memory
isc_version_err	*version_err*	335544677L	too many versions
isc_inval_key_posn	NOT APPLICABLE	335544678L	invalid key position
isc_no_segments_err	NOT APPLICABLE	335544679L	segments not allowed in expression index <*string*>

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_crrp_data_err	*crrp_data_err*	335544680L	sort error: corruption in data structure
isc_rec_size_err	NOT APPLICABLE	335544681L	new record size of <*long*> bytes is too big
isc_dsql_field_ref	NOT APPLICABLE	335544682L	Inappropriate self-reference of column
isc_req_depth_exceeded	*req_depth_exceeded*	335544683L	request depth exceeded. (Recursive definition?)
isc_no_field_access	*no_field_access*	335544684L	cannot access column <*string*> in view <*string*>
isc_no_dbkey	NOT APPLICABLE	335544685L	dbkey not available for multi-table views
isc_dsql_open_cursor_request	NOT APPLICABLE	335544688L	The prepare statement identifies a prepare statement with an open cursor
isc_ib_error	*ib_error*	335544689L	Firebird error
isc_cache_redef	NOT APPLICABLE	335544690L	Cache redefined
isc_cache_too_small	NOT APPLICABLE	335544691L	Cache length too small
isc_precision_err	NOT APPLICABLE	335544697L	Precision should be greater than 0
isc_scale_nogt	NOT APPLICABLE	335544698L	Scale cannot be greater than precision
isc_expec_short	*expec_short*	335544699L	Short integer expected
isc_expec_long	*expec_long*	335544700L	Long integer expected
isc_expec_ushort	*expec_ushort*	335544701L	Unsigned short integer expected

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_like_escape_invalid	*like_escape_invalid*	335544702L	Invalid ESCAPE sequence
isc_svcnoexe	NOT APPLICABLE	335544703L	service <*string*> does not have an associated executable
isc_net_lookup_err	*net_lookup_err*	335544704L	Network lookup failure for host "<*string*>"
isc_service_unknown	NOT APPLICABLE	335544705L	Undefined service <*string*>/<*string*>
isc_host_unknown	NOT APPLICABLE	335544706L	Host unknown
isc_grant_nopriv_on_base	*grant_nopriv_on_base*	335544707L	user does not have GRANT privileges on base table/view for operation
isc_dyn_fld_ambiguous	NOT APPLICABLE	335544708L	Ambiguous column reference.
isc_dsql_agg_ref_err	NOT APPLICABLE	335544709L	Invalid aggregate reference
isc_complex_view	NOT APPLICABLE	335544710L	navigational stream <*long*> references a view with more than one base table.
isc_unprepared_stmt	NOT APPLICABLE	335544711L	attempt to execute an unprepared dynamic SQL statement
isc_expec_positive	*expec_positive*	335544712L	Positive value expected.
isc_dsql_sqlda_value_err	NOT APPLICABLE	335544713L	Incorrect values within SQLDA structure
isc_invalid_array_id	*invalid_array_id*	335544714L	invalid blob id

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_ext_file_uns_op	NOT APPLICABLE	335544715L	operation not supported for EXTERNAL FILE table <*string*>
isc_svc_in_use	NOT APPLICABLE	335544716L	service is currently busy: <*string*>
isc_err_stack_limit	*err_stack_limit*	335544717L	stack size insufficient to execute current request
isc_invalid_key	*invalid_key*	335544718L	invalid key for find operation
isc_net_init_error	NOT APPLICABLE	335544719L	error initializing the network software
isc_loadlib_failure	*loadlib_failure*	335544720L	unable to load required library <*string*>
isc_network_error	*network_error*	335544721L	unable to complete network request to host "<*string*>"
isc_net_connect_err	NOT APPLICABLE	335544722L	failed to establish a connection
isc_net_connect_listen_err	NOT APPLICABLE	335544723L	error while listening for an incoming connection
isc_net_event_connect_err	NOT APPLICABLE	335544724L	failed to establish a secondary connection for event processing
isc_net_event_listen_err	NOT APPLICABLE	335544725L	error while listening for an incoming event connection request
isc_net_read_err	*net_read_err*	335544726L	error reading data from the connection
isc_net_write_err	*net_write_err*	335544727L	error writing data to the connection

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_integ_index_deactivate	NOT APPLICABLE	335544728L	cannot deactivate index used by an Integrity Constraint
isc_integ_deactivate_primary	NOT APPLICABLE	335544729L	cannot deactivate primary index
isc_unsupported_network_drive	NOT APPLICABLE	335544732L	access to databases on file servers is not supported
isc_io_create_err	NOT APPLICABLE	335544733L	error while trying to create file
isc_io_open_err	*io_open_err*	335544734L	error while trying to open file
isc_io_close_err	*io_close_err*	335544735L	error while trying to close file
isc_io_read_err	*io_read_err*	335544736L	error while trying to read from file
isc_io_write_err	*io_write_err*	335544737L	error while trying to write to file
isc_io_delete_err	NOT APPLICABLE	335544738L	error while trying to delete file
isc_io_access_err	*io_access_err*	335544739L	error while trying to access file
isc_udf_exception	*udf_exception*	335544740L	exception <*integer*> detected in blob filter or user defined function
isc_lost_db_connection	*lost_db_connection*	335544741L	connection lost to database
isc_no_write_user_priv	NOT APPLICABLE	335544742L	user cannot write to RDB$USER_PRIVILEGES
isc_token_too_long	*token_too_long*	335544743L	token size exceeds limit
isc_max_att_exceeded	NOT APPLICABLE	335544744L	maximum user count exceeded; contact your database administrator

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_login_same_as_role_name	NOT APPLICABLE	335544745L	your login <*string*> is same as one of the SQL role names; ask your database administrator to set up a valid Firebird login
isc_reftable_requires_pk	NOT APPLICABLE	335544746L	"REFERENCES table" without "(column)"; requires PRIMARY KEY on referenced table
isc_usrname_too_long	NOT APPLICABLE	335544747L	the username entered is too long. Maximum length is 31 bytes.
isc_password_too_long	NOT APPLICABLE	335544748L	the password specified is too long. Maximum length is 8 bytes.
isc_usrname_required	*usrname_required*	335544749L	a username is required for this operation.
isc_password_required	NOT APPLICABLE	335544750L	a password is required for this operation
isc_bad_protocol	NOT APPLICABLE	335544751L	the network protocol specified is invalid
isc_dup_usrname_found	NOT APPLICABLE	335544752L	a duplicate user name was found in the security database
isc_usrname_not_found	*usrname_not_found*	335544753L	the user name specified was not found in the security database
isc_error_adding_sec_record	NOT APPLICABLE	335544754L	error while attempting to add the user

TABLE 9 Firebird status array error codes (*continued*)

Error code	GDSCODE	Number	Message
isc_error_modifying_sec_record	NOT APPLICABLE	335544755L	error while attempting to modify the user record
isc_error_deleting_sec_record	NOT APPLICABLE	335544756L	error while attempting to delete the user record
eisc_rror_updating_sec_db	NOT APPLICABLE	335544757L	error while updating the security database
isc_sort_rec_size_err	*sort_rec_size_err*	335544758L	sort record size is too big
isc_bad_default_value	*bad_default_value*	335544759L	cannot assign a NULL default value to a column with a NOT NULL constraint
isc_invalid_clause	*invalid_clause*	335544760L	the specified user-entered string is not valid
isc_too_many_handles	*too_many_handles*	335544761L	too many open handles to database
isc_optimizer_blk_exc	NOT APPLICABLE	335544762L	optimizer implementation limits are exceeded; for example, only 256 conjuncts (ANDs and ORs) are allowed

CHAPTER 9

System Tables and Views

This topic describes the Firebird system tables and SQL system views.

Important Only Firebird system object names can begin with the characters "RDB$". No other object name in Firebird can begin with this character sequence, including tables, views, triggers, stored procedures, indexes, generators, domains, and roles.

Overview

The Firebird system tables contain and track metadata. Firebird automatically creates system tables when a database is created. Each time a user creates or modifies metadata through data definition, the SQL data definition utility automatically updates the system tables.
SQL system views provide information about existing integrity constraints for a database. You must create system views yourself by creating and running an ISQL script after database definition—see System views on page 393 for the code that creates them as well as the resulting table structures.
To see system tables, use this ISQL command:

```
SHOW SYSTEM TABLES;
```

The following ISQL command lists system views along with database views:

```
SHOW VIEWS;
```

System tables

This table lists the Firebird system tables. The names of system tables and their columns start with RDB$.

TABLE 9–1 List of system tables

RDB$CHARACTER_SETS	RDB$LOG_FILES
RDB$COLLATIONS	RDB$PAGES
RDB$CHECK_CONSTRAINTS	RDB$PROCEDURE_PARAMETERS
RDB$DATABASE	RDB$PROCEDURES
RDB$DEPENDENCIES	RDB$REF_CONSTRAINTS
RDB$EXCEPTIONS	RDB$RELATION_CONSTRAINTS
RDB$FIELD_DIMENSIONS	RDB$RELATION_FIELDS
RDB$FIELDS	RDB$RELATIONS
RDB$FILES	RDB$ROLES
RDB$FILTERS	RDB$SECURITY_CLASSES
RDB$FORMATS	RDB$TRANSACTIONS
RDB$FUNCTION_ARGUMENTS	RDB$TRIGGER_MESSAGES
RDB$FUNCTIONS	RDB$TRIGGERS
RDB$GENERATORS	RDB$TYPES
RDB$INDEX_SEGMENTS	RDB$USER_PRIVILEGES
RDB$INDICES	RDB$VIEW_RELATIONS

RDB$CHARACTER_SETS

RDB$CHARACTER_SETS describes the valid character sets available in Firebird.

TABLE 9–2 RDB$CHARACTER_SETS

Column name	Datatype	Length	Description
RDB$CHARACTER_SET_NAME	CHAR	31	Name of a character set that Firebird recognizes
RDB$FORM_OF_USE	CHAR	31	Reserved for internal use. Subtype 2
RDB$NUMBER_OF_CHARACTERS	INTEGER		Number of characters in a particular character set; for example, the set of Japanese characters
RDB$DEFAULT_COLLATE_NAME	CHAR	31	Subtype 2: default collation sequence for the character set
RDB$CHARACTER_SET_ID	SMALLINT		A unique identification for the character set
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether character set is: • User-defined (value of 0 or NULL) • System-defined (value of 1)
RDB$DESCRIPTION	BLOB	80	Subtype text: user-written description of the character set
RDB$FUNCTION_NAME	CHAR	31	Reserved for internal use; subtype 2
RDB$BYTES_PER_CHARACTER	SMALLINT		Size of character in bytes

RDB$CHECK_CONSTRAINTS

RDB$CHECK_CONSTRAINTS stores database integrity constraint information for CHECK constraints. In addition, the table stores information for constraints implemented with NOT NULL.

TABLE 9–3 RDB$CHECK_CONSTRAINTS

Column name	Datatype	Length	Description
RDB$CONSTRAINT_NAME	CHAR	31	Subtype 2: Name of a CHECK or NOT NULL constraint
RDB$TRIGGER_NAME	CHAR	31	Subtype 2: Name of the trigger that enforces the CHECK constraint; for a NOT NULL constraint, name of the source column in RDB$RELATION_FIELDS

RDB$COLLATIONS

RDB$COLLATIONS records the valid collating sequences available for use in Firebird.

TABLE 9–4 RDB$COLLATIONS

Column name	Datatype	Length	Description
RDB$COLLATION_NAME	CHAR	31	Name of a valid collation sequence in Firebird
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence
RDB$CHARACTER_SET_ID	SMALLINT		Identifier of the underlying character set of this collation sequence • Required before collation can proceed • Determines which character set is in use Corresponds to the RDB$CHARACTER_SET_ID column in the RDB$CHARACTER_SETS table
RDB$COLLATION_ATTRIBUTES	SMALLINT		Reserved for internal use
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the generator is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the collation sequence
RDB$FUNCTION_NAME	CHAR	31	Reserved for internal use

RDB$DATABASE

RDB$DATABASE defines a database.

TABLE 9–5 RDB$DATABASE

Column name	Datatype	Length	Description
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the database; when a comment is included in a CREATE or ALTER SCHEMA|DATABASE statement, ISQL writes to this column
RDB$RELATION_ID	SMALLINT		For internal use by Firebird
RDB$SECURITY_CLASS	CHAR	31	Subtype 2: Security class defined in the RDB$SECURITY_CLASSES table; the access control limits described in the named security class apply to all database usage
RDB$CHARACTER_SET_NAME	CHAR	31	Subtype 2; Name of character set

RDB$DEPENDENCIES

RDB$DEPENDENCIES keeps track of the tables and columns upon which other system objects depend. These objects include views, triggers, and computed columns. Firebird uses this table to ensure that a column or table cannot be deleted if it is used by any other object.

TABLE 9–6 RDB$DEPENDENCIES

Column name	Datatype	Length	Description
RDB$DEPENDENT_NAME	CHAR	31	Subtype 2; names the object this table tracks: a view, trigger, or computed column
RDB$DEPENDED_ON_NAME	CHAR	31	Subtype 2; names the table referenced by the object named above
RDB$FIELD_NAME	CHAR	31	Subtype 2; names the column referenced by the object named above
RDB$DEPENDENT_TYPE	SMALLINT		Describes the object type of the object referenced in the RDB$DEPENDENT_NAME column; type codes (RDB$TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure • 6 - expression_index • 7 - exception • 8 - user • 9 - field • 10 - index All other values are reserved for future use

TABLE 9–6 RDB$DEPENDENCIES

Column name	Datatype	Length	Description
RDB$DEPENDED_ON_TYPE	SMALLINT		Describes the object type of the object referenced in the RDB$DEPENDED_ON_NAME column; type codes (RDB$TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure • 6 - expression_index • 7 - exception • 8 - user • 9 - field • 10 - index All other values are reserved for future use

RDB$EXCEPTIONS

RDB$EXCEPTIONS describes error conditions related to stored procedures, including user-defined exceptions.

TABLE 9–7 RDB$EXCEPTIONS

Column name	Datatype	Length	Description
RDB$EXCEPTION_NAME	CHAR	31	Subtype 2; exception name
RDB$EXCEPTION_NUMBER	INTEGER		Number for the exception

TABLE 9–7 RDB$EXCEPTIONS

Column name	Datatype	Length	Description
RDB$MESSAGE	VARCHAR	78	Text of exception message
RDB$DESCRIPTION	BLOB	80	Subtype Text: Text description of the exception
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the exception is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$FIELD_DIMENSIONS

RDB$FIELD_DIMENSIONS describes each dimension of an array column.

TABLE 9–8 RDB$FIELD_DIMENSIONS

Column name	Datatype	Length	Description
RDB$FIELD_NAME	CHAR	31	Subtype 2; names the array column described by this table; the column name must exist in the RDB$FIELD_NAME column of RDB$FIELDS
RDB$DIMENSION	SMALLINT		Identifies one dimension of the ARRAY column; the first dimension is identified by the integer 0
RDB$LOWER_BOUND	INTEGER		Indicates the lower bound of the previously specified dimension
RDB$UPPER_BOUND	INTEGER		Indicates the upper bound of the previously specified dimension

RDB$FIELDS

RDB$FIELDS defines the characteristics of a column. Each domain or column has a corresponding row in RDB$FIELDS. Columns are added to tables by means of an entry in the RDB$RELATION_FIELDS table, which describes local characteristics.

For domains, RDB$FIELDS includes domain name, null status, and default values. SQL columns are defined in RDB$RELATION_FIELDS. For both domains and simple columns, RDB_RELATION_FIELDS can contain default and null status information.

TABLE 9–9 RDB$FIELDS

Column name	Datatype	Length	Description
RDB$FIELD_NAME	CHAR	31	Unique name of a domain or system-assigned name for a column: • RDB$*nnn* for user fields that did not specify a domain • specific name if the user previously created a domain that is used for this field • system field; one of the system-predefined domains that have meaningful names starting with RDB$, such as RDB$FILE_LENGTH. The actual column names are stored in the RDB$FIELD_SOURCE column of RDB$RELATION_FIELDS.
RDB$QUERY_NAME	CHAR	31	Not used for SQL objects.
RDB$VALIDATION_BLR	BLOB	80	Not used for SQL objects.
RDB$VALIDATION_SOURCE	BLOB	80	Not used for SQL objects.
RDB$COMPUTED_BLR	BLOB	80	Subtype BLR; for computed columns, contains the BLR (Binary Language Representation) of the expression the database evaluates at the time of execution.
RDB$COMPUTED_SOURCE	BLOB	80	Subtype Text: For computed columns, contains the original CHAR source expression for the column.
RDB$DEFAULT_VALUE	BLOB	80	Stores default rule; subtype BLR.

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$DEFAULT_SOURCE	BLOB	80	Subtype Text; SQL description of a default value.
RDB$FIELD_LENGTH	SMALLINT		Length in bytes of the field this row defines. For CHAR, VARCHAR, and NCHAR datatypes, this is the maximum length of the field. Firebird uses this length when creating indexes on columns. If this value is greater than 252, you cannot create an index on a column that uses this field definition. For non-CHAR related datatypes, the lengths are as follows: • D_FLOAT - 8 • DOUBLE - 8 • DATE - 4 • BLOB - 8 • TIME - 4 • INT64 - 8 • SHORT - 2 • LONG - 4 • QUAD - 8 • FLOAT - 4 • TIMESTAMP - 8
RDB$FIELD_PRECISION	SMALLINT		Stores the precision for numeric and decimal types.
RDB$FIELD_SCALE	SMALLINT		Stores negative scale for numeric and decimal types: the power of 10 by which the stored value is multiplied to get the intended value (for example, 1.234 = 1234 * 10^{-3}).

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$FIELD_TYPE	SMALLINT		Specifies the datatype of the column being defined; changing the value of this column automatically changes the datatype for all columns based on the column being defined. Valid values are: • BLOB - 261 • CHAR - 14 • CSTRING - 40 • D_FLOAT - 11 • DOUBLE - 27 • FLOAT - 10 • INT64 - 16 • INTEGER - 8 • QUAD - 9 • SMALLINT - 7 • DATE - 12 (dialect 3 DATE) • TIME - 13 • TIMESTAMP - 35 (DATE in older versions) • VARCHAR - 37 Restrictions: • The value of this column cannot be changed to or from blob • Non-numeric data causes a conversion error in a column changed from CHAR to numeric Changing data from CHAR to numeric and back again adversely affects index performance; for best results, delete and re-create indexes when making this type of change

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$FIELD_SUB_TYPE	SMALLINT		Used to distinguish types of BLOBs, CHARs, and integers **1** If RDB$FIELD_TYPE is 261 (BLOB), predefined subtypes can be: • 0 - unspecified • 1 - text • 2 - BLR (Binary Language Representation) • 3 - access control list • 4 - reserved for future use • 5 - encoded description of a table's current metadata • 6 - description of multi-database transaction that finished irregularly **2** If RDB$FIELD_TYPE is 14 (CHAR), columns can be: • 0 - type is unspecified • 1 - fixed BINARY data Corresponds to the RDB$FIELD_SUB_TYPE column in the RDB$COLLATIONS table **3** If RDB$FIELD_TYPE is 7 (SMALLINT), 8 (INTEGER), or 16 (INT64), the original declaration was: • 0 or NULL - RDB$FIELD_TYPE • 1 - NUMERIC • 2 - DECIMAL
RDB$MISSING_VALUE	BLOB	80	Not used for SQL objects

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$MISSING_SOURCE	BLOB	80	Not used for SQL objects
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the column being defined
RDB$SYSTEM_FLAG	SMALLINT		For system tables
RDB$QUERY_HEADER	BLOB	80	Not used for SQL objects
RDB$SEGMENT_LENGTH	SMALLINT		Used for blob columns only; a non-binding suggestion for the length of blob buffers
RDB$EDIT_STRING	VARCHAR	125	Not used for SQL objects
RDB$EXTERNAL_LENGTH	SMALLINT		Length of the column as it exists in an external table; if the column is not in an external table, this value is 0
RDB$EXTERNAL_SCALE	SMALLINT		Scale factor for an external column of an integer datatype; the scale factor is the power of 10 by which the integer is multiplied

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$EXTERNAL_TYPE	SMALLINT		Indicates the datatype of the column as it exists in an external table; valid values are: • SMALLINT - 7 • INTEGER - 8 • QUAD - 9 • FLOAT - 10 • D_FLOAT - 11 • CHAR - 14 • TIME • DOUBLE - 27 • DATE - 35 • VARCHAR - 37 • 'C' string (null terminated text) - 40 • BLOB - 261 • TIMESTAMP
RDB$DIMENSIONS	SMALLINT		For an ARRAY datatype, specifies the number of dimensions in the array; for a non-array column, the value is 0
RDB$NULL_FLAG	SMALLINT		Indicates whether a column can contain a NULL value Valid values are: • Empty: Can contain NULL values • 1: Cannot contain NULL values

TABLE 9–9 RDB$FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$CHARACTER_LENGTH	SMALLINT		Length in characters of the field this row defines. For CHAR, VARCHAR, and NCHAR datatypes, this is the quotient of RDB$FIELD_LENGTH divided by the number of bytes per character in the character set of the field. For other datatypes, this length value is not meaningful, and should be NULL
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence
RDB$CHARACTER_SET_ID	SMALLINT		ID indicating character set for the character or blob columns; joins to the CHARACTER_SET_ID column of the RDB$CHARACTER_SETS system table

RDB$FILES

RDB$FILES lists the secondary files and shadow files for a database.

TABLE 9–10 RDB$FILES

Column name	Datatype	Length	Description
RDB$FILE_NAME	VARCHAR	253	Names either a secondary file or a shadow file for the database
RDB$FILE_SEQUENCE	SMALLINT		*Either* the order that secondary files are to be used in the database or the order of files within a shadow set
RDB$FILE_START	INTEGER		Specifies the starting page number for a secondary file or shadow file
RDB$FILE_LENGTH	INTEGER		Specifies the file length in blocks
RDB$FILE_FLAGS	SMALLINT		Reserved for system use
RDB$SHADOW_NUMBER	SMALLINT		Set number: indicates to which shadow set the file belongs; if the value of this column is 0 or missing, Firebird assumes the file being defined is a secondary file, not a shadow file

RDB$FILTERS

RDB$FILTERS tracks information about a blob filter.

TABLE 9–11 RDB$FILTERS

Column name	Datatype	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name for the filter defined by this row
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the filter being defined
RDB$MODULE_NAME	VARCHAR	253	Names the library where the filter executable is stored
RDB$ENTRYPOINT	CHAR	31	The entry point within the filter library for the blob filter being defined
RDB$INPUT_SUB_TYPE	SMALLINT		The blob subtype of the input data (see Table 9–12)
RDB$OUTPUT_SUB_TYPE	SMALLINT		The blob subtype of the output data (see Table 9–12)
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the filter is: • User-defined (value of 0) • System-defined (value greater than 0)

Predefined blob subtypes are listed in Table 9–12. Subtypes 0 and 1 are intended for general use; all other blob subtypes with positive integer values are for system use only.

TABLE 9–12 Blob subtypes

ID	Name	Description
0		Unstructured, generally applied to binary data or data of an indeterminate type
1	TEXT	Unformatted text
2	BLR	Binary language representation of triggers, stored procedures, views, etc.
3	ACL	Access control list
4		(Reserved for future use)
5		Encoded description of a table's current metadata
6		Description of multi-database transaction that finished irregularly

Note Blob subtype names are reserved words only in the context of a blob subtype declaration. In all other Firebird contexts they are unreserved.

RDB$FORMATS

RDB$FORMATS keeps track of the format versions of the columns in a table. Firebird assigns the table a new format number at each change to a column definition. Direct metadata operations such as ALTER TABLE increment the format version; so do creating, dropping, activating, and deactivating triggers. This table allows existing application programs to access a changed table, without needing to be recompiled.

Note The format version number has an upper limit of 255; therefore a maximum of 255 metadata changes can be made to a table and its triggers. Once this limit is reached, no more metadata changes to that table are allowed until the database has been backed up and restored.

WARNING ***Not for user update***; modifying RDB$FORMATS in any way corrupts a database.

TABLE 9–13 RDB$FORMATS

Column name	Datatype	Length	Description
RDB$RELATION_ID	SMALLINT		Names a table that exists in RDB$RELATIONS
RDB$FORMAT	SMALLINT		Specifies the format number of the table; a table can have any number of different formats, depending on the number of updates to the table
RDB$DESCRIPTOR	BLOB	80	Subtype Format: Lists each column in the table, along with its datatype, length, and scale (if applicable)

RDB$FUNCTION_ARGUMENTS

RDB$FUNCTION_ARGUMENTS defines the attributes of a function argument.

TABLE 9–14 RDB$FUNCTION_ARGUMENTS

Column name	Datatype	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name of the function with which the argument is associated; must correspond to a function name in RDB$FUNCTIONS
RDB$ARGUMENT_POSITION	SMALLINT		Position of the argument described in the RDB$FUNCTION_NAME column in relation to the other arguments
RDB$MECHANISM	SMALLINT		Specifies whether the argument is passed by value (value of 0) or by reference (value of 1)
RDB$FIELD_TYPE	SMALLINT		Datatype of the argument being defined Valid values are: • BLOB - 261 • CHAR - 14 • CSTRING - 40 • D_FLOAT - 11 • DOUBLE - 27 • FLOAT - 10 • INT64 - 16 • INTEGER - 8 • QUAD - 9 • SMALLINT - 7 • DATE - 12 (dialect 3 DATE) • TIME - 13 • TIMESTAMP - 35 (DATE in older versions) • VARCHAR - 37

TABLE 9–14 RDB$FUNCTION_ARGUMENTS (*continued*)

Column name	Datatype	Length	Description
RDB$FIELD_SCALE	SMALLINT		Scale factor for an argument that has an integer datatype; the scale factor is the power of 10 by which the integer is multiplied
RDB$FIELD_LENGTH	SMALLINT		The length of the argument defined in this row Valid column lengths are: • BLOB - 8 • D_FLOAT - 8 • DATE - 4 • DOUBLE - 8 • FLOAT - 4 • INT64 - 8 • INTEGER - 4 • QUAD - 8 • SMALLINT - 2 • TIME - 4 • TIMESTAMP - 8
RDB$FIELD_SUB_TYPE	SMALLINT		If RDB$FIELD_TYPE is 7 (SMALLINT), 8 (INTEGER), or 16 (INT64), subtype can be: • 0 or NULL - RDB$FIELD_TYPE • 1 - NUMERIC • 2 - DECIMAL
RDB$CHARACTER_SET_ID	SMALLINT		Unique numeric identifier for a character set
RDB$FIELD_PRECISION	SMALLINT		The declared precision of the DECIMAL or NUMERIC function argument

RDB$FUNCTIONS

RDB$FUNCTIONS defines a user-defined function.

TABLE 9–15 RDB$FUNCTIONS

Column name	Datatype	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name for a function
RDB$FUNCTION_TYPE	SMALLINT		Reserved for future use
RDB$QUERY_NAME	CHAR	31	Alternate name for the function that can be used in ISQL
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the function being defined
RDB$MODULE_NAME	VARCHAR	253	Names the function library where the executable function is stored
RDB$ENTRYPOINT	CHAR	31	Entry point within the function library for the function being defined
RDB$RETURN_ARGUMENT	SMALLINT		Position of the argument returned to the calling program; this position is specified in relation to other arguments
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the function is: • User-defined (value of 0) • System-defined (value of 1)

RDB$GENERATORS

RDB$GENERATORS stores information about generators, which provide the ability to generate a unique identifier for a table.

TABLE 9–16 RDB$GENERATORS

Column name	Datatype	Length	Description
RDB$GENERATOR_NAME	CHAR	31	Name of the generator used in calls to gen_id to query or alter its value; actually an offset in the generator pages, not really an identifier. Not stable over backup/restore.
RDB$GENERATOR_ID	SMALLINT		Unique system-assigned ID number for the generator
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the generator is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$INDEX_SEGMENTS

RDB$INDEX_SEGMENTS specifies the columns that comprise an index for a table. Modifying these rows corrupts rather than changes an index unless the RDB$INDICES row is deleted and re-created in the same transaction.

TABLE 9–17 RDB$INDEX_SEGMENTS

Column name	Datatype	Length	Description
RDB$INDEX_NAME	CHAR	31	The index associated with this index segment; if the value of this column changes, the RDB$INDEX_NAME column in RDB$INDICES must also be changed
RDB$FIELD_NAME	CHAR	31	The index segment being defined; the value of this column must match the value of the RDB$FIELD_NAME column in RDB$RELATION_FIELDS
RDB$FIELD_POSITION	SMALLINT		Position of the index segment being defined; corresponds to the sort order of the index

RDB$INDICES

RDB$INDICES defines the index structures that allow Firebird to locate rows in the database more quickly. Because Firebird provides both simple indexes (a single-key column) and multi-segment indexes (multiple-key columns), each index defined in this table must have corresponding occurrences in the RDB$INDEX_SEGMENTS table.

TABLE 9–18 RDB$INDICES

Column name	Datatype	Length	Description
RDB$INDEX_NAME	CHAR	31	Names the index being defined; if the value of this column changes, change its value in the RDB$INDEX_SEGMENTS table. Implicit indexes are named according to type: • RDB$PRIMARY*nnn*: automatically generated by primary key • RDB$nnn: unique • RDB$FOREIGN*nnn*: generated by foreign key • RDB$INDEX_*nnn*: system index indexes created via CREATE INDEX have the names given by the user
RDB$RELATION_NAME	CHAR	31	Names the table associated with this index; the table must be defined in the RDB$RELATIONS table
RDB$INDEX_ID	SMALLINT		Contains an internal identifier for the index being defined; do *not* write to this column

TABLE 9–18 RDB$INDICES (*continued*)

Column name	Datatype	Length	Description
RDB$UNIQUE_FLAG	SMALLINT		Specifies whether the index allows duplicate values Values: • 0 - allows duplicate values • 1 - does not allow duplicate values Eliminate duplicates before creating a unique index
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the index
RDB$SEGMENT_COUNT	SMALLINT		Number of segments in the index; a value of 1 indicates a simple index
RDB$INDEX_INACTIVE	SMALLINT		Indicates whether the index is: • Active (value of 0) • Inactive (value of 1)
RDB$INDEX_TYPE	SMALLINT		Indicates whether the index is: • Ascending (value of 0 or NULL) • Descending (value of 1)
RDB$FOREIGN_KEY	CHAR	31	Name of FOREIGN KEY constraint for which the index is implemented
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the index is: • User-defined (value of 0) • System-defined (value greater than 0)

TABLE 9–18 RDB$INDICES (*continued*)

Column name	Datatype	Length	Description
RDB$EXPRESSION_BLR	BLOB	80	Subtype BLR: Contains the BLR (Binary Language Representation) for the expression, evaluated by the database at execution time; used for PC semantics
RDB$EXPRESSION_SOURCE	BLOB	80	Subtype Text: Contains original text source for the column; used for PC semantics
RDB$STATISTICS	DOUBLE PRECISION		Selectivity factor for the index; the optimizer uses index selectivity, a measure of uniqueness for indexed columns, to choose an access strategy for a query

RDB$LOG_FILES

RDB$LOG_FILES is no longer used.

RDB$PAGES

RDB$PAGES keeps track of each page allocated to the database.

WARNING ***Not for user update***; modifying RDB$PAGES in any way corrupts a database.

TABLE 9–19 RDB$PAGES

Column name	Datatype	Length	Description
RDB$PAGE_NUMBER	INTEGER		The physically allocated page number
RDB$RELATION_ID	SMALLINT		Identifier number of the table for which this page is allocated
RDB$PAGE_SEQUENCE	INTEGER		The sequence number of this page in the table to other pages allocated for the previously identified table
RDB$PAGE_TYPE	SMALLINT		Type of page; *this information is for system use only*: 0 - Undefined 1 - Database header page 2 - Page inventory page 3 - Transaction inventory page 4 - Pointer page 5 - Data page 6 - Index root page 7 - Index (B-tree) page 8 - Blob data page 9 - Gen-ids 10 - Write-ahead log information

RDB$PROCEDURE_PARAMETERS

RDB$PROCEDURE_PARAMETERS stores information about each parameter for each of a database's procedures.

TABLE 9–20 RDB$PROCEDURE_PARAMETERS

Column name	Datatype	Length	Description
RDB$PARAMETER_NAME	CHAR	31	Parameter name
RDB$PROCEDURE_NAME	CHAR	31	Name of the procedure in which the parameter is used
RDB$PARAMETER_NUMBER	SMALLINT		Parameter sequence number
RDB$PARAMETER_TYPE	SMALLINT		Parameter datatype Values are: • 0 = input • 1 = output
RDB$FIELD_SOURCE	CHAR	31	Global column name; this name should have a corresponding entry in RDB$FIELDS; the matching column is RDB$FIELD_NAME
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the parameter
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the parameter is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$PROCEDURES

RDB$PROCEDURES stores information about a database's stored procedures.

TABLE 10 RDB$PROCEDURES

Column name	Datatype	Length	Description
RDB$PROCEDURE_NAME	CHAR	31	Procedure name
RDB$PROCEDURE_ID	SMALLINT		Procedure number
RDB$PROCEDURE_INPUTS	SMALLINT		Number of input parameters
PROCEDURE_OUTPUTS	SMALLINT		Number of output parameters
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the procedure
RDB$PROCEDURE_SOURCE	BLOB	80	Subtype Text: Source code for the procedure
RDB$PROCEDURE_BLR	BLOB	80	Subtype BLR: BLR (Binary Language Representation) of the procedure source
RDB$SECURITY_CLASS	CHAR	31	Security class of the procedure
RDB$OWNER_NAME	CHAR	31	User who created the procedure (the owner for SQL security purposes)
RDB$RUNTIME	BLOB	80	Subtype Summary: Describes procedure metadata; used for performance enhancement
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the procedure is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$REF_CONSTRAINTS

RDB$REF_CONSTRAINTS stores referential integrity constraint information.

TABLE 9–21 RDB$REF_CONSTRAINTS

Column name	Datatype	Length	Description
RDB$CONSTRAINT_NAME	CHAR	31	Name of a referential constraint; should have a corresponding entry in RDB$RELATION_CONSTRAINTS; the matching column is RDB$CONSTRAINT_NAME
RDB$CONST_NAME_UQ	CHAR	31	Name of a referenced PRIMARY KEY or UNIQUE constraint; should have a corresponding entry in RDB$RELATION_CONSTRAINTS; the matching column is RDB$CONSTRAINT_NAME
RDB$MATCH_OPTION	CHAR	7	Reserved for later use; currently defaults to FULL
RDB$UPDATE_RULE	CHAR	11	Specifies the type of action on the foreign key when the primary key is updated; values are RESTRICT, NO ACTION, CASCADE, SET NULL, or SET DEFAULT
RDB$DELETE_RULE	CHAR	11	Specifies the type of action on the foreign key when the primary key is DELETED; values are RESTRICT, NO ACTION, CASCADE, SET NULL, or SET DEFAULT

RDB$RELATION_CONSTRAINTS

RDB$RELATION_CONSTRAINTS stores information about integrity constraints for tables.

TABLE 9–22 RDB$RELATION_CONSTRAINTS

Column name	Datatype	Length	Description
RDB$CONSTRAINT_NAME	CHAR	31	Name of a table constraint
RDB$CONSTRAINT_TYPE	CHAR	11	Type of table constraint ;constraint types are: PRIMARY KEY UNIQUE FOREIGN KEY CHECK NOT NULL
RDB$RELATION_NAME	CHAR	31	Name of the table for which the constraint is defined
RDB$DEFERRABLE	CHAR	3	Reserved for later use; currently defaults to NO
RDB$INITIALLY_DEFERRED	CHAR	3	Reserved for later use; currently defaults to NO
RDB$INDEX_NAME	CHAR	31	Name of the index used by UNIQUE, PRIMARY KEY, or FOREIGN KEY constraints

RDB$RELATION_FIELDS

For database tables, RDB$RELATION_FIELDS lists columns and describes column characteristics for domains. SQL columns are defined in RDB$RELATION_FIELDS. The column name in RDB$FIELD_SOURCE points to RDB$FIELDS.RDB$FIELD_NAME that contains a system name ("SQL$<*n*>"). This entry includes information such as column type and length. For both domains and simple columns, this table may contain default and nullability information.

TABLE 9–23 RDB$RELATION_FIELDS

Column name	Datatype	Length	Description
RDB$FIELD_NAME	CHAR	31	Name of the column whose characteristics being defined; the combination of the values in this column and in the RDB$RELATION_NAME column in this table must be unique
RDB$RELATION_NAME	CHAR	31	Table to which a particular column belongs; a table with this name must appear in RDB$RELATIONS The combination of the values in this column and in the RDB$FIELD_NAMES column in this table must be unique
RDB$FIELD_SOURCE	CHAR	31	The name for this column in the RDB$FIELDS table; if the column is based on a domain, contains the domain name
RDB$QUERY_NAME	CHAR	31	Alternate column name for use in ISQL; supersedes the value in RDB$FIELDS
RDB$BASE_FIELD	CHAR	31	Views only: The name of the column from RDB$FIELDS in a table or view that is the base for a view column being defined; for the base column: • RDB$BASE_FIELD provides the column name • RDB$VIEW_CONTEXT, a column in this table, provides the source table name

TABLE 9–23 RDB$RELATION_FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$EDIT_STRING	VARCHAR	125	Not used in SQL
RDB$FIELD_POSITION	SMALLINT		The position of the column in relation to other columns: • ISQL obtains the ordinal position for displaying column values when printing rows from this column • **gpre** uses the column order for SELECT and INSERT statements If two or more columns in the same table have the same value for this column, those columns appear in random order
RDB$QUERY_HEADER	BLOB	80	Not used in SQL
RDB$UPDATE_FLAG	SMALLINT		Not used by Firebird; included for compatibility with other DSRI-based systems
RDB$FIELD_ID	SMALLINT		Identifier for use in BLR (Binary Language Representation) to name the column • Because this identifier changes during database backup and restore, try to use it in transient requests only • Do *not* modify this column
RDB$VIEW_CONTEXT	SMALLINT		Alias used to qualify view columns by specifying the table location of the base column; it must have the same value as the alias used in the view BLR (Binary Language Representation) for this context stream
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the column being defined

TABLE 9–23 RDB$RELATION_FIELDS (*continued*)

Column name	Datatype	Length	Description
RDB$DEFAULT_VALUE	BLOB	80	Subtype BLR: BLR (Binary Language Representation) for default clause
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the column is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$SECURITY_CLASS	CHAR	31	Names a security class defined in the RDB$SECURITY_CLASSES table; the access restrictions defined by this security class apply to all users of this column
RDB$COMPLEX_NAME	CHAR	31	Reserved for future use
RDB$NULL_FLAG	SMALLINT		Indicates whether the column may contain NULLs: • Allows NULLs (value of 0 or NULL) • Disallows NULLs (value of 1)
RDB$DEFAULT_SOURCE	BLOB	80	Subtype Text: SQL source to define defaults
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence; if not NULL, should have a corresponding entry in RDB$COLLATIONS; the matching column has the same name

RDB$RELATIONS

RDB$RELATIONS defines some of the characteristics of tables and views. Other characteristics, such as the columns included in the table and a description of each column, are stored in the RDB$RELATION_FIELDS table.

TABLE 9–24 RDB$RELATIONS

Column name	Datatype	Length	Description
RDB$VIEW_BLR	BLOB	80	Subtype BLR: For a view, contains the BLR (Binary Language Representation) of the query Firebird evaluates at the time of execution
RDB$VIEW_SOURCE	BLOB	80	Subtype Text: For a view, contains the original source query for the view definition
RDB$_DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the table being defined
RDB$RELATION_ID	SMALLINT		Contains the internal identification number used in BLR (Binary Language Representation) requests; do *not* modify this column
RDB$SYSTEM_FLAG	SMALLINT		Indicates the contents of a table, either: • User-data (value of 0) • System information (value greater than 0) Do *not* set this column to 1 when creating tables

TABLE 9–24 RDB$RELATIONS (*continued*)

Column name	Datatype	Length	Description
RDB$DBKEY_LENGTH	SMALLINT		Length of the database key Values are: • For tables: 8 • For views: 8 times the number of tables referenced in the view definition Do *not* modify the value of this column
RDB$FORMAT	SMALLINT		For Firebird internal use only; do *not* modify; has a corresponding entry in RDB$FORMATS; the matching column has the same name
RDB$FIELD_ID	SMALLINT		The number of columns in the table; this column is maintained by Firebird: do *not* modify the value of this column
RDB$RELATION_NAME	CHAR	31	The unique name of the table defined by this row
RDB$SECURITY_CLASS	CHAR	31	Security class defined in the RDB$SECURITY_CLASSES table; access controls defined in the security class apply to all uses of this table
RDB$EXTERNAL_FILE	VARCHAR	253	• The file in which the external table is stored • If this is blank, the table does not correspond to an external file
RDB$RUNTIME	BLOB	80	Subtype Summary: Describes table metadata; used for performance enhancement

TABLE 9–24 RDB$RELATIONS (*continued*)

Column name	Datatype	Length	Description
RDB$EXTERNAL_DESCRIPTION	BLOB	80	Subtype EXTERNAL_FILE_DESCRIPTION; user-written description of the external file
RDB$OWNER_NAME	CHAR	31	Identifies the creator of the table or view; the creator is considered the owner for SQL security (GRANT/REVOKE) purposes
RDB$DEFAULT_CLASS	CHAR	31	Default security class that Firebird applies to columns newly added to a table using the SQL security system
RDB$FLAGS	SMALLINT		1= SQL-defined trigger 2 = ignore permission checking (User-defined triggers require that a user executing them have underlying access permission to the objects accessed by the trigger. However, internal, system-defined triggers occasionally need to bypass those permission checks to enforce database integrity.)

RDB$ROLES

RDB$roles lists roles that have been defined in the database and the owner of each role.

TABLE 9–25 RDB$ROLES

Column name	Datatype	Length	Description
RDB$ROLE_NAME	CHAR	31	Name of role being defined
RDB$OWNER_NAME	CHAR	31	Name of Firebird user who is creating the role

RDB$SECURITY_CLASSES

RDB$SECURITY_CLASSES defines access control lists and associates them with databases, tables, views, and columns in tables and views. For all SQL objects, the information in this table is duplicated in the RDB$USER_PRIVILEGES system table.

TABLE 9–26 RDB$SECURITY_CLASSES

Column name	Datatype	Length	Description
RDB$SECURITY_CLASS	CHAR	31	Security class being defined; if the value of this column changes, change its name in the RDB$SECURITY_CLASS column in RDB$DATABASE, RDB$RELATIONS, and RDB$RELATION_FIELDS
RDB$ACL	BLOB	80	Subtype ACL: Access control list that specifies users and the privileges granted to those users
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the security class being defined

RDB$TRANSACTIONS

RDB$TRANSACTIONS keeps track of all multi-database transactions.

TABLE 9–27 RDB$TRANSACTIONS

Column name	Datatype	Length	Description
RDB$TRANSACTION_ID	INTEGER		Identifies the multi-database transaction being described
RDB$TRANSACTION_STATE	SMALLINT		Indicates the state of the transaction Valid values are: • 0 - limbo • 1 - committed • 2 - rolled back
RDB$TIMESTAMP	DATE		Reserved for future use
RDB$TRANSACTION_DESCRIPTION	BLOB	80	Subtype TRANSACTION_DESCRIPTION; describes a prepared multi-database transaction, available if the reconnect fails

RDB$TRIGGER_MESSAGES

RDB$TRIGGER_MESSAGES defines a trigger message and associates the message with a particular trigger.

TABLE 9–28 RDB$TRIGGER_MESSAGES

Column name	Datatype	Length	Description
RDB$TRIGGER_NAME	CHAR	31	Names the trigger associated with this trigger message; the trigger name must exist in RDB$TRIGGERS
RDB$MESSAGE_NUMBER	SMALLINT		The message number of the trigger message being defined; the maximum number of messages per trigger is 32,767
RDB$MESSAGE	VARCHAR	78	The source for the trigger message

RDB$TRIGGERS

RDB$TRIGGERS defines triggers.

TABLE 9–29 RDB$TRIGGERS

Column name	Datatype	Length	Description
RDB$TRIGGER_NAME	CHAR	31	Names the trigger being defined
RDB$RELATION_NAME	CHAR	31	Name of the table associated with the trigger being defined; this name must exist in RDB$RELATIONS
RDB$TRIGGER_SEQUENCE	SMALLINT		Sequence number for the trigger being defined; determines when a trigger is executed in relation to others of the same type • Triggers with the same sequence number execute in alphabetic order by trigger name • If this number is not assigned by the user, Firebird assigns a value of 0
RDB$TRIGGER_TYPE	SMALLINT		The type of trigger being defined Values are: • 1 - BEFORE INSERT • 2 - AFTER INSERT • 3 - BEFORE UPDATE • 4 - AFTER UPDATE • 5 - BEFORE DELETE • 6 - AFTER DELETE

TABLE 9–29 RDB$TRIGGERS (*continued*)

Column name	Datatype	Length	Description
RDB$TRIGGER_SOURCE	BLOB	80	Subtype Text: Original source of the trigger definition; the ISQL SHOW TRIGGERS statement displays information from this column
RDB$TRIGGER_BLR	BLOB	80	Subtype BLR: BLR (Binary Language Representation) of the trigger source
RDB$DESCRIPTION	BLOB	80	Subtype Text: User-written description of the trigger being defined; when including a comment in a CREATE TRIGGER or ALTER TRIGGER statement, ISQL writes to this column
RDB$TRIGGER_INACTIVE	SMALLINT		Indicates whether the trigger being defined is: • Active (value of 0) • Inactive (value of 1)
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the trigger is: • User-defined (value of 0 or NULL) • System-defined (value greater than 0)
RDB$FLAGS	SMALLINT		

RDB$TYPES

RDB$TYPES records enumerated datatypes and alias names for Firebird character sets and collation orders. This capability is not available in the current release.

TABLE 9–30 RDB$TYPES

Column name	Datatype	Length	Description
RDB$FIELD_NAME	CHAR	31	Column for which the enumerated datatype is being defined
RDB$TYPE	SMALLINT		Identifies the internal number that represents the column specified above; type codes (same as RDB$DEPENDENT_TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure All other values are reserved for future use
RDB$TYPE_NAME	CHAR	31	Text that corresponds to the internal number
RDB$DESCRIPTION	BLOB	80	Subtype Text: Contains a user-written description of the enumerated datatype being defined
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the datatype is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$USER_PRIVILEGES

RDB$USER_PRIVILEGES keeps track of the privileges assigned to a user through a SQL GRANT statement. There is one occurrence of this table for each user/privilege intersection.

TABLE 9–31 RDB$USER_PRIVILEGES

Column name	Datatype	Length	Description
RDB$USER	CHAR	31	Names the user who was granted the privilege listed in the RDB$PRIVILEGE column
RDB$GRANTOR	CHAR	31	Names the user who granted the privilege
RDB$PRIVILEGE	CHAR	6	Identifies the privilege granted to the user listed in the RDB$USER column, above Valid values are: • A (ALL) • S (SELECT) • D (DELETE) • I (INSERT) • U (UPDATE) • X (EXECUTE) • R (REFERENCE) • M (MEMBER OF, for roles)
RDB$GRANT_OPTION	SMALLINT		For normal privileges, indicates whether the privilege was granted with the WITH GRANT OPTION (value of 1) or not (value of 0) For member of a role privilege, indicates whether the role was granted with the WITH ADMIN OPTION (value of 2) or not (value of 0)

TABLE 9–31 RDB$USER_PRIVILEGES

Column name	Datatype	Length	Description
RDB$RELATION_NAME	CHAR	31	Identifies the table to which the privilege applies
RDB$FIELD_NAME	CHAR	31	For update privileges, identifies the column to which the privilege applies
RDB$USER_TYPE	SMALLINT		Indicates the user type: 8 = user, including SYSDBA 13 = role
RDB$OBJECT_TYPE	SMALLINT		Type of object for which the user or role has been granted permissions

RDB$VIEW_RELATIONS

RDB$VIEW_RELATIONS is the conjunct table between views and tables, used to avoid deleting a table that has dependent views. It is not used by SQL objects.

TABLE 9–32 RDB$VIEW_RELATIONS

Column name	Datatype	Length	Description
RDB$VIEW_NAME	CHAR	31	Name of a view: The combination of RDB$VIEW_NAME and RDB$VIEW_CONTEXT must be unique
RDB$RELATION_NAME	CHAR	31	Name of a table referenced in the view definition
RDB$VIEW_CONTEXT	SMALLINT		Alias used to qualify view columns; must have the same value as the alias used in the view BLR (Binary Language Representation) for this query
RDB$CONTEXT_NAME	CHAR	31	Textual version of the alias identified in RDB$VIEW_CONTEXT This variable must: • Match the value of the RDB$VIEW_SOURCE column for the corresponding table in RDB$RELATIONS • Be unique in the view

System views

You can create a SQL script using the code provided in this section to create four views that provide information about existing integrity constraints for a database. You must create the database prior to creating

these views. SQL system views are a subset of system views defined in the SQL-92 standard. Since they are defined by ANSI SQL-92, the names of the system views and their columns do not start with RDB$.

- The CHECK_CONSTRAINTS view

```
CREATE VIEW CHECK_CONSTRAINTS (
    CONSTRAINT_NAME,
    CHECK_CLAUSE
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$TRIGGER_SOURCE
FROM RDB$CHECK_CONSTRAINTS RC, RDB$TRIGGERS RT
     WHERE RT.RDB$TRIGGER_NAME = RC.RDB$TRIGGER_NAME;
```

- The CONSTRAINTS_COLUMN_USAGE view

```
CREATE VIEW CONSTRAINTS_COLUMN_USAGE (
    TABLE_NAME,
    COLUMN_NAME,
    CONSTRAINT_NAME
) AS
    SELECT RDB$RELATION_NAME, RDB$FIELD_NAME, RDB$CONSTRAINT_NAME
    FROM RDB$RELATION_CONSTRAINTS RC, RDB$INDEX_SEGMENTS RI
     WHERE RI.RDB$INDEX_NAME = RC.RDB$INDEX_NAME;
```

- The REFERENTIAL_CONSTRAINTS view

```
CREATE VIEW REFERENTIAL_CONSTRAINTS (
    CONSTRAINT_NAME,
    UNIQUE_CONSTRAINT_NAME,
    MATCH_OPTION,
    UPDATE_RULE,
    DELETE_RULE
) AS
    SELECT RDB$CONSTRAINT_NAME, RDB$CONST_NAME_UQ, RDB$MATCH_OPTION,
RDB$UPDATE_RULE, RDB$DELETE_RULE
     FROM RDB$REF_CONSTRAINTS;
```

- The TABLE_CONSTRAINTS view

```
CREATE VIEW TABLE_CONSTRAINTS (
    CONSTRAINT_NAME,
    TABLE_NAME,
    CONSTRAINT_TYPE,
    IS_DEFERRABLE,
    INITIALLY_DEFERRED
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$RELATION_NAME,
RDB$CONSTRAINT_TYPE, RDB$DEFERRABLE, RDB$INITIALLY_DEFERRED
     FROM RDB$RELATION_CONSTRAINTS;
```

CHECK_CONSTRAINTS

CHECK_CONSTRAINTS identifies all CHECK constraints defined in the database.

TABLE 9–33 CHECK_CONSTRAINTS

Column name	Datatype	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the CHECK constraint; nullable
CHECK_CLAUSE	BLOB	80	Subtype Text: Nullable; original source of the trigger definition, stored in the RDB$TRIGGER_SOURCE COLUMN in RDB$TRIGGERS

CONSTRAINTS_COLUMN_USAGE

CONSTRAINTS_COLUMN_USAGE identifies columns used by PRIMARY KEY and UNIQUE constraints. For FOREIGN KEY constraints, this view identifies the columns defining the constraint.

TABLE 9–34 CONSTRAINTS_COLUMN_USAGE

Column name	Datatype	Length	Description
TABLE_NAME	CHAR	31	Table for which the constraint is defined; nullable
COLUMN_NAME	CHAR	31	Column used in the constraint definition; nullable
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint; nullable

REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS identifies all referential constraints defined in a database.

TABLE 9–35 REFERENTIAL_CONSTRAINTS

Column name	Datatype	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint; nullable
UNIQUE_CONSTRAINT_NAME	CHAR	31	Name of the UNIQUE or PRIMARY KEY constraint corresponding to the specified referenced column list; nullable
MATCH_OPTION	CHAR	7	Reserved for future use; always set to FULL; nullable
UPDATE_RULE	CHAR	11	Reserved for future use; always set to RESTRICT; nullable
DELETE_RULE	CHAR	11	Reserved for future use; always set to RESTRICT; nullable

TABLE_CONSTRAINTS

TABLE_CONSTRAINTS identifies all constraints defined in a database.

TABLE 9–36 TABLE_CONSTRAINTS

Column name	Datatype	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint; nullable
TABLE_NAME	CHAR	31	Table for which the constraint is defined; nullable
CONSTRAINT_TYPE	CHAR	11	Possible values are UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK; nullable
IS_DEFERRABLE	CHAR	3	Reserved for future use; always set to No; nullable
INITIALLY_DEFERRED	CHAR	3	Reserved for future use; always set to No; nullable

CHAPTER 10

Resources and References

Recommended reading

InterBase® 6 Documentation set
especially the API Guide (APIGuide.pdf) and the Embedded SQL Guide (EmbedSQL.pdf). Available in printed form in the media kit, from the Borland Shop at http://www.borland.com. Beta version can be downloaded from ftp://ftpc.inprise.com/pub/interbase/techpubs/ib_b60_doc.zip

Data Modeling Essentials: Analysis, Design and Innovation (2nd Edition)
by Graeme Simsion, revised and updated by Graham Witt and Graeme Simsion
THE book for learning from the ground up about data analysis, relationships, normalization and creative ways to solve tough problems just got better! Read this book before you read any other - everything else is just a follow-up. The new (2001) edition adds new sections on UML and an object-oriented approach, capturing patterns and a whole new chapter on modeling for data warehousing. A walk-through example for presenting a large data model has been added to the Appendix.

SQL For Smarties: Advanced SQL Programming
by Joe Celko
Classic book of magic spells for SQL programming.

Data and Databases: Concepts in Practice
by Joe Celko
Not exactly a primer, but this book contains a good deal of material to cure you of the "spreadsheet mentality" when designing and mending databases.

SQL Puzzles and Answers
by Joe Celko
Hands-on approach to beating your SQL task into shape.

The Essence of SQL
by David Rozenshtein and Tom Bondur
Very concise, very much a beginner book.

The Practical SQL Handbook
by Judith S. Bowman et al.
How-to and desktop reference for standard SQL, well-reviewed.

A guide to the SQL Standard
by Hugh Darwen and Chris Date
All the things you wanted to know—and much that you didn't know you didn't know—about SQL-92, by the RDBMS "gods".

Understanding the New SQL: a Complete Guide
by Jim Melton and Alan Simon
Covers SQL-89 and SQL-92, comprehensive, good lookup reference for the beginner. Includes some basic modeling theory.

Mastering SQL
by Martin Gruber
Updated and expanded new version of the author's 'Understanding SQL' which makes standard SQL reachable, even for the beginner, and helps solid skills to develop quickly.

Reference web sites

The IBPhoenix site: http://www.ibphoenix.com
information and news center for users developing with Firebird or any version of InterBase®. It has an on-line Knowledgebase, dozens of authoritative articles, contacts for commercial support and consulting, links to tools and projects.

The Firebird site: http://firebird.sourceforge.net
information and news on the development side of the Firebird product. All Firebird binaries can be downloaded through this site. You can link through to the Firebird project page at Sourceforge from there; or go directly to http://sourceforge.net/projects/firebird.

Claudio Valderrama's '**Unofficial InterBase site**': http://www/cvalde.com
home of the Interbase Webring—links to many other sites where people are doing interesting things with Firebird and tools development. Contains an eclectic collection of articles, mended software, news links, etc.

Ivan Prenosil's site: http://www.volny.cz/iprenosil/interbase/
insightful articles and tools written and maintained by a guru in Firebird and InterBase.

Firebird forums

ib-support: join at http://www.yahoogroups.com/community/ib-support
This is the **only** support forum unless you have a specific Java-related problem. It's also the first place to raise a suspected bug for preliminary discussion.

firebird-java: join at http://www.yahoogroups.com/community/firebird-java
This is mainly a developers' forum for the JDBC/JCA drivers but there is some InterClient knowledge there. It's also the place to raise suspected bug issues for the Java drivers.
Use **ib-support** for database-related questions (SQL, configuration, network issues, etc.).

ib-conversions: join at http://www.yahoogroups.com/community/ib-conversions
This is a help list for people wanting to convert from another DBMS to Firebird. Only conversion questions should be raised here: take support questions to **ib-support**.

ib-priorities: join at http://www.yahoogroups.com/community/ib-priorities
Forum for raising and discussing new features that you would like in future versions of Firebird. No support questions are permitted here.

ib-architect: join at http://www.yahoogroups.com/community/ib-architect
Forum for raising and discussing architectural issues to do with the design and development of Firebird and its interfaces. Again, no support question are permitted. Discussions are usually quite rarefied.

firebird-tools: join at http://www.yahoogroups.com/community/firebird-tools
Forum for people developing tools and plug-ins for Firebird.

ibdi: join at http://www.yahoogroups.com/community/ibdi
General discussion forum. No support questions allowed but just about anything else "goes", as long as it is related to Firebird or InterBase in some way. Topics range far and wide, from logos to fluffy toys to deployment discussions.

Several lists run from Sourceforge, for developers actually working on the Firebird product. Support questions are NOT WELCOME on these lists:

firebird-devel: join at https://lists.sourceforge.net/lists/listinfo/firebird-devel
This list is for workers, not lurkers. Bug discussions are welcome here, provided you have already visited the Bug Tracker at the project site and established that it is not already known.

firebird-docs: join at https://lists.sourceforge.net/lists/listinfo/firebird-docs
This is also a non-lurkers' list, a forum for people working on documentation, or who want to start.

irebird-website: join at https://lists.sourceforge.net/lists/listinfo/firebird-website
Another non-lurkers' list, a forum for people working on the Firebird website, or who want to start.

News group interface

All of these lists are mirrored to a news server at news://news.atkin.com. Anyone can read the list traffic through this interface, but you must be a subscribed member of the list community in order to post messages through your news reader.

Firebird tools

Data access components

for Delphi™ and Borland C++Builder™

- **IB Objects**—http://www.ibobjects.com—with a user support list at http://www.yahoogroups.com/community/ibobjects
- **FIBPlus**—http://www.devrace.com
- **InterBaseXpress (IBX)**—obtainable from http://community.borland.com. You need to join CodeCentral to access the software. Instructions on site. Newsgroup at news://forums.inprise.com, join borland.public.interbase.interbasexpress.

DB management tools

IB_SQL: download from http://www.ibobjects.com/ibo_IB_SQL.html
Small footprint, free Win32 client tool for quick access management of Firebird and any InterBase version from 4.x up, any server platform. Datapump, scripting tool, SQL trace monitor, statistics, export, query forms, user management. Create custom desktops for different projects. Create databases, View/update/insert data, run scripts, perform DSQL, browse and download metadata. Built with IBObjects v. 4.

IBAccess: download from http://sourceforge.net/projects/ibaccess
Open source, free Linux or Win32 application to manage Firebird or Interbase servers (classic and superserver) and databases. Built with IBX. Intuitive interface to create tables, triggers, constraints, procedures. More information at http://www.ibaccess.org.

IBQuery: download from http://www.torry.net/apps/utilities/database/mitecibquery.zip
Ultra light, elegant, free Win32 Firebird/InterBase 6.x management tool. SQL Editor with results in text format for easy copy and paste, object explorer, data explorer, user management, export data to XLS, DBF, CSV, raw. Database creation and validation, statistics, server log view.

DB management tools (cont.)

IBConsole: obtainable from http://community.borland.com. You need to join CodeCentral to access the software. Instructions on site.

Developed as part of InterBase 6 before open sourcing, now open source and free under IPL, under ongoing development, ships with commercial InterBase 6.x products. Almost compatible with Firebird—some system queries won't work or are unreliable, but tool is mostly serviceable. Service tools (backup, validation, etc.) work with Superserver, unavailable for Classic.

IBAdmin: commercial—information at http://www.sqlly.com/IBAdmin.htm

Includes stored procedure debugger, statistics and WISQL-like functionality. Current commercial versions come in Lite, Standard and Professional versions. Visual database designer, grant manager, SQL debugger, code editor with code insight and code completion, Database Comparer tool, dependencies explorer and more. Version 1.1 is free.

InterBase Workbench: commercial—trial version at http://www.interbaseworkbench.com

Full, GUI desktop environment for development and system administration. User participation in the development of this tool is encouraged. Built with IBObjects.

QuickDesk: commercial—trial version at http://ems-hitech.com/quickdesk/download.phtml

Commercial, complete Firebird/InterBase developer's desktop. Many features including message localization, stored procedure debugger, conversion and export utilities, grant manager, etc. Built with FIBPlus.

IBExpert: commercial—trial version at http://www.h-k.de

Another complete commercial developer's desktop for Firebird/InterBase. Features include capability to work with two databases at once, BLOB viewer, multi-language (German, English, user-defined), custom keyboard, UDF editor...etc. Powered by IB Objects.

Backup

IBBackup: http://www.entwicklerforum.org/downloads/ibbackup_setup.exe

Free GUI wrapper for GBAK and more utilities, avoiding the annoyances and failures of IBConsole and the confusion of command-line GBAK. Option to archive your gbk as a zipfile. Full graphical help file and several other helpful utilities for the SYSDBA or developer.

ODBC drivers

Firebird ODBC/JDBC Driver: download from http://www.ibphoenix.com/ibp_60_odbc.html

Free, open source JDBC-compliant ODBC driver for Firebird and InterBase 6.x, sponsored by IBPhoenix and community donations. Source code released under the Initial Developer's Public License, now based in the Firebird CVS tree at Sourceforge (for developers interested in working on the code).

XTG ODBC Driver: download from http://www.xtgsystems.com/download/ib6odbc.zip

Free, open source Firebird/InterBase 6.x ODBC driver, conforming to the ODBC 3 API CORE level. Supported on Win32 (windows 9x/NT; tested in Windows 95/98, windows NT4 SP3 and win2000). Version 1.0.0 beta 15) may contain some bugs but is usable. Full installation kit with binary. Project source at http://ofbodbc.sourceforge.net/

Gemini ODBC Driver: commercial—downloadevaluation from http://www.ibdatabase.com/ibgem_21_eval.zip.

The ODBC driver for Firebird/Interbase. ODBC interfaces are based on the Call Level Interface (CLI) specification designed by SQL Access Group and then adopted by X/Open and ISO/IEC as an appendix to current standard of SQL language. The driver conforms to the ODBC specification described in the ODBC Programmer's Reference, version 3.51. Available for Win32 and Linux platforms. More information at http://www.geocities.com/ibdatabase/

Easysoft ODBC Driver: commercial—trial version from http://www.easysoft.com/support/trials.phtml?product=2201

For Windows and Linux Intel. Support for UNICODE on some platforms. Information at http://www.easysoft.com/products/interbase/

More tools

Dozens more tools—both free and commercial—can be accessed from the Contributed Downloads page at http://www.ibphoenix.com. Link to http://www.ibphoenix.com/ibp_contrib_download.html

Several useful utilities and components can be downloaded from the IB Objects community code page at http://www.ibobjects.com/ibocontributed.html