Firebird 2.5 Language Reference

Paul Vinkenoog
Dmitry Yemanov
Thomas Woinke

Firebird 2.5 Language Reference

by Paul Vinkenoog, Dmitry Yemanov, and Thomas Woinke

Table of Contents

List of Tables

Chapter 1. Introduction

Subject matter

What's this book about?

To be updated.

Authorship

To be updated.

Chapter 2. Background

SQL flavors

This reference describes the SQL language supported by Firebird. However, there are different subsets of SQL that apply to different areas and thus should be distinguished. Namely, they are:

- Dynamic SQL (DSQL)
- Procedural SQL (PSQL)
- Embedded SQL (ESQL)
- Interactive SQL (ISQL)

Dynamic SQL is the major part of the language which corresponds to the Part 2 (SQL/Foundation) part of the SQL specification. DSQL represents statements passed by client applications through the public Firebird API and processed by the database engine.

Procedural SQL is the extention to the Dynamic SQL which additionally allows compound statements containing local variables, assignments, conditions, loops and other procedural constructs. PSQL corresponds to the Part 4 (SQL/PSM) part of the SQL specifications. Originally, PSQL extenstions were available in persistent stored modules (procedures and triggers) only, but recently they became surfaced in Dynamic SQL as well (see EXECUTE BLOCK).

Embedded SQL defines the DSQL subset supported by Firebird GPRE - the application which allows you to embed SQL constructs into your host programming language (C, C++, Pascal, Cobol, etc) and preprocess those embedded constructs into the proper Firebird API calls. Please note that only a part of DSQL statements and expressions are supported in ESQL.

Interactive ISQL means the language that can be executed using Firebird ISQL - the command-line application for interactive SQL access to databases. As it's a regular client application, its native language is DSQL. But it also offers a few additional commands

Both DSQL and PSQL subsets are completely presented in this reference. Neither ESQL nor ISQL flavors are described here unless mentioned explicitly.

SQL dialects

The SQL dialect is a term that defines the specific features of the SQL language that are available when accessing the database. SQL dialects can be defined at the database level and at the connection level. There are three dialects available:

- Dialect 1 stores both date and time information in a DATE data type and has a TIMESTAMP data type which is identical to DATE. Double quotes are used to delimit string data. The precision for NUMERIC and DECIMAL data types is less than in dialect 3 and if the precision is greater than 9, Firebird internally stores these as long floating point values. BIGINT is not permitted as a data type. Identifiers are case-insensitive. Generator values are stored as 32-bit integers.
- Dialect 2 is available only on the Firebird client connection and cannot be set in the database. It is intended to assist debugging of possible problems with legacy data when migrating a database from dialect 1 to 3.
- Dialect 3 databases allow numbers (DECIMAL and NUMERIC data types) to be internally stored as long fixed point values (scaled integers) when the precision is greater than 9. The TIME data type is able to be used and stores time data only. The DATE data type stores on date information. BIGINT is available as a 64-bit integer data type. Double quotes can be used but only for identifiers

that are case-sensitive, not for string data which has to use single quotes. Generator values are stored as 64-bit integers.

Dialect 1 is targeted to provide support for legacy (pre-v6) InterBase applications so that they would work the same way with Firebird. Dialect 2 is used as a transition dialect intended to highlight the issues while migrating to dialect 3. Newly developed databases and applications are highly recommended to use dialect 3. Both database and connection dialects should match, except the migration case with Dialect 2.

This reference describes the semantics of SQL dialect 3 unless specified otherwise.

Error conditions

Processing of the every SQL statement either completes successfully or fails due to a specific error condition.

Chapter 3. Language structure

Basics: statements, tokens, keywords

To be written.

Identifiers

To be written.

Literals

To be written.

Operators and specials

To be written.

Comments

To be written.

Chapter 4. Data types

Numeric types

SMALLINT

SMALLINT is the 16-bit signed integer type.

Syntax:

```
SMALLINT
```

SMALLINT numbers range from -2^{15} to $2^{15}-1$, or from -32768 to 32767.

INTEGER

INTEGER is the 32-bit signed integer type.

Syntax:

```
INTEGER
```

A shorthand form INT is also available.

INTEGER numbers range from -2^{31} to $2^{31}-1$, or from -2147483648 to 2147483647.

BIGINT data type

BIGINT is the 64-bit signed integer type.

Syntax:

```
BIGINT
```

BIGINT numbers range from -2^{63} to $2^{63}-1$, or from -9223372036854775808 to 9223372036854775807.

Note

The BIGINT data type is available in Dialect 3 only.

FLOAT data type

FLOAT (precision) is the approximate number with the binary precision equal to or greater than specified by the <precision>.

Syntax:

```
FLOAT [(precision)]
```

The precision must be positive, but there's no explicit upper limit (however, it's physically limited by 32767). If the precision declaration is omitted, then zero value is implied.

If the precision is less than 8 then the FLOAT data type is internally implemented as a single precision floating-point value (commonly it is 32-bit). The FLOAT data type with a greater declared precision

is implemented as a double precision floating-point value (commonly it is 64-bit). Both formats are covered by the IEEE 754 standard.

The declared precision is not strictly enforced, it only dictates the choice between the two supported binary precisions.

REAL data type

REAL is the approximate number with the binary precision less than 8.

Syntax:

```
REAL
```

REAL is a predefined FLOAT data type implemented as a single precision floating-point value, so please refer to the appropriate section for the details.

DOUBLE PRECISION data type

DOUBLE PRECISION is the approximate number with the binary precision equal to or greater than 8.

Syntax:

```
DOUBLE PRECISION
```

DOUBLE PRECISION is a predefined FLOAT data type implemented as a double precision floating-point value, so please refer to the appropriate section for the details.

LONG FLOAT data type

LONG FLOAT is the approximate number with the binary precision equal to or greater than 8.

Syntax:

```
LONG FLOAT [(precision)]
```

LONG FLOAT is non-standard alternative to the DOUBLE PRECISION data type, so please refer to the appropriate section for the details. The optional precision declaration is ignored.

NUMERIC data type

NUMERIC (precision, scale) is the exact number with the decimal precision and scale specified by the <precision> and <scale>.

Syntax:

```
NUMERIC [precision [, scale]]
```

The scale of NUMERIC is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of NUMERIC is the total count of decimal digits in the number.

The precision must be positive, the maximum supported value is 18. The scale must be zero or positive, up to the specified precision.

If the scale is omitted, then zero value is implied, thus meaning an integer value of the specified precision, i.e. NUMERIC (P) is equivalent to NUMERIC (P, 0). If both the precision and the scale are omitted, then precision of 9 and zero scale are implied, i.e. NUMERIC is equivalent to NUMERIC (9, 0).

The internal representation of the NUMERIC data type may vary. Numerics with the precision up to (and including) 4 are always stored as scaled short integers (SMALLINT). Numerics with the precision up to (and including) 9 are always stored as scaled regular integers (INTEGER). Storage of higher precision numerics depends on the SQL dialect. In Dialect 3, they are stored as scaled large integers (BIGINT). In Dialect 1, however, large integers are not available, therefore they are stored as double precision floating-point values (DOUBLE PRECISION).

The effective precision limit for the given value depends on the corresponding storage. For example, NUMERIC (5) will be stored as INTEGER, thus allowing values in the precision range up to (and including) NUMERIC (9). So beware that the declared precision is not strictly enforced.

Values outside the range limited by the effective precision are not allowed. Values with the scale larger than the declared one will be rounded to the declared scale while performing an assignment.

DECIMAL data type

DECIMAL (precision, scale) is the exact number with the decimal scale specified by the <scale> and the decimal precision equal to or greater than the value of the specified <precision>.

Syntax:

```
DECIMAL [precision [, scale]]
```

A shorthand form DEC is also available.

DECIMAL data type is almost identical to the NUMERIC data type, so please refer to the appropriate section for the details. The only difference is that DECIMAL is never internally stored as a small integer, so its minimal effective precision is equal to 9.

Note

Accordingly to the SQL standard, NUMERIC is declared to be a strict data type which enforces the declared precision, while DECIMAL can accept more decimal digits than declared. In Firebird, however, both enforce the effective precision rather than the declared one, with NUMERIC being just a bit stricter in the lower precision range.

Character types

CHARACTER data type

CHARACTER (length) is the fixed-length string with the character length equal to the specified <length>.

Syntax:

```
CHARACTER [(length)] [CHARACTER SET charset]
```

A shorthand form CHAR is also available.

The declared length must be positive. If the length specification is omitted, length of 1 (one character) is implied. The maximum supported byte length is 32767, thus the maximum supported character length depends on the declared character set.

The CHARACTER data type definition may include the character set name, so that all the characters of the string value belong to the declared character set. If the character set name is omitted, the database-wise default character set is implied. See the complete list of supported character sets in Appendix B.

Values of the CHARACTER data type are right padded with the appropriate padding character up to the declared length, and are physically stored this way. However, the trailing padding characters are semantically insignificant and ignored when comparing CHARACTER values. Strings longer than the declared length are not accepted, unless the excess characters are all equal to the appropriate padding character, in this case the string is truncated to the declared length.

Note

The default padding character is the space character (ASCII code 0x20) and it applies to the every character set except OCTETS which represents binary strings and thus has binary zero (ASCII code 0x00) as the padding character.

NATIONAL CHARACTER data type

NATIONAL CHARACTER (length) is the fixed-length string with the character length equal to the specified <length> and having the predefined character set of ISO8859_1.

Syntax:

```
NATIONAL CHARACTER [(length)]
```

Shorthand forms NATIONAL CHAR and NCHAR are also available.

Besides having the character set hardcoded, NATIONAL CHARACTER data type is absolutely equal to the CHARACTER data type, so please refer to the appropriate section for the details.

CHARACTER VARYING data type

CHARACTER VARYING (length) is the varying-length string with the character length limited by the specified <length>.

Syntax:

```
CHARACTER VARYING (length) [CHARACTER SET charset]
```

Shorthand forms CHAR VARYING and VARCHAR are also available.

The declared length is mandatory and it must be positive. The maximum supported byte length is 32765, thus the maximum supported character length depends on the declared character set.

The CHARACTER data type definition may include the character set name, so that all the characters of the string value belong to the declared character set. If the character set name is omitted, the database-wise default character set is implied. See the complete list of supported character sets in Appendix B.

Values of the CHARACTER VARYING data type are never padded. Internally, the CHARACTER VARYING values are prefixed with the 16-bit length counter, so they are stored using the actual string length. Strings longer than the declared length are not accepted.

NATIONAL CHARACTER VARYING data type

NATIONAL CHARACTER VARYING (length) is the varying-length string with the character length limited by the specified <length> and having the predefined character set of ISO8859_1.

Syntax:

```
NATIONAL CHARACTER VARYING [(length)]
```

A shorthand form NATIONAL CHAR VARYING is also available.

Besides having the character set hardcoded, NATIONAL CHARACTER VARYING data type is absolutely equal to the CHARACTER VARYING data type, so please refer to the appropriate section for the details.

Date/time types

DATE data type

DATE represents the temporal value containing the year-month-day part.

Syntax:

```
DATE
```

DATE values range from 01-Jan-0001 to 31-Dec-9999.

Internally, DATE values are stored as 32-bit signed integers, representing the offset (in days) from the baseline date (17-Nov-1858).

Note

This definition applies to Dialect 3 only. In Dialect 1, the DATE data type is a replacement for the TIMESTAMP data type, thus containing both date and time parts. See the appropriate section for the details.

TIME data type

TIME represents the temporal value containing the hour-minute-second part, including second fractions.

Syntax:

```
TIME
```

TIME values range from 00:00:00.0000 AM to 23:59:59.9999 PM.

Internally, TIME values are stored as 32-bit scaled unsigned integers, representing both the number of seconds passed since the day start (which is a baseline, so that zero time value means a midnight) and the number of second fractions passed since the start of the current second. The scale value is 10000, so the supported precision is 1/10000 second (100 microseconds).

Note

The TIME data type is available in Dialect 3 only.

TIMESTAMP data type

TIMESTAMP represents the temporal value containing both year-month-day and hour-minute-second parts, including second fractions.

Syntax:

```
TIMESTAMP
```

TIMESTAMP values range from 01-Jan-0001 00:00:00.0000 AM to 31-Dec-9999 23:59:59.9999 PM.

Internally, TIMESTAMP values are stored as a combination of two 32-bit integers, each representing its own part (date or time), so its total size is 64 bits. See prior sections for the storage details.

Binary types

BLOB data type

BLOB is the data type representing text or binary data of arbitrary length.

Syntax:

```
BLOB [SUB_TYPE subtype]
   [SEGMENT SIZE segsize]
   [CHARACTER SET charset]
```

Shortcuts:

```
BLOB (segsize)
BLOB (segsize, subtype)
BLOB (, subtype)
```

By default, every blob internally consists of a chained list of segments, each up to 64KB in size. Small segments are combined together so that they could share the same disk page. There are three storage levels of blobs. Blobs of level 0 fit on a single disk page and may even reside at the same data page where the rest of the fields are stored. Blobs of level 1 are always stored on dedicated disk pages and consist of the root page (which itself does not store any blob data but contains an inventory of other data pages) and a number of data pages the blob content is splitted among. Blobs of level 2 extend the aforementioned scheme by having a root inventory page which enumerates other inventory pages that in turn point to the data pages containing the blob contents. This three-level storage layout along with the database page size establish the practical size limit for blobs, which is equal to 2GB for the maximum supported page size of 16KB.

So it can be said that BLOB segments represent the logical blob structure rather than the physical one. They do not directly affect the blob storage on disk and the appropriate low-level I/O operations performed by Firebird internally, but they define the granulation of the blob contents and the way how the blob is accessed at the interface layer. If you need to read the whole blob, you should iterate through all the segments. If you need to read just a few bytes in the middle of the blob, you should read all the segments preceding these bytes.

Note

There is also a concept of so called streamed blobs. They are not splitted into separate segments and are directly accessible for reading and writing from any specified position. The choice between segmented and streamed blobs is done by the application developer when creating the blob contents, it's not covered by the BLOB declaration in SQL.

The BLOB sub-type describes the blob contents. While Firebird usually does not guess about the internal structure of the BLOB data, there are some predefined sub-types that may affect the BLOB handling. The BLOB sub-type can be declared via either the corresponding numeric value, or by using a mnemonic name for the predefined sub-types, e.g.:

```
BLOB SUB_TYPE 0
BLOB SUB_TEXT BINARY
```

If the BLOB sub-type is omitted, sub-type of 0 (BINARY) is implied. However, if the BLOB declararion includes the character set (ses below), then sub-type of 1 (TEXT) is implied.

BLOB sub-types can be negative, zero or positive values in the range from -32768 to 32767. Negative BLOB sub-types are user-defined ones and can be used by application developers to distinguish between different formats of binary contents. Zero BLOB sub-type is predefined for binary blobs that are processed transparently without any assumptions about their contents. BLOB sub-type of 1 is predefined for text blobs that are processed similarly to the character data types. They may be treated

like very long character strings, including conversions from and to character data types, usage in the various string-wise operators and built-in functions, and so on. Binary and text blobs can be used by application developers as well. Positive BLOB sub-types greater than 1 are reserved for internal use, they are not allowed for user blobs.

The BLOB segment size defines the number of bytes to be processed (read or written) by a single interface operation with that blob. It must be a positive value limited by 65535 bytes. If the BLOB segment size is omitted, then the value of 80 bytes is implied. The BLOB segment size also

The BLOB declaration may include the character set name, so that all the characters the blob contents consists of belong to the specified character set. The character set can be specified for text and binary blobs only. If this happens for a binary blob, then the sub-type gets implicitly changed to TEXT. If the character set name is omitted, the database-wise default character set is implied. See the complete list of supported character sets in Appendix B.

Arrays

To be written.

Comparison rules

Values of the same data type are compared accordingly to the natural rules of the corresponding data type. Blobs are compared either byte-wise or character-wise for binary and text blobs respectively, using their entire contents.

Mixed-type comparisons, however, are processed using the special priority rules. The value with the data type of the less priority is implicitly converted to the data type of the value with greater priority and then both values are compared naturally. The priority list is the following (starting with the least priority):

- CHARACTER, CHARACTER VARYING
- SMALLINT
- INTEGER
- BIGINT
- REAL
- DOUBLE PRECISION
- DATE, TIME
- TIMESTAMP

The derived data types (NUMERIC, DECIMAL, FLOAT, LONG FLOAT) are compared accordingly to the priorities of their underlying data types. Scaled numeric values are adjusted to their maximal scale before comparing.

Character strings are compared without regard to their trailing padding characters, if any. International character strings (any character set except NONE, OCTETS and ASCII) are compared accordingly to their collation rules, explicit or implicit.

When DATE or TIME value is converted to TIMESTAMP, it gets its missing part completed. DATE values get zero time part (midnight), TIME values get the today's date as their date part.

If one of the comparison arguments is a blob, then custom rules are used. In this case, the second argument gets converted to a string value (binary or character) and then the string comparison rules apply.

Coercion rules

There are situations where a few values of different data types must provide the unified result. In this case, those values should be converted to their common (less restrictive) data type. Below are the rules applied (in the specified order) for those implicit conversions:

- If any data type is BLOB, the result is also BLOB
- If any data type is CHARACTER VARYING, or if one data type is CHARACTER and another one is numeric or date/time, the result is CHARACTER VARYING
- If any data type is CHARACTER, the result is also CHARACTER
- If any datatype is an approximate numeric, then each data type must be numeric and the result is an approximate numeric
- If all data types are exact numerics, the result is an exact numeric with the maximal precision and maximal scale
- If any data type is DATE, TIME or TIMESTAMP, then each data type must be the same DATE, TIME or TIMESTAMP and the result is also the same DATE, TIME or TIMESTAMP

If the coercion result is BLOB and there is any binary blob in the list, the resulting blob is also binary.

If the coercion results in a text blob, the resulting charset is derived using the following rules:

- If any input character set is OCTETS, the resulting character set is also OCTETS
- If any input character set is neither NONE nor ASCII, it is chosen as the resulting character set
- If any input character set is ASCII, or if any input part is numeric or date/time, result is ASCII
- Otherwise, result is NONE

Collation rules

To be written.

Chapter 5. Common language elements

Value expressions

To be written.

Select expressions

To be written.

Predicates

To be written.

Chapter 6. DML statements

DELETE

Available in. DSQL, ESQL, PSQL

Description. DELETE removes rows from a database table or from one or more tables underlying a view. WHERE and ROWS clauses can limit the number of rows deleted. If neither WHERE nor ROWS is present, DELETE removes all the rows in the relation.

Syntax.

```
DELETE
   [TRANSACTION name]
   FROM {tablename | viewname} [[AS] alias]
   [WHERE {search-conditions | CURRENT OF cursorname}]
   [PLAN plan_items]
   [ORDER BY sort_items]
   [ROWS <m> [TO <n>]]
   [RETURNING <values> [INTO <variables>]]

<m>, <n>     ::=  Any expression evaluating to an integer.
<values>     ::=  value_expression [, value_expression ...]
<variables>  ::=  :varname [, :varname ...]
```

Restrictions

- The TRANSACTION directive is only available in ESQL.
- In a pure DSQL session, WHERE CURRENT OF isn't of much use, since there exists no DSQL statement to create a cursor.
- The PLAN, ORDER BY and ROWS clauses are not available in ESQL.
- The RETURNING clause is not available in ESQL.
- The "INTO <*variables*>" subclause is only available in PSQL.
- When returning values into the context variable NEW, this name must not be preceded by a colon (write NEW, not :NEW).

Aliases

Description. An alias can represent a relation throughout the statement. Attention: the alias obscures the formal relation name. Once declared, you must use either the alias or nothing to qualify field names, as the examples show.

Examples.

Supported usage:

```
delete from Cities where name starting 'Alex'

delete from Cities where Cities.name starting 'Alex'

delete from Cities C where name starting 'Alex'

delete from Cities C where C.name starting 'Alex'
```

Not possible:

```
delete from Cities C where Cities.name starting 'Alex'
```

TRANSACTION

Available in. ESQL

Description. The optional TRANSACTION clause specifies under which active transaction the statement should be executed.

Example.

```
delete transaction tr_cleanup from Buses where date_endlife is not null
```

WHERE

Description. A WHERE clause limits the deletion to the rows matching the search condition, or – in ESQL and PSQL only – to the current row of a named cursor.

Examples.

```
delete from People where firstname <> 'Boris' and lastname <> 'Johnson'

delete from Cities where current of Cur_Cities;  -- ESQL and PSQL only
```

A delete using WHERE CURRENT OF is called a *positioned delete*, because it deletes the record at the current position. A delete using "WHERE <*condition*>" is called a *searched delete*, because the engine has to search for the record(s) meeting the condition.

PLAN

Available in. DSQL, PSQL

Description. A PLAN clause allows the user to optimize the operation manually.

Example.

```
delete from Submissions
  where date_entered < '1-Jan-2002'
  plan (Submissions index ix_subm_date)
```

ORDER BY

Available in. DSQL, PSQL

Description. The ORDER BY clause orders the set before the actual deletion takes place. It only makes sense in combination with ROWS, but is also valid without it.

Examples.

```
delete from Purchases order by date rows 1           -- deletes oldest pu

delete from Sales order by custno desc rows 1 to 10  -- deletes from high

delete from Sales order by custno desc  -- deletes all sales, ORDER BY cl
```

ROWS

Available in. DSQL, PSQL

Description. Limits the amount of rows deleted to a specified number or range.

Syntax.

```
ROWS <m> [TO <n>]

<m>, <n>  ::=  Any expression evaluating to an integer.
```

With a single argument *m*, the deletion is limited to the first *m* rows of the dataset defined by the table or view and the optional WHERE and ORDER BY clauses.

Points to note:

- If *m* > the total number of rows in the dataset, the entire set is deleted.
- If *m* = 0, no rows are deleted.
- If *m* < 0, an error is raised.

With two arguments *m* and *n*, the deletion is limited to rows *m* to *n* inclusively. Row numbers are 1-based.

Points to note when using two arguments:

- If *m* > the total number of rows in the dataset, no rows are deleted.
- If *m* lies within the set but *n* doesn't, the rows from *m* to the end of the set are deleted.
- If *m* < 1 or *n* < 1, an error is raised.
- If *n* = *m*-1, no rows are deleted.
- If *n* < *m*-1, an error is raised.

Examples.

```
delete from popgroups order by name desc rows 1  -- will probably delete Z

delete from popgroups order by formed rows 5     -- deletes 5 oldest group

delete from popgroups rows 5 to 12      -- no ordering, may delete any 8
```

RETURNING

Available in. DSQL, PSQL

Description. A DELETE statement removing *at most one row* may optionally include a RETURNING clause in order to return values from the deleted row. The clause, if present, need not contain all the relation's columns and may also contain other columns or expressions. When returning into the NEW context variable within a trigger, the preceding colon must be omitted.

Examples.

```
delete from Scholars
  where firstname = 'Henry' and lastname = 'Higgins'
  returning lastname, fullname, id

delete from Dumbbells
  order by iq desc
  rows 1
  returning lastname, iq into :lname, :iq;

delete from TempSales ts
  where ts.id = tempid
  returning ts.qty into new.qty;  -- not ":new.qty"
```

Notes.

- In DSQL, a statement with a RETURNING clause *always* returns exactly one row. If no record was actually deleted, the fields in this row are all `NULL`. This behaviour may change in a later version of Firebird.

- In PSQL, if no row was deleted, nothing is returned, and the target variables keep their existing values.

EXECUTE BLOCK

Available in. DSQL

Description. Executes a block of PSQL code as if it were a stored procedure, optionally with input and output parameters and variable declarations. This allows the user to perform "on-the-fly" PSQL within a DSQL context.

Syntax.

```
EXECUTE BLOCK [(<inparams>)]
     [RETURNS (<outparams>)]
AS
   [<declarations>]
BEGIN
   [<PSQL statements>]
END

<inparams>          ::=  <param_decl> = ? [, <inparams> ]
<outparams>         ::=  <param_decl>     [, <outparams>]
<param_decl>        ::=  paramname <type> [NOT NULL] [COLLATE collation]
<type>              ::=  sql_datatype | [TYPE OF] domain | TYPE OF COLUMN
<declarations>      ::=  See PSQL::DECLARE for the exact syntax
<PSQL statements>   ::=  See the PSQL chapter
```

Examples.

This example injects the numbers 0 through 127 and their corresponding ASCII characters into the table ASCIITABLE:

```
execute block
as
declare i int = 0;
begin
  while (i < 128) do
  begin
    insert into AsciiTable values (:i, ascii_char(:i));
    i = i + 1;
  end
end
```

The next example calculates the geometric mean of two numbers and returns it to the user:

```
execute block (x double precision = ?, y double precision = ?)
returns (gmean double precision)
as
begin
  gmean = sqrt(x*y);
  suspend;
end
```

Because this block has input parameters, it has to be prepared first. Then the parameters can be set and the block executed. It depends on the client software how this must be done and even if it is possible at all – see the notes below.

Our last example takes two integer values, `smallest` and `largest`. For all the numbers in the range `smallest` .. `largest`, the block outputs the number itself, its square, its cube and its fourth power.

```
execute block (smallest int = ?, largest int = ?)
returns (number int, square bigint, cube bigint, fourth bigint)
as
begin
  number = smallest;
  while (number <= largest) do
  begin
    square = number * number;
    cube   = number * square;
    fourth = number * cube;
    suspend;
    number = number + 1;
  end
end
```

Again, it depends on the client software if and how you can set the parameter values.

Input and output parameters

Executing a block without input parameters should be possible with every Firebird client that allows the user to enter his or her own DSQL statements. If there are input parameters, things get trickier: these parameters must get their values after the statement is prepared but before it is executed. This requires special provisions, which not every client application offers. (Firebird's own isql, for one, doesn't.)

The server only accepts question marks ("`?`") as placeholders for the input values, not "`:a`", "`:MyParam`" etc., or literal values. Client software may support the "`:xxx`" form though, and will preprocess it before sending it to the server.

If the block has output parameters, you *must* use SUSPEND or nothing will be returned.

Output is always returned in the form of a result set, just as with a SELECT statement. You can't use RETURNING_VALUES or execute the block INTO some variables, even if there is only one result row.

For more information about parameter and variable declarations, [TYPE OF] *`domain`*, TYPE OF COLUMN etc., consult the chapter on Procedural SQL, in particular *PSQL::DECLARE*.

Statement terminators

Some clients, especially those allowing the user to submit several statements at once, may require you to surround the EXECUTE BLOCK statement with SET TERM lines, like this:

```
set term #;
execute block (...)
as
begin
  statement1;
  statement2;
end
#
set term ;#
```

As an example, in Firebird's isql client you must set the terminator to something other than "`;`" before you type in the EXECUTE BLOCK statement. If you don't, isql will try to execute the part you have typed so far as soon as you hit Enter after a line with a semicolon.

EXECUTE PROCEDURE

Available in. DSQL, ESQL, PSQL

Description. Executes a stored procedure (SP), optionally taking input parameters and/or returning output values.

Syntax.

```
EXECUTE PROCEDURE
   [TRANSACTION transaction]
   procname
   [<in_item> [, <in_item> ...]]
   [RETURNING_VALUES <out_item> [, <out_item> ...]]

<in_item>   ::=  <inparam> [<nullind>]
<out_item>  ::=  <outvar>  [<nullind>]
<inparam>   ::=  an expression evaluating to the declared parameter type
<outvar>    ::=  a host language or PSQL variable to receive the return va
<nullind>   ::=  [INDICATOR]:host_lang_intvar
```

Notes

- In ESQL, input parameters must be literals or host language variables. For output parameters, host variables must be specified in the RETURNING_VALUES clause. `NULL` indicators must be host language variables of type integer, with less than zero indicating `NULL` and zero or greater indicating not `NULL` (which means that a proper value is present in the corresponding parameter).
- In PSQL, input parameters may be any expression that resolves to the expected type. For output parameters, local variables must be specified in the RETURNING_VALUES clause.
- In DSQL, input parameters may be any expression that resolves to the expected type. The handling of output parameters depends on the client software.
- In PSQL and DSQL, `NULL` indicators are neither valid nor necessary. `NULL`s are passed via the input/output parameters themselves.
- TRANSACTION clauses are not supported in PSQL.
- In ESQL, variable names used as parameters or outvars must be preceded by a colon (":"). In PSQL the colon is generally optional, but forbidden for the trigger context variables OLD and NEW.

Examples.

In PSQL (with optional colons):

```
execute procedure MakeFullName
  :FirstName, :MiddleName, :LastName
  returning_values :FullName;
```

The same call in ESQL (with obligatory colons):

```
exec sql
```

```
  execute procedure MakeFullName
    :FirstName, :MiddleName, :LastName
    returning_values :FullName;
```

...and in Firebird's command-line utility isql (with literal parameters):

```
execute procedure MakeFullName
  'J', 'Edgar', 'Hoover';
```

Note: In isql, don't use RETURNING_VALUES. Any output values are shown automatically.

Finally, a PSQL example with expression parameters:

```
execute procedure MakeFullName
  'Mr./Mrs. ' || FirstName, MiddleName, upper(LastName)
  returning_values FullName;
```

INSERT

Available in. DSQL, ESQL, PSQL

Description. Adds rows to a database table or to one or more tables underlying a view. If the field values are given in a VALUES clause, exactly one row is inserted. The values may also be provided by a SELECT statement, in which case zero to many rows may be inserted. With the DEFAULT VALUES clause, no values are provided at all and exactly one row is inserted.

Syntax.

```
INSERT [TRANSACTION name]
   INTO {tablename | viewname}
   {DEFAULT VALUES | [(<column_list>)] <value_source>}
   [RETURNING <value_list> [INTO <variables>]]

<column_list>   ::=  colname [, colname ...]
<value_source>  ::=  VALUES (<value_list>) | <select_stmt>
<value_list>    ::=  value_expression [, value_expression ...]
<variables>     ::=  :varname [, :varname ...]
<select_stmt>   ::=  a SELECT or UNION whose result set fits the target c
```

Restrictions

- The TRANSACTION directive is only available in ESQL.
- The RETURNING clause is not available in ESQL.
- The "INTO <*variables*>" subclause is only available in PSQL.
- When returning values into the context variable NEW, this name must not be preceded by a colon (":").
- No column may appear more than once in the column list.

INSERT ... VALUES

The VALUES list must provide a value for every column in the column list, in the same order and of the correct type. If the column list is absent, values must be provided for every column in the table or view (computed columns excluded).

String literals may optionally be preceded by a character set name, using *introducer syntax*, in order to let the engine know how to interpret the input.

Examples.

```
insert into cars (make, model, year)
  values ('Ford', 'T', 1908)

insert into cars
  values ('Ford', 'T', 1908, 'USA', 850)

/* assuming that the columns of table Cars are: make, model, year, country

insert into People
  values (_ISO8859_1 'Hans-Jörg Schäfer')  -- notice the '_' prefix
```

INSERT ... SELECT

Here, the output columns of the SELECT statement must provide a value for every target column in the column list, in the same order and of the correct type. If the column list is absent, values must be provided for every column in the table or view (computed columns excluded).

Examples.

```
insert into cars (make, model, year)
  select (make, model, year) from new_cars

insert into cars
  select * from new_cars

/* assuming that table New_cars has the exact same columns as table Cars

insert into Members (number, name)
  select number, name from NewMembers where Accepted = 1
    union
  select number, name from SuspendedMembers where Vindicated = 1
```

Of course, the column names in the source table need not be the same as those in the target table. Any type of SELECT statement is permitted, as long as its output columns exactly match the insert columns in number, order and type. Types need not be exactly the same, but they must be assignment-compatible.

INSERT ... DEFAULT VALUES

The DEFAULT VALUES clause allows insertion of a record without providing any values at all, neither directly nor from a SELECT statement. This is only possible if every NOT NULL or CHECKed column in the table either has a valid default declared or gets such a value from a BEFORE INSERT trigger. Furthermore, triggers providing required field values must not depend on the presence of input values.

Example.

```
insert into journal default values
  returning entry_id
```

The RETURNING clause

An INSERT statement adding *at most one row* may optionally include a RETURNING clause in order to return values from the inserted row. The clause, if present, need not contain all of the insert columns and may also contain other columns or expressions. The returned values reflect any changes that may have been made in BEFORE triggers, but not those in AFTER triggers.

Examples.

```
insert into Scholars (firstname, lastname, address, phone, email)
  values ('Henry', 'Higgins', '27A Wimpole Street', '3231212', null)
  returning lastname, fullname, id

insert into Dumbbells (firstname, lastname, iq)
  select fname, lname, iq from Friends order by iq rows 1
  returning id, firstname, iq into :id, :fname, :iq;
```

Notes.

- RETURNING is only supported for VALUES inserts and singleton SELECT inserts.
- In DSQL, a statement with a RETURNING clause *always* returns exactly one row. If no record was actually inserted, the fields in this row are all `NULL`. This behaviour may change in a later version of Firebird. In PSQL, if no row was inserted, nothing is returned, and the target variables keep their existing values.
- The RETURNING clause is not available in ESQL.

Inserting into BLOB columns

Inserting into BLOB columns is only possible under the following circumstances:

1. The client application has made special provisions for such inserts, using the Firebird API. In this case, the modus operandi is application-specific and outside the scope of this manual.
2. The value inserted is a text string of at most 32767 bytes. Please notice: if the value is not a string literal, beware of concatenations, as these may exceed the maximum length.
3. You are using the “INSERT ... SELECT” form and one or more columns in the result set are BLOBs.
4. You use the INSERT CURSOR statement (ESQL only).

INSERT CURSOR

Available in. ESQL

Description. In Embedded SQL only, you can insert data, in segment-sized chunks, into a BLOB through a special BLOB cursor. As you can only write one segment at a time, this is usually done in a (host language) loop.

Syntax.

```
INSERT CURSOR blobcursor VALUES (:buf [INDICATOR] :size)

blobcursor  ::=  an opened BLOB insert cursor
buf         ::=  host variable containing the data segment
size        ::=  data size in bytes; must be less than or equal to the se
```

Example.

```
exec sql
  insert cursor cur_jpeg values (:imgbuf indicator :seglen);
```

Please notice that, unlike INSERT, INSERT CURSOR does not write anything to the table itself. The segments are written to a BLOB structure which, at that time, “floats” freely in the database, unconnected to any other object.

After inserting all the segments in this manner, you must close the BLOB cursor and perform a regular INSERT statement to insert the BLOB ID (and possibly other fields) into the table.

Updating BLOB fields is done in the same way, except that you finalize the entire operation by issuing a regular UPDATE statement instead of an INSERT statement.

A complete discussion of ESQL statements is outside the scope of this document. For more information about ESQL, please consult the *InterBase 6 Embedded SQL Guide* (google for `60EmbedSQL.zip`).

MERGE

Available in. DSQL, PSQL

Description. Merges data into a table or updatable view. The source may a table, view or "anything you can SELECT from" in general. Each source record will be used to update one or more target records, insert a new record in the target table, or neither. The action taken depends on the provided condition and the WHEN clause(s). The condition will typically contain a comparison of fields in the source and target relations.

Syntax.

```
MERGE INTO target [[AS] target-alias]
   USING source [[AS] source-alias]
   ON condition
   WHEN MATCHED THEN UPDATE SET colname = value [, colname = value ...]
   WHEN NOT MATCHED THEN INSERT [(<columns>)] VALUES (<values>)

target     ::=  a table or updatable view
source     ::=  a table, GTT, view, selectable SP, derived table or CTE
<columns>  ::=  colname [, colname ...]
<values>   ::=  value   [, value   ...]

Note: It is allowed to provide only one of the WHEN clauses
```

Examples.

```
merge into books b
  using purchases p
  on p.title = b.title and p.type = 'bk'
  when matched then
    update set b.desc = b.desc || '; ' || p.desc
  when not matched then
    insert (title, desc, bought) values (p.title, p.desc, p.bought)

merge into customers c
  using (select * from customers_delta where id > 10) cd
  on (c.id = cd.id)
  when matched then update set name = cd.name
  when not matched then insert (id, name) values (cd.id, cd.name)
```

Note

WHEN NOT MATCHED should be seen from the point of view of the *source* (the relation in the USING clause). That is: if a source record doesn't have a match in the target table, the INSERT clause is executed. Conversely, records in the target table without a matching source record don't cause anything to happen.

Warning

If the WHEN MATCHED clause is present and multiple source records match the same record(s) in the target table, the UPDATE clause is executed for all the matching source

records, each update overwriting the previous one. This is non-standard behaviour: SQL-2003 specifies that an exception must be raised in such cases.

SELECT

Available in. DSQL, ESQL, PSQL

The SELECT statement retrieves data from the database and hands them to the application or the enclosing SQL statement. Data are returned in zero or more *rows*, each containing one or more *columns* or *fields*. The total of rows returned is the *result set* of the statement.

Global syntax.

```
SELECT
   [TRANSACTION name]
   [FIRST <m>] [SKIP <n>]
   [DISTINCT | ALL] <columns>
   [INTO <host-varlist>]
   FROM source [[AS] alias]
   [<joins>]
   [WHERE <condition>]
   [GROUP BY <grouping-list>
     [HAVING <aggregate-condition>]]
   [PLAN <plan-expr>]
   [UNION [DISTINCT | ALL] <other-select>]
   [ORDER BY <ordering-list>]
   [ROWS m [TO n]]
   [FOR UPDATE [OF <columns>]]
   [WITH LOCK]
   [INTO <PSQL-varlist>]
```

The only mandatory parts of the SELECT statement are:

- The SELECT keyword, followed by a columns list. This part specifies *what* you want to retrieve.
- The FROM keyword, followed by a selectable object. This tells the engine *where* you want to get it *from*.

In its most basic form, SELECT retrieves a number of columns from a single table or view, like this:

```
select id, name, address
  from contacts
```

Or, to retrieve all the columns:

```
select * from sales
```

In practice, the rows retrieved are often limited by a WHERE clause. The result set may be sorted by an ORDER BY clause, and FIRST, SKIP or ROWS may further limit the number of output rows. The column list may contain all kinds of expressions instead of just column names, and the source need not be a table or view: it may also be a derived table, a common table expression (CTE) or a selectable stored procedure (SP). Multiple sources may be combined in a JOIN, and multiple result sets may be combined in a UNION.

The following sections discuss the available SELECT subclauses and their usage in detail.

The TRANSACTION directive

Available in. ESQL

This ESQL-only directive tells the engine to execute the statement under the specified transaction, which must have been previously declared and opened, e.g.:

```
select transaction tr_getsales
  partno, desc, price, amount
  from v_sales
  where custno = 101
```

FIRST, SKIP and ROWS

Available in. DSQL, PSQL

Syntax.

```
SELECT
   [FIRST <m>] [SKIP <n>]
   FROM ...
   ...

<m>, <n>  ::=  integer-literal | query-parameter | (integer-expression)
```

or

```
SELECT
   ...
   FROM ...
   ...
   ROWS m [TO n]

m, n  ::=  integer-expression
```

Please notice: FIRST and SKIP are Firebird-specific, non-SQL-compliant keywords. You are adviced to use the ROWS syntax wherever possible.

FIRST limits the output of a query to the first so-many rows. SKIP will suppress the given number of rows before starting to return output.

FIRST and SKIP are both optional. When used together as in "FIRST *m* SKIP *n*", the *n* topmost rows of the output set are discarded and the first *m* rows of the remainder are returned.

SKIP 0 is allowed, but of course rather pointless. FIRST 0 is also allowed and returns an empty set. Negative SKIP and/or FIRST values result in an error.

If a SKIP lands past the end of the dataset, an empty set is returned. If the number of rows in the dataset (or the remainder after a SKIP) is less than the value given after FIRST, that smaller number of rows is returned. These are valid results, not error conditions.

Any argument to FIRST and SKIP that is not an integer literal or an SQL parameter must be enclosed in parentheses. This implies that a subselect must be enclosed in *two* pairs of parentheses.

Examples

The following query will return the first 10 names from the People table:

```
select first 10 id, name from People
  order by name asc
```

The following query will return everything *but* the first 10 names:

```
select skip 10 id, name from People
  order by name asc
```

And this one returns the last 10 rows. Notice the double parentheses:

```
select skip ((select count(*) - 10 from People))
  id, name from People
  order by name asc
```

This query returns rows 81–100 of the People table:

```
select first 20 skip 80 id, name from People
  order by name asc
```

As said, FIRST and SKIP are not standard SQL. In new code, it's better to use the standards-compliant ROWS keyword.

Contrary to FIRST and SKIP, ROWS accepts any kind of integer expression as argument without parentheses. (Of course, parentheses may be necessary *within* the expression, and a subselect still needs to be parenthesized.)

With a single argument *m*, ROWS returns the first *m* rows of the dataset.

Points to note:

- If *m* > the total number of rows in the dataset, the entire set is returned.
- If *m* = 0, an empty set is returned.
- If *m* < 0, an error is raised.

With two arguments *m* and *n*, rows *m* to *n* of the dataset are returned, inclusively. Row numbers are 1-based.

Points to note when using two arguments:

- If *m* > the total number of rows in the dataset, an empty set is returned.
- If *m* lies within the set but *n* doesn't, the rows from *m* to the end of the set are returned.
- If *m* < 1 or *n* < 1, an error is raised.
- If *n* = *m*-1, an empty set is returned.
- If *n* < *m*-1, an error is raised.

The SQL-compliant ROWS syntax obviates the need for FIRST and SKIP, except in one case: a SKIP without FIRST, which returns the entire remainder of the set after skipping a given number of rows. (Well, this is not entirely true. You can supply a second argument that you know is bigger than the number of rows in the set, or request COUNT(*) with a subselect. But SKIP is simpler and clearer here.)

You cannot use ROWS together with FIRST and/or SKIP in a single SELECT statement, but you can use one form in the top-level statement and the other in subselects, or use the two syntaxes in different subselects.

When used with a UNION, the ROWS subclause applies to the UNION as a whole and must be placed after the last SELECT. If you want to limit the output of one or more individual SELECTs within the UNION, you have two options: either use FIRST/SKIP on those SELECT statements (probably of limited use, as you can't use ORDER BY on individual selects within a union), or convert them to derived tables with ROWS clauses.

Below are the previous examples rewritten using ROWS. Notice that ROWS is placed at or near the end of the statement, whereas FIRST and SKIP come even before the columns list.

```
select id, name from People
  order by name asc
  rows 1 to 10

select id, name from People
  order by name asc
```

```
  rows 11 to (select count(*) from People)

select id, name from People
  order by name asc
  rows (select count(*) - 9 from People)
    to (select count(*) from People)

select id, name from People
  order by name asc
  rows 81 to 100
```

Both FIRST/SKIP and ROWS can be used without an ORDER BY clause, but this rarely makes sense, unless you want to just “get an idea” about a table without being interested in the actual data. In that case, a statement like “select * from UnknownTable rows 20” may give you a quick insight without risking lots of network traffic and thousands of data rows flying across your screen.

The column list

The column list contains one or more comma-separated value expressions. Each expression provides a value for one output column, except * (“star”), which stands for all the columns in a relation (i.e. a table, view or selectable stored procedure).

Syntax.

```
SELECT
   [...]
   [DISTINCT | ALL] <output-column> [, <output-column> ...]
   [...]
   FROM ...

<output-column>     ::=  [qualifier.]*
                           | <value-expression> [COLLATE collation] [[AS]

<value-expression>  ::=  [qualifier.]table-column
                           | [qualifier.]view-column
                           | [qualifier.]selectable-SP-outparm
                           | constant
                           | context-variable
                           | function-call
                           | single-value-subselect
                           | CASE-construct
                           | “any other expression returning a single
                              value of a Firebird data type or NULL”

qualifier           ::=  a relation name or alias
collation           ::=  a valid collation name (only for character type
```

It is always valid to qualify a column name (or “*”) with the name or alias of the table, view or selectable SP to which it belongs, followed by a dot. Qualifying becomes mandatory if the column name occurs in more than one relation taking part in a join. Qualifying “*” is mandatory if it isn't the only item in the column list.

Please notice that aliases obfuscate the original relation name: once a table, view or SP has been aliased, you can only use the alias as a qualifier; the relation name itself has become unavailable.

The column list may optionally be preceded by one of the keywords DISTINCT or ALL. DISTINCT filters out any duplicate rows. That is, if two or more rows have the same values in every corresponding column, only one of them is included in the result set. ALL shows all the rows including duplicates. ALL is the default and therefore rarely used; it is supported for reasons of SQL compliance.

A COLLATE clause will not change the appearance of the column as such. However, if the specified collation changes the case or accent sensitivity of the column, it may influence:

- The ordering, if an ORDER BY clause is also present and the column is involved in it.
- Grouping, if the column is part of a GROUP BY clause.
- The rows retrieved (and hence the total number of rows in the result set), if DISTINCT is used.

Examples of SELECT queries with different types of column lists

A simple SELECT using only column names:

```
select cust_id, cust_name, phone
  from customers
  where city = 'London'
```

A query featuring a concatenation expression and a function call in the columns list:

```
select 'Mr./Mrs. ' || lastname, street, zip, upper(city)
  from contacts
  where date_last_purchase(id) = current_date
```

A query with two subselects:

```
select p.fullname,
       (select name from classes c where c.id = p.class) as class,
       (select name from mentors m where m.id = p.mentor) as mentor
from pupils p
```

The following query accomplishes the same as the previous one using joins instead of subselects:

```
select p.fullname,
       c.name as class,
       m.name as mentor
from pupils p
  join classes c on c.id = p.class
  join mentors m on m.id = p.mentor
```

This query uses a CASE construct to determine the correct title, e.g. when sending mail to a person:

```
select case upper(sex)
         when 'F' then 'Mrs.'
         when 'M' then 'Mr.'
         else ''
       end as title,
       lastname,
       address
from employees
```

Querying a selectable stored procedure:

```
select * from interesting_transactions(2010, 3, 'S')
  order by amount
```

Selecting from columns of a derived table. A derived table is a parenthesized SELECT statement whose result set is used in an enclosing query as if it were a regular table or view. The derived table is shown in bold here:

```
select fieldcount,
       count(relation) as num_tables
from   (select r.rdb$relation_name as relation,
               count(*) as fieldcount
```

```
            from   rdb$relations r
                   join rdb$relation_fields rf
                     on rf.rdb$relation_name = r.rdb$relation_name
            group by relation)
    group by fieldcount
```

Asking the time through a context variable (CURRENT_TIME):

```
select current_time from rdb$database
```

For those not familiar with RDB$DATABASE: this is a system table that is present in all Firebird databases and is guaranteed to contain exactly one row. Although it wasn't created for this purpose, it has become standard practice among Firebird programmers to select from this table if you want to select "from nothing", i.e., if you need data that are not bound to a any table or view, but can be derived from the expressions in the output columns alone. Another example is:

```
select power(12, 2) as twelve_squared, power(12, 3) as twelve_cubed
  from rdb$database
```

Finally, an example where you select some meaningful information from RDB$DATABASE itself:

```
select rdb$character_set_name from rdb$database
```

As you may have guessed, this will give you the default character set of the database.

Selecting INTO variables

Available in. ESQL, PSQL

In PSQL code or embedded SQL, the results of a SELECT statement may be loaded – on a row-by-row basis – into local variables (PSQL) or host languages variables (ESQL). In fact, this is often the only way to do anything with the returned values at all. The number, order and types of the variables must match the columns in the output row.

A "plain" SELECT statement can only be used in ESQL or PSQL if it returns at most one row – in other words, if it is a *singleton* select. For multirow selects, PSQL provides the FOR SELECT loop, which is discussed in the PSQL chapter. In addition, both PSQL and ESQL support the DECLARE CURSOR statement, which binds a named cursor to a SELECT statement. The cursor can then be used to walk the result set.

Syntax. In embedded SQL, the INTO clause is placed between the column list and the FROM keyword:

```
SELECT
   [...]
   <column-list>
   [INTO <variable-list>]
   FROM ...
   [...]

<variable-list>  ::=  :hostvar [, :hostvar ...]
```

In PSQL, the INTO clause must appear at the very end of the statement:

```
SELECT
   [...]
   <column-list>
   FROM ...
   [...]
   [INTO <variable-list>]
```

```
<variable-list>  ::=  [:]psqlvar [, [:]psqlvar ...]
```

Notice that in PSQL, the colons before the variable names are optional.

Examples

In ESQL, with `min_amt`, `avg_amt` and `max_amt` host language (e.g. C) variables:

```
select min(amount), avg(cast(amount as float)), max(amount)
  into :min_amt, :avg_amt, :max_amt
  from orders
  where artno = 372218;
```

In PSQL, with `min_amt`, `avg_amt` and `max_amt` previously defined PSQL variables or output parameters:

```
select min(amount), avg(cast(amount as float)), max(amount)
  from orders
  where artno = 372218
  into min_amt, avg_amt, max_amt;
```

(The CAST serves to make the average a broken number. Otherwise – since `amount` is presumably an integer field – it would be truncated to the nearest lower integer.)

In a PSQL trigger:

```
select list(name, ', ')
  from persons p
  where p.id in (new.father, new.mother)
  into new.parentnames;
```

The FROM clause

The FROM clause specifies the source(s) from which the data are to be retrieved. In its simplest form, this is just a single table or view. But the source can also be a selectable stored procedure, a derived table or a common table expression. Multiple sources can be combined using various types of joins.

This section concentrates on single-source selects. Joins are discussed in the next section.

Syntax.

```
SELECT
   ...
   FROM <source>
   [<joins>]
   [...]

<source>          ::=  {table
                          | view
                          | selectable-stored-procedure [(args)]
                          | <derived-table>
                          | <common-table-expression>}
                       [[AS] alias]

<derived-table>   ::=  (select-statement) [[AS] alias]
                         [(<column-aliases>)]

<common-table-expression>
                  ::=  WITH [RECURSIVE] <cte-def> [, <cte-def> ...]
```

```
                        select-statement

<cte-def>          ::=  name [(<column-aliases>)] AS (select-statement)

<column-aliases>   ::=  column-alias [, column-alias ...]
```

Selecting from a table or view

When selecting from a single table or view, the FROM clause need not contain anything more than the name. An alias may be useful or even necessary if there are subselects that refer to the main select statement (as they often do – subqueries like this are called *correlated subqueries*).

Examples

```
select id, name, sex, age from actors
  where state = 'Ohio'

select * from birds
  where type = 'flightless'
  order by family, genus, species

select firstname,
       middlename,
       lastname,
       date_of_birth,
       (select name from schools s where p.school = s.id) schoolname
from pupils p
where year_started = '2012'
order by schoolname, date_of_birth
```

Selecting from a stored procedure

A *selectable stored procedure* is a procedure that:

- contains at least one output parameter, and
- utilizes the SUSPEND keyword so the caller can fetch the output rows one by one, just as when selecting from a table or view.

The output parameters of a selectable stored procedure correspond to the columns of a regular table.

Selecting from a stored procedure without input parameters is just like selecting from a table or view:

```
select * from suspicious_transactions
  where assignee = 'John'
```

Any required input parameters must be specified after the procedure name, enclosed in parentheses:

```
select name, az, alt from visible_stars('Brugge', current_date, '22:30')
  where alt >= 20
  order by az, alt
```

Values for optional parameters (that is, parameters for which default values have been defined) may be omitted or provided. However, if you provide them only partly, the parameters you omit must all be at the tail end.

Supposing that the procedure `visible_stars` from the previous example has two optional parameters: *`min_magn`* (numeric(3,1)) and *`spectral_class`* (varchar(12)), the following queries are all valid:

```
select name, az, alt from visible_stars('Brugge', current_date, '22:30')
```

```
select name, az, alt from visible_stars('Brugge', current_date, '22:30', 4
select name, az, alt from visible_stars('Brugge', current_date, '22:30', 4
```

But this one isn't, because there's a “hole” in the parameter list:

```
select name, az, alt from visible_stars('Brugge', current_date, '22:30',
```

An alias for a selectable stored procedure is specified *after* the parameter list:

```
select number,
       (select name from contestants c where c.number = gw.number)
from get_winners('#34517', 'AMS') gw
```

If you qualify a column (output parameter) with the full procedure name, don't include the parameter list:

```
select number,
       (select name from contestants c where c.number = get_winners.numbe
from get_winners('#34517', 'AMS')
```

Selecting from a derived table

A derived table is a valid SELECT statement enclosed in parentheses, optionally followed by a table alias and/or column aliases. The result set of the statement acts as a virtual table which the enclosing statement can query.

Derived tables are discussed in detail in the section *Derived tables (“SELECT FROM SELECT”)*. Here, we only give an example.

Suppose we have a table COEFFS which contains the coefficients of a number of quadratic equations we have to solve. It has been defined like this:

```
create table coeffs (
  a double precision not null,
  b double precision not null,
  c double precision not null,
  constraint chk_a_not_zero check (a <> 0)
)
```

Depending on the values of a, b and c, each equation may have zero, one or two solutions in #. It is possible to find these solutions with a single-level query on table COEFFS, but the code will look rather messy and several values (like the discriminant) will have to be calculated multiple times per row. A derived table can help keep things clean here:

```
select
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
  from
    (select b, b*b - 4*a*c, 2*a from coeffs) (b, D, denom)
```

If we want to show the coefficients next to the solutions (which may not be a bad idea), we can alter the query like this:

```
select
  a, b, c,
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
  from
    (select a, b, c, b*b - 4*a*c as D, 2*a as denom
     from coeffs)
```

Notice that whereas the first query used a column aliases list for the derived table, the second adds aliases internally where needed. Both methods work, as long as every column is guaranteed to have a name.

Selecting from a CTE

A common table expression or CTE is a more complex, but also more powerful type of derived table. A preamble, starting with the keyword WITH, defines one or more named CTE's, each optionally with a column aliases list. The main query, which follows the preamble, can then access these CTE's as if they were regular tables or views. Once the main query has run to completion, the CTE's go out of scope.

For a full discussion of CTE's, please refer to the section *Common Table Expressions ("WITH ... AS ... SELECT")*.

The following is a rewrite of our derived table example as a CTE:

```
with vars (b, D, denom) as (
  select b, b*b - 4*a*c, 2*a from coeffs
)
select
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
from vars
```

Except for the fact that the calculations that have to be made first are now at the beginning, this isn't a great improvement over the derived table version. But we can now also eliminate the double calculation of sqrt(D) for every row:

```
with vars (b, D, denom) as (
  select b, b*b - 4*a*c, 2*a from coeffs
),
vars2 (b, D, denom, sqrtD) as (
  select b, D, denom, iif (D >= 0, sqrt(D), null) from vars
)
select
  iif (D >= 0, (-b - sqrtD) / denom, null) sol_1,
  iif (D >  0, (-b + sqrtD) / denom, null) sol_2
from vars2
```

The code is a little more complicated now, but it might execute more efficiently (depending on what takes more time: executing the SQRT function or passing the values of `b`, `D` and `denom` through an extra CTE). Incidentally, we could have done the same with derived tables, but that would involve nesting.

Joins

Joins combine data from two sources into a single set. This is done on a row-by-row basis and usually involves the checking of a *join condition* in order to determine which rows should be merged and appear in the resulting dataset. There are several types (INNER, OUTER) and classes (qualified, natural, etc.) of joins, each with their own syntax and rules.

Since joins can be chained, the datasets involved in a join may themselves be joined sets.

Syntax.

```
SELECT
   ...
   FROM <source>
   [<joins>]
```

```
   [...]

<source>          ::=  {table
                          | view
                          | selectable-stored-procedure [(args)]
                          | derived-table
                          | common-table-expression}
                       [[AS] alias]

<joins>           ::=  <join> [<join> ...]

<join>            ::=  [<join-type>] JOIN <source> <join-condition>
                         | NATURAL [<join-type>] JOIN <source>
                         | {CROSS JOIN | ,} <source>

<join-type>       ::=  INNER | {LEFT | RIGHT | FULL} [OUTER]

<join-condition>  ::=  ON condition | USING (column-list)
```

Inner vs. outer joins

A join always combines data rows from two sets (usually referred to as the left set and the right set). By default, only rows that meet the join condition (i.e., that match at least one row in the other set when the join condition is applied) make it into the result set. This default type of join is called an *inner join*. Suppose we have the following two tables:

Table A:

ID	S
87	Just some text
235	Silence

Table B:

CODE	X
-23	56.7735
87	416.0

If we join these tables like this:

```
select *
  from A
  join B on A.id = B.code
```

then the result set will be:

ID	S	CODE	X
87	Just some text	87	416.0

The first row of A has been joined with the second row of B because together they met the condition “A.id = B.code”. The other rows from the source tables have no match in the opposite set and are therefore not included in the join. Remember, this is an INNER join. We can make that fact explicit by writing:

```
select *
  from A
  inner join B on A.id = B.code
```

However, since INNER is the default, this is rarely done.

It is perfectly possible that a row in the left set matches several rows from the right set or vice versa. In that case, all those combinations are included, and we can get results like:

ID	S	CODE	X
87	Just some text	87	416.0
87	Just some text	87	-1.0
-23	Don't know	-23	56.7735
-23	Still don't know	-23	56.7735
-23	I give up	-23	56.7735

Sometimes we want (or need) *all* the rows of one or both of the sources to appear in the joined set, regardless of whether they match a record in the other source. This is where outer joins come in. A LEFT outer join includes all the records from the left set, but only matching records from the right set. In a RIGHT outer join it's the other way around. FULL outer joins include all the records from both sets. In all outer joins, the “holes” (the places where an included source record doesn't have a match in the other set) are filled up with NULLs.

In order to make an outer join, you must specify LEFT, RIGHT or FULL, optionally followed by the keyword OUTER.

Below are the results of the various outer joins when applied to our original tables A and B:

```
select *
  from A
  left [outer] join B on A.id = B.code
```

ID	S	CODE	X
87	Just some text	87	416.0
235	Silence	*<null>*	*<null>*

```
select *
  from A
  right [outer] join B on A.id = B.code
```

ID	S	CODE	X
<null>	*<null>*	-23	56.7735
87	Just some text	87	416.0

```
select *
  from A
  full [outer] join B on A.id = B.code
```

ID	S	CODE	X
<null>	*<null>*	-23	56.7735
87	Just some text	87	416.0
235	Silence	*<null>*	*<null>*

Qualified joins

Qualified joins specify conditions for the combining of rows. This happens either explicitly in an ON clause or implicitly in a USING clause.

Syntax.

```
<qualified-join>  ::=  [<join-type>] JOIN <source> <join-condition>

<join-type>       ::=  INNER | {LEFT | RIGHT | FULL} [OUTER]

<join-condition>  ::=  ON condition | USING (column-list)
```

Explicit-condition joins

Most qualified joins have an ON clause, with an explicit condition that can be any valid boolean expression but usually involves some comparison between the two sources involved.

Quite often, the condition is an equality test (or a number of ANDed equality tests) using the "=" operator. Joins like these are called *equi-joins*. (The examples in the section on inner and outer joins were al equi-joins.)

Examples of joins with an explicit condition:

```
/* Select all Detroit customers who made a purchase
   in 2013, along with the purchase details: */
select * from customers c
  join sales s on s.cust_id = c.id
  where c.city = 'Detroit' and s.year = 2013

/* Same as above, but include non-buying customers: */
select * from customers c
  left join sales s on s.cust_id = c.id
  where c.city = 'Detroit' and s.year = 2013

/* For each man, select the women who are taller than he.
   Men for whom no such woman exists are not included. */
select m.fullname as man, f.fullname as woman
  from males m
  join females f on f.height > m.height

/* Select all pupils with their class and mentor.
   Pupils without a mentor are also included.
   Pupils without a class are not included. */
select p.firstname, p.middlename, p.lastname,
       c.name, m.name
  from pupils p
  join classes c on c.id = p.class
  left join mentors m on m.id = p.mentor
```

Named columns joins

Equi-joins often compare columns that have the same name in both tables. If this is the case, we can also use the second type of qualified join: the *named columns join*. Named columns joins have a USING clause which states just the column names. So instead of this:

```
select * from flotsam f
  join jetsam j
  on f.sea = j.sea
  and f.ship = j.ship
```

we can also write:

```
select * from flotsam
  join jetsam using (sea, ship)
```

which is considerably shorter. The result set is a little different though – at least when using “SELECT *”:

- The explicit-condtion join – with the ON clause – will contain each of the columns SEA and SHIP twice: once from table FLOTSAM, and once from table JETSAM. Obviously, they will have the same values.
- The named columns join – with the USING clause – will contain these columns only once.

If you want all the columns in the result set of the named columns join, set up your query like this:

```
select f.*, j.*
  from flotsam f
  join jetsam j using (sea, ship)
```

This will give you the exact same result set as the explicit-condition join.

For an OUTER named columns join, there's an additional twist when using “SELECT *” or an unqualified column name from the USING list:

If a row from one source set doesn't have a match in the other but must still be included because of the LEFT, RIGHT or FULL directive, the merged column in the joined set gets the non-NULL value. That is fair enough, but now you can't tell whether this value came from the left set, the right set, or both. This can be especially deceiving when the value came from the right hand set, because “*” always shows combined columns in the left hand part – even in the case of a RIGHT join.

Whether this is a problem or not depends on the situation. If it is, use the “a.*, b.*” approach shown above, with a and b the names or aliases of the two sources. Or better yet, avoid “*” altogether in your serious queries and qualify all column names in joined sets. This has the additional benefit that it forces you to think about which data you want to retrieve and where from.

It is your responsibility to make sure that the column names in the USING list are of compatible types between the two sources. If the types are compatible but not equal, the engine converts them to the type with the broadest range of values before comparing the values. This will also be the data type of the merged column that shows up in the result set if “SELECT *” or the unqualified column name is used. Qualified columns on the other hand will always retain their original data type.

Natural joins

Taking the idea of the named columns join a step further, a *natural join* performs an automatic equi-join on all the columns that have the same name in the left and right table. The data types of these columns must be compatible.

Syntax.

```
<natural-join>  ::=  NATURAL [<join-type>] JOIN <source>

<join-type>     ::=  INNER | {LEFT | RIGHT | FULL} [OUTER]
```

Given these two tables:

```
create table TA (
  a bigint,
  s varchar(12),
  ins_date date
)

create table TB (
  a bigint,
  descr varchar(12),
```

```
  x float,
  ins_date date
)
```

a natural join on TA and TB would involve the columns *a* and *ins_date*, and the following two statements would have the same effect:

```
select * from TA
  natural join TB

select * from TA
  join TB using (a, ins_date)
```

Like all joins, natural joins are inner joins by default, but you can turn them into outer joins by specifying LEFT, RIGHT or FULL before the JOIN keyword.

Caution: if there are no columns with the same name in the two source relations, a CROSS JOIN is performed. We'll get to this type of join in a minute.

A note on equality

The "=" operator, which is explicitly used in many conditional joins and implicitly in named column joins and natural joins, only matches values to values. According to the SQL standard, NULL is not a value and hence two NULLs are neither equal nor unequal to one another. If you need NULLs to match each other in a join, use the IS NOT DISTINCT FROM operator. This operator returns true if the operands have the same value *or* if they are both NULL.

```
select *
  from A join B
  on A.id is not distinct from B.code
```

Likewise, in the – extremely rare – cases where you want to join on *in*equality, use IS DISTINCT FROM, not "<>", if you want NULL to be considered different from any value and two NULLs considered equal:

```
select *
  from A join B
  on A.id is distinct from B.code
```

This note about equality and inequality operators applies everywhere in Firebird SQL, not only in join conditions.

Cross joins

A cross join produces the full set product of the two data sources. This means that it successfully matches every row in the left source to every row in the right source.

Syntax.

```
<cross-join>  ::=  {CROSS JOIN | ,} <source>
```

Please notice that the comma syntax is deprecated! It is only supported to keep legacy code working and may disappear in some future version.

Cross-joining two sets is equivalent to joining them on a tautology (a condition that is always true). The following two statements have the same effect:

```
select * from TA
  cross join TB

select * from TA
```

```
  join TB on 1 = 1
```

Cross joins are inner joins, because they only include matching records – it just so happens that *every* record matches! An outer cross join, if it existed, wouldn't add anything to the result, because what outer joins add are non-matching records, and these don't exist in cross joins.

Cross joins are seldom useful, except if you want to list all the possible combinations of two or more variables. Suppose you are selling a product that comes in different sizes, different colors and different materials. If these variables are each listed in a table of their own, this query would return all the combinations:

```
select m.name, s.size, c.name
  from materials m
  cross join sizes s
  cross join colors c
```

Ambiguous field names in joins

Firebird rejects unqualified field names in a query if these field names exist in more than one dataset involved in a join. This is even true for inner equi-joins where the field name figures in the ON clause like this:

```
select a, b, c
  from TA
  join TB on TA.a = TB.a
```

There is one exception to this rule: with named columns joins and natural joins, the unqualified field name of a column taking part in the matching process may be used legally and refers to the merged column of the same name. For named columns joins, these are the columns listed in the USING clause. For natural joins, they are the columns that have the same name in both relations. But please notice again that, especially in outer joins, plain *`colname`* isn't always the same as *`left.colname`* or *`right.colname`*. Types may differ, and one of the qualified columns may be NULL while the other isn't. In that case, the value in the merged, unqualified column may mask the fact that one of the source values is absent.

The WHERE clause

The WHERE clause serves to limit the rows returned to the ones that the caller is interested in. The condition following the keyword WHERE can be as simple as a check like “AMOUNT = 3” or it can be a multilayered, convoluted expression containing subselects, predicates, function calls, mathematical and logical operators, context variables and more.

The condition in the WHERE clause is often called the *search condition*, the *search expression* or simply the *search*.

In DSQL and ESQL, the search expression may contain parameters. This is useful if a query has to be repeated a number of times with different input values. In the SQL string as it is passed to the server, question marks are used as placeholders for the parameters. They are called *positional parameters* because they can only be told apart by their position in the string. Connectivity libraries often support *named parameters* of the form *`:id`*, *`:amount`*, *`:a`* etc. These are more user-friendly; the library takes care of translating the named parameters to positional parameters before passing the statement to the server.

The search condition may also contain local (PSQL) or host (ESQL) variable names, preceded by a colon.

Syntax.

```
SELECT ...
   FROM ...
```

```
   [...]
   WHERE <search-condition>
   [...]

<search-condition>  ::=  a boolean expression returning
                         TRUE, FALSE or possibly UNKNOWN (NULL)
```

Only those rows for which the search condition evaluates to TRUE are included in the result set. Be careful with possible NULL outcomes: if you negate a NULL expression with NOT, the result will still be NULL and the row will not pass. This is demonstrated in one of the examples below.

Examples

```
select genus, species from mammals
  where family = 'Felidae'
  order by genus

select * from persons
  where birthyear in (1880, 1881)
     or birthyear between 1891 and 1898

select name, street, borough, phone
  from schools s
  where exists (select * from pupils p where p.school = s.id)
  order by borough, street

select * from employees
  where salary >= 10000 and position <> 'Manager'

select name from wrestlers
  where region = 'Europe'
    and weight > all (select weight from shot_putters
                      where region = 'Africa')

select id, name from players
  where team_id = (select id from teams where name = 'Buffaloes')

select sum (population) from towns
  where name like '%dam'
  and province containing 'land'

select password from usertable
  where username = current_user
```

The following example shows what can happen if the search condition evaluates to NULL.

Suppose you have a table listing some children's names and the number of marbles they possess. At a certain moment, the table contains these data:

CHILD	**MARBLES**
Anita	23
Bob E.	12
Chris	*<null>*
Deirdre	1
Eve	17
Fritz	0
Gerry	21

CHILD	MARBLES
Hadassah	*<null>*
Isaac	6

First, please notice the difference between `NULL` and 0: Fritz is *known* to have no marbles at all, Chris's and Hadassah's marble counts are unknown.

Now, if you issue this SQL statement:

```
select list(child) from marbletable where marbles > 10
```

you will get the names Anita, Bob E., Eve and Gerry. These children all have more than 10 marbles.

If you negate the expression:

```
select list(child) from marbletable where not marbles > 10
```

it's the turn of Deirdre, Fritz and Isaac to fill the list. Chris and Hadassah are not included, because they aren't *known* to have ten marbles or less. Should you change that last query to:

```
select list(child) from marbletable where marbles <= 10
```

the result will still be the same, because the expression `NULL <= 10` yields `UNKNOWN`. This is not the same as `TRUE`, so Chris and Hadassah are not listed. If you want them listed with the "poor" children, change the query to:

```
select list(child) from marbletable where marbles <= 10 or marbles is nul
```

Now the search condition becomes true for Chris and Hadassah, because "`marbles is null`" obviously returns `TRUE` in their case. In fact, the search condition cannot be `NULL` for anybody now.

Lastly, two examples of SELECT queries with parameters in the search. It depends on the application how you should define query parameters and even if it is possible at all. Notice that queries like these cannot be executed immediately: they have to be *prepared* first. Once a parameterized query has been prepared, the user (or calling code) can supply values for the parameters and have it executed many times, entering new values before every call. How the values are entered and the execution started is up to the application. In a GUI environment, the user typically types the parameter values in one or more text boxes and then clicks an "Execute", "Run" or "Refresh" button.

```
select name, address, phone frome stores
  where city = ? and class = ?

select * from pants
  where model = :model and size = :size and color = :col
```

The last query cannot be passed directly to the engine; the application must convert it to the other format first, mapping named parameters to positional parameters.

The GROUP BY clause

GROUP BY merges output rows that have the same combination of values in its item list into a single row. Aggregate functions in the select list are applied to each group individually instead of to the dataset as a whole.

If the select list only contains aggregate columns – or, more generally, columns whose values don't depend on individual rows in the underlying set – GROUP BY is optional. When omitted, the final result set of will consist of a single row (provided that at least one aggregated column is present).

If the select list contains both aggregate columns and columns whose values may vary per row, the GROUP BY clause becomes mandatory.

Syntax.

```
SELECT ... FROM ...
   GROUP BY <grouping-item> [, <grouping-item> ...]
   [HAVING <grouped-row-condition>]
   ...

<grouping-item>          ::=  <non-aggr-select-item>
                                | <non-aggr-expression>

<non-aggr-select-item>   ::=  column-copy
                                | column-alias
                                | column-position

<non-aggr-expression>    ::=  any non-aggregate expression that is not
                              in the select list, e.g. unselected columns
                              from the source set or expressions that
                              don't depend on the data in the set at all
```

A general rule of thumb is that every non-aggregate item in the SELECT list must also be in the GROUP BY list. You can do this in three ways:

1. By copying the item verbatim from the select list, e.g. "`class`" or "`'D:' || upper(doccode)`".
2. By specifying the column alias, if it exists.
3. By specifying the column position as an integer *literal* between 1 and the number of columns. Integer values resulting from expressions or parameter substitutions are simply invariables and will be used as such in the grouping. They will have no effect though, as their value is the same for each row.

Please notice: If you group by a column position, the expression at that position is copied internally from the select list. If it concerns a subquery, that subquery will be executed at least twice.

In addition to the required items, the grouping list may also contain:

- Columns from the source table that are not in the select list, or non-aggregate expressions based on such columns. Adding such columns may further subdivide the groups. But since these columns are not in the select list, you can't tell which aggregated row corresponds to which value in the column. So, in general, if you are interested in this information, you also include the column or expression in the select list – which brings you back to the standard "every non-aggregate column in the select list must also be in the grouping list" mantra.
- Expressions that aren't dependent on the data in the underlying set, e.g. constants, context variables, single-value non-correlated subselects etc. This is only mentioned for completeness, as adding such items is utterly pointless: they don't affect the grouping at all. "Harmless but useless" items like these may also figure in the select list without being copied to the grouping list.

Examples

When the select list only contains aggregate columns, GROUP BY is not mandatory:

```
select count(*), avg(age) from students
  where sex = 'M'
```

This will return a single row listing the number of male students and their average age. Adding expressions that don't depend on values in individual rows of table STUDENTS doesn't change that:

```
select count(*), avg(age), current_date from students
```

```
  where sex = 'M'
```

The row will now have an extra column showing the current date, but other than that, nothing fundamental has changed. A GROUP BY clause is still not required.

However, in both the above examples it is *allowed*. This is perfectly valid:

```
select count(*), avg(age) from students
  where sex = 'M'
  group by class
```

and will return a row for each class that has boys in it, listing the number of boys and their average age in that particular class. (If you also leave the `current_date` field in, this value will be repeated on every row, which is not very exciting.)

The above query has a major drawback though: it gives you information about the different classes, but it doesn't tell you which row applies to which class. In order to get that extra bit of information, the non-aggregate column CLASS must be added to the select list:

```
select class, count(*), avg(age) from students
  where sex = 'M'
  group by class
```

Now we have a useful query. Notice that the addition of column CLASS also makes the GROUP BY clause mandatory. We can't drop that clause anymore, unless we also remove CLASS from the column list.

The output of our last query may look something like this:

CLASS	**COUNT**	**AVG**
2A	12	13.5
2B	9	13.9
3A	11	14.6
3B	12	14.4
...	...	...

The headings “COUNT” and “AVG” are not very informative. In a simple case like this, you might get away with that, but in general you should give aggregate columns a meaningful name by aliasing them:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
```

As you may recall from the formal syntax of the columns list, the AS keyword is optional.

Adding more non-aggregate (or rather: row-dependent) columns requires adding them to the GROUP BY clause too. For instance, you might want to see the above information for girls as well; and you may also want to differentiate between boarding and day students:

```
select class,
       sex,
       boarding_type,
       count(*) as number,
       avg(age) as avg_age
  from students
```

```
group by class, sex, boarding_type
```

This may give you the following result:

CLASS	SEX	BOARDING_TYPE	NUMBER	AVG_AGE
2A	F	BOARDING	9	13.3
2A	F	DAY	6	13.5
2A	M	BOARDING	7	13.6
2A	M	DAY	5	13.4
2B	F	BOARDING	11	13.7
2B	F	DAY	5	13.7
2B	M	BOARDING	6	13.8
...	...	...	...	...

Each row in the result set corresponds to one particular combination of the variables class, sex and boarding type. The aggregate results – number and average age – are given for each of these rather specific groups individually. In a query like this, you don't see a total for boys as a whole, or day students as a whole. That's the tradeoff: the more non-aggregate columns you add, the more you can pinpoint very specific groups, but the more you also lose sight of the general picture. Of course you can still obtain the “coarser” aggregates through separate queries.

HAVING

Just as a WHERE clause limits the rows in a dataset to those that meet the search condition, so the HAVING subclause imposes restrictions on the aggregated rows in a grouped set. HAVING is optional, and can only be used in conjunction with GROUP BY.

The condition(s) in the HAVING clause can refer to:

- Any aggregated column in the select list. This is the most widely used alternative.
- Any aggregated expression that is not in the select list, but allowed in the context of the query. This is sometimes useful too.
- Any column in the GROUP BY list. While legal, it is more efficient to filter on these non-aggregated data at an earlier stage: in the WHERE clause.
- Any expression whose value doesn't depend on the contents of the dataset (like a constant or a context variable). This is valid but utterly pointless, because it will either suppress the entire set or leave it untouched, based on conditions that have nothing to do with the set itself.

A HAVING clause can *not* contain:

- Non-aggregated column expressions that are not in the GROUP BY list.
- Column positions. An integer in the HAVING clause is just an integer.
- Column aliases – not even if they appear in the GROUP BY clause!

Examples

Building on our earlier examples, this could be used to skip small groups of students:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
```

```
  where sex = 'M'
  group by class
  having count(*) >= 5
```

To select only groups that have a minimum age spread:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
  having max(age) - min(age) > 1.2
```

Notice that if you're really interested in this information, you'd normally include `min(age)` and `max(age)` – or the expression "`max(age) - min(age)`" – in the select list as well!

To include only 3rd classes:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
  having class starting with '3'
```

Better would be to move this condition to the WHERE clause:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M' and class starting with '3'
  group by class
```

The PLAN clause

The PLAN clause enables the user to submit a data retrieval plan, thus overriding the plan that the optimizer would have generated automatically.

Syntax.

```
PLAN <plan-expr>

<plan-expr>    ::=  (<plan-item> [, <plan-item> ...])
                  | <sorted-item>
                  | <joined-item>
                  | <merged-item>

<sorted-item>  ::=  SORT (<plan-item>)

<joined-item>  ::=  JOIN (<plan-item>, <plan-item> [, <plan-item> ...])

<merged-item>  ::=  [SORT] MERGE (<sorted-item>, <sorted-item> [, <sorted

<plan-item>    ::=  <basic-item> | <plan-expr>

<basic-item>   ::=  <relation>
```

```
                        {NATURAL
                         | INDEX (<indexlist>)
                         | ORDER index [INDEX (<indexlist>)]}

<relation>      ::=  table
                     | view [table]

<indexlist>     ::=  index [, index ...]

table, view     ::=  name or alias
```

Every time a user submits a query to the Firebird engine, the optimizer computes a data retrieval strategy. Most Firebird clients can make this retrieval plan visible to the user. In Firebird's own isql utility, this is done with the command SET PLAN ON. If you are studying query plans rather than running queries, SET PLANONLY ON will show the plan without executing the query.

In most situations, you can trust that Firebird will select the optimal query plan for you. However, if you have complicated queries that seem to be underperforming, it may very well be worth your while to examine the plan and see if you can improve on it.

Simple plans

The simplest plans consist of just a relation name followed by a retrieval method. E.g., for an unsorted single-table select without a WHERE clause:

```
select * from students
  plan (students natural)
```

If there's a WHERE or a HAVING clause, you can specify the index to be used for finding matches:

```
select * from students
  where class = '3C'
  plan (students index (ix_stud_class))
```

The INDEX directive is also used for join conditions (to be discussed a little later). It can contain a list of indexes, separated by commas.

ORDER specifies the index for sorting the set if an ORDER BY or GROUP BY clause is present:

```
select * from students
  plan (students order pk_students)
  order by id
```

ORDER and INDEX can be combined:

```
select * from students
  where class >= '3'
  plan (students order pk_students index (ix_stud_class))
  order by id
```

It is perfectly OK if ORDER and INDEX specify the same index:

```
select * from students
  where class >= '3'
  plan (students order ix_stud_class index (ix_stud_class))
  order by class
```

For sorting sets when there's no usable index available (or if you want to suppress its use), leave out ORDER and prepend the plan expression with SORT:

```
select * from students
```

```
  plan sort (students natural)
  order by name
```

Or when an index is used for the search:

```
select * from students
  where class >= '3'
  plan sort (students index (ix_stud_class))
  order by name
```

Notice that SORT, unlike ORDER, is outside the parentheses. This reflects the fact that the data rows are retrieved unordered and sorted afterwards by the engine.

When selecting from a view, specify the view and the table involved. For instance, if you have a view FRESHMEN that selects just the first-year students:

```
select * from freshmen
  plan (freshmen students natural)
```

Or, for instance:

```
select * from freshmen
  where id > 10
  plan sort (freshmen students index (pk_students))
  order by name desc
```

Please notice: if you have aliased a table or view, you must use the alias name, not the original name, in the PLAN clause.

Composite plans

When a join is made, you can specify the index which is to be used for matching. You must also use the JOIN directive on the two streams in the plan:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan join (s natural, c index (pk_classes))
```

The same join, sorted on an indexed column:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan join (s order pk_students, c index (pk_classes))
  order by s.id
```

And on a non-indexed column:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan sort (join (s natural, c index (pk_classes)))
  order by s.name
```

With a search added:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  where s.class <= '2'
```

```
  plan sort (join (s index (fk_student_class), c index (pk_classes)))
  order by s.name
```

As a left outer join:

```
select s.id, s.name, s.class, c.mentor
  from classes c
  left join students s on c.name = s.class
  where s.class <= '2'
  plan sort (join (c natural, s index (fk_student_class)))
  order by s.name
```

If there is no index available to match the join criteria (or if you don't want to use it), the plan must first sort both streams on their join column(s) and then merge them. This is achieved with the SORT directive (which we've already met) and MERGE instead of JOIN:

```
select * from students s
  join classes c on c.cookie = s.cookie
  plan merge (sort (c natural), sort (s natural))
```

Adding an ORDER BY clause means the result of the merge must also be sorted:

```
select * from students s
  join classes c on c.cookie = s.cookie
  plan sort (merge (sort (c natural), sort (s natural)))
  order by c.name, s.id
```

Finally, we add a search condition on two indexable colums of table STUDENTS:

```
select * from students s
  join classes c on c.cookie = s.cookie
  where s.id < 10 and s.class <= '2'
  plan sort (merge (sort (c natural),
                    sort (s index (pk_students, fk_student_class))))
  order by c.name, s.id
```

As follows from the formal syntax definition, JOINs and MERGEs in the plan may combine more than two streams. Also, every plan expression may be used as a plan item in an encompassing plan. This means that plans of certain complicated queries may have various nesting levels.

Finally, instead of MERGE you may also write SORT MERGE. As this makes absolutely no difference and may create confusion with “real” SORT directives (the ones that *do* make a difference), it's probably best to stick to plain MERGE.

UNION

A UNION concatenates two or more datasets, thus increasing the number of rows but not the number of columns. Datasets taking part in a UNION must have the same number of columns, and columns at corresponding positions must be of the same type. Other than that, they may be totally unrelated.

By default, a union suppresses duplicate rows. UNION ALL shows all rows, including any duplicates. The optional DISTINCT keyword makes the default behaviour explicit.

Syntax.

```
<union>              ::=  <individual-select>
                          UNION [DISTINCT | ALL]
                          <individual-select>
                          [UNION [DISTINCT | ALL]
                           <individual-select>
```

```
                                ...]
                               [<union-wide-clauses>]

<individual-select>   ::=  SELECT
                           [TRANSACTION name]
                           [FIRST <m>] [SKIP <n>]
                           [DISTINCT | ALL] <columns>
                           [INTO <host-varlist>]
                           FROM source [[AS] alias]
                           [<joins>]
                           [WHERE <condition>]
                           [GROUP BY <grouping-list>
                           [HAVING <aggregate-condition>]]
                           [PLAN <plan-expr>]

<union-wide-clauses>  ::=  [ORDER BY <ordering-list>]
                           [ROWS m [TO n]]
                           [FOR UPDATE [OF <columns>]]
                           [WITH LOCK]
                           [INTO <PSQL-varlist>]
```

Unions take their column names from the *first* select query. If you want to alias union columns, do so in the column list of the topmost SELECT. Aliases in other participating selects are allowed and may even be useful, but will not propagate to the union level.

If a union has an ORDER BY clause, the only allowed sort items are integer literals indicating 1-based column positions, optionally followed by an ASC/DESC and/or a NULLS FIRST/LAST directive. This also implies that you cannot order a union by anything that isn't a column in the union. (You can, however, wrap it in a derived table, which gives you back all the usual sort options.)

Unions are allowed in subqueries of any kind and can themselves contain subqueries. They can also contain joins, and can take part in a join when wrapped in a derived table.

Examples

This query presents information from different music collections in one dataset using unions:

```
select id, title, artist, length, 'CD' as medium
  from cds
union
select id, title, artist, length, 'LP'
  from records
union
select id, title, artist, length, 'MC'
  from cassettes
order by 3, 2  -- artist, title
```

If `id`, `title`, `artist` and `length` are the only fields in the tables involved, the query can also be written as:

```
select c.*, 'CD' as medium
  from cds c
union
select r.*, 'LP'
  from records r
union
select c.*, 'MC'
  from cassettes c
order by 3, 2  -- artist, title
```

Qualifying the “stars” is necessary here because they aren't the only item in the column list. Notice how the “c” aliases in the first and third select don't bite each other; they don't have union scope, but only apply to their individual select query.

The next query retrieves names and phone numbers from translators and proofreaders. Translators who also work as proofreaders will show up only once in the result set, provided their phone number is the same in both tables. The same result can be obtained without DISTINCT. With ALL, these people would appear twice.

```
select name, phone from translators
  union distinct
select name, telephone from proofreaders
```

A UNION within a subquery:

```
select name, phone, hourly_rate from clowns
where hourly_rate < all
  (select hourly_rate from jugglers
     union
   select hourly_rate from acrobats)
order by hourly_rate
```

MATERIAL COPIED FROM THE LRU

The sections contained herein have been copied from the Firebird 2.5 LRU (Language Reference Update). Some of them can be deleted, others extended to become Language Reference sections, yet others contain material that should be merged with the LR sections above.

FROM LRU: Aggregate functions: Extended functionality

Changed in. 1.5

Description. Several types of mixing and nesting aggregate functions are supported since Firebird 1.5. They will be discussed in the following subsections. To get the complete picture, also look at the SELECT :: GROUP BY sections.

Mixing aggregate functions from different contexts

Firebird 1.5 and up allow the use of aggregate functions from different contexts inside a single expression.

Example.

```
select
  r.rdb$relation_name as "Table name",
  ( select max(i.rdb$statistics) || ' (' || count(*) || ')'
    from rdb$relation_fields rf
    where rf.rdb$relation_name = r.rdb$relation_name
  ) as "Max. IndexSel (# fields)"
from
  rdb$relations r
  join rdb$indices i on (i.rdb$relation_name = r.rdb$relation_name)
group by r.rdb$relation_name
having max(i.rdb$statistics) > 0
order by 2
```

This admittedly rather contrived query shows, in the second column, the maximum index selectivity of any index defined on a table, followed by the table's field count between parentheses. Of course you would normally display the field count in a separate column, or in the column with the table name,

but the purpose here is to demonstrate that you can combine aggregates from different contexts in a single expression.

Warning

Firebird 1.0 also executes this type of query, but gives the wrong results!

Aggregate functions and GROUP BY items inside subqueries

Since Firebird 1.5 it is possible to use aggregate functions and/or expressions contained in the GROUP BY clause inside a subquery.

Examples.

This query returns each table's ID and field count. The subquery refers to `flds.rdb$relation_name`, which is also a GROUP BY item:

```
select
  flds.rdb$relation_name as "Relation name",
  ( select rels.rdb$relation_id
    from rdb$relations rels
    where rels.rdb$relation_name = flds.rdb$relation_name
  ) as "ID",
  count(*) as "Fields"
from rdb$relation_fields flds
group by flds.rdb$relation_name
```

The next query shows the last field from each table and its 1-based position. It uses the aggregate function MAX in a subquery.

```
select
  flds.rdb$relation_name as "Table",
  ( select flds2.rdb$field_name
    from rdb$relation_fields flds2
    where
      flds2.rdb$relation_name = flds.rdb$relation_name
      and flds2.rdb$field_position = max(flds.rdb$field_position)
  ) as "Last field",
  max(flds.rdb$field_position) + 1 as "Last fieldpos"
from rdb$relation_fields flds
group by 1
```

The subquery also contains the GROUP BY item `flds.rdb$relation_name`, but that's not immediately obvious because in this case the GROUP BY clause uses the column number.

Subqueries inside aggregate functions

Using a singleton subselect inside (or as) an aggregate function argument is supported in Firebird 1.5 and up.

Example.

```
select
  r.rdb$relation_name as "Table",
  sum( (select count(*)
        from rdb$relation_fields rf
        where rf.rdb$relation_name = r.rdb$relation_name)
  ) as "Ind. x Fields"
```

```
from
  rdb$relations r
  join rdb$indices i
    on (i.rdb$relation_name = r.rdb$relation_name)
group by
  r.rdb$relation_name
```

Nesting aggregate function calls

Firebird 1.5 allows the indirect nesting of aggregate functions, provided that the inner function is from a lower SQL context. Direct nesting of aggregate function calls, as in "COUNT(MAX(price))", is still forbidden and punishable by exception.

Example. See under *Subqueries inside aggregate functions*, where COUNT() is used inside a SUM().

Aggregate statements: Stricter HAVING and ORDER BY

Firebird 1.5 and above are stricter than previous versions about what can be included in the HAVING and ORDER BY clauses. If, in the context of an aggregate statement, an operand in a HAVING or ORDER BY item contains a column name, it is only accepted if one of the following is true:

- The column name appears in an aggregate function call (e.g. "`HAVING MAX(SALARY) > 10000`").
- The operand equals or is based upon a non-aggregate column that appears in the GROUP BY list (by name or position).

"Is based upon" means that the operand need not be exactly the same as the column name. Suppose there's a non-aggregate column "STR" in the select list. Then it's OK to use expressions like "UPPER(STR)", "STR || '!'" or "SUBSTRING(STR FROM 4 FOR 2)" in the HAVING clause – even if these expressions don't appear as such in the SELECT or GROUP BY list.

FROM LRU: [AS] before relation alias

Added in. IB

Description. The keyword AS can optionally be placed before a relation alias, just as it can be placed before a column alias. This feature dates back to InterBase times, but wasn't documented in the IB Language Reference.

Syntax.

```
SELECT ... FROM <relation> [AS] alias

<relation>  ::=  A table, view, or selectable SP
```

Examples.

```
select order_no, total, fullname
  from orders as o join customers as c on o.cust_id = c.cust_id

select order_no, total, fullname
  from orders o join customers c on o.cust_id = c.cust_id
```

The two queries are fully equivalent.

FROM LRU: COLLATE subclause for text BLOB columns

Added in. 2.0

Description. COLLATE subclauses are now also supported for text BLOBs.

Example.

```
select NameBlob from MyTable
  where NameBlob collate pt_br = 'João'
```

FROM LRU: Common Table Expressions (“WITH ... AS ... SELECT”)

Available in. DSQL, PSQL

A common table expression or CTE can be described as a virtual table or view, defined in a preamble to a main query, and going out of scope after the main query's execution. The main query can reference any CTEs defined in the preamble as if they were regular tables or views. CTEs can be recursive, i.e. self-referencing, but they cannot be nested.

Syntax.

```
<cte-construct>  ::=  <cte-defs>
                      <main-query>

<cte-defs>       ::=  WITH [RECURSIVE] <cte> [, <cte> ...]

<cte>            ::=  name [(<column-list>)] AS (<cte-stmt>)

<column-list>    ::=  column-alias [, column-alias ...]

<cte-stmt>       ::=  any SELECT statement or UNION

<main-query>     ::=  the main SELECT statement, which can refer to the
                      CTEs defined in the preamble
```

Example.

```
with dept_year_budget as (
  select fiscal_year,
         dept_no,
         sum(projected_budget) as budget
  from proj_dept_budget
  group by fiscal_year, dept_no
)
select d.dept_no,
       d.department,
       dyb_2008.budget as budget_08,
       dyb_2009.budget as budget_09
from department d
     left join dept_year_budget dyb_2008
       on d.dept_no = dyb_2008.dept_no
       and dyb_2008.fiscal_year = 2008
     left join dept_year_budget dyb_2009
       on d.dept_no = dyb_2009.dept_no
       and dyb_2009.fiscal_year = 2009
where exists (
  select * from proj_dept_budget b
  where d.dept_no = b.dept_no
)
```

Notes.

- A CTE definition can contain any legal SELECT statement, as long as it doesn't have a “WITH...” preamble of its own (no nesting).
- CTEs defined for the same main query can reference each other, but care should be taken to avoid loops.
- CTEs can be referenced from anywhere in the main query.
- Each CTE can be referenced multiple times in the main query, possibly with different aliases.
- When enclosed in parentheses, CTE constructs can be used as subqueries in SELECT statements, but also in UPDATEs, MERGEs etc.
- In PSQL, CTEs are also supported in FOR loop headers:

```
for with my_rivers as (select * from rivers where owner = 'me')
    select name, length from my_rivers into :rname, :rlen
do
begin
  ..
end
```

Recursive CTEs

A recursive (self-referencing) CTE is a UNION which must have at least one non-recursive member, called the *anchor*. The non-recursive member(s) must be placed before the recursive member(s). Recursive members are linked to each other and to their non-recursive neighbour by UNION ALL operators. The unions between non-recursive members may be of any type.

Recursive CTEs require the RECURSIVE keyword to be present right after WITH. Each recursive union member may reference itself only once, and it must do so in a FROM clause.

A great benefit of recursive CTEs is that they use far less memory and CPU cycles than an equivalent recursive stored procedure.

The execution pattern of a recursive CTE is as follows:

- The engine begins execution from a non-recursive member.
- For each row evaluated, it starts executing each recursive member one by one, using the current values from the outer row as parameters.
- If the currently executing instance of a recursive member produces no rows, execution loops back one level and gets the next row from the outer result set.

Example with a recursive CTE.

```
with recursive
  dept_year_budget as (
    select fiscal_year,
           dept_no,
           sum(projected_budget) as budget
    from proj_dept_budget
    group by fiscal_year, dept_no
  ),
  dept_tree as (
    select dept_no,
           head_dept,
           department,
           cast('' as varchar(255)) as indent
```

```
    from department
    where head_dept is null
    union all
    select d.dept_no,
           d.head_dept,
           d.department,
           h.indent || '  '
    from department d
         join dept_tree h on d.head_dept = h.dept_no
  )
select d.dept_no,
       d.indent || d.department as department,
       dyb_2008.budget as budget_08,
       dyb_2009.budget as budget_09
from dept_tree d
     left join dept_year_budget dyb_2008
       on d.dept_no = dyb_2008.dept_no
       and dyb_2008.fiscal_year = 2008
     left join dept_year_budget dyb_2009
       on d.dept_no = dyb_2009.dept_no
       and dyb_2009.fiscal_year = 2009
```

Notes on recursive CTEs.

- Aggregates (DISTINCT, GROUP BY, HAVING) and aggregate functions (SUM, COUNT, MAX etc) are not allowed in recursive union members.
- A recursive reference cannot participate in an outer join.
- The maximum recursion depth is 1024.

FROM LRU: Derived tables (“SELECT FROM SELECT”)

A derived table is the result set of a SELECT query, used in an outer SELECT as if it were an ordinary table. Put another way, it is a subquery in the FROM clause.

Syntax.

```
(select-query)
   [[AS] derived-table-alias]
   [(<derived-column-aliases>)]

<derived-column-aliases>  ::=  column-alias [, column-alias ...]
```

Examples.

The derived table in the query below (shown in boldface) contains all the relation names in the database followed by their field count. The outer SELECT produces, for each existing field count, the number of relations having that field count.

```
select fieldcount,
       count(relation) as num_tables
from   (select r.rdb$relation_name as relation,
               count(*) as fieldcount
        from   rdb$relations r
               join rdb$relation_fields rf
                 on rf.rdb$relation_name = r.rdb$relation_name
        group by relation)
group by fieldcount
```

A trivial example demonstrating the use of a derived table alias and column aliases list (both are optional):

```
select dbinfo.descr,
       dbinfo.def_charset
from   (select * from rdb$database) dbinfo
         (descr, rel_id, sec_class, def_charset)
```

Notes.

- Derived tables can be nested.
- Derived tables can be unions and can be used in unions. They can contain aggregate functions, subselects and joins, and can themselves be used in aggregate functions, subselects and joins. They can also be or contain queries on selectable stored procedures. They can have WHERE, ORDER BY and GROUP BY clauses, FIRST, SKIP or ROWS directives, etc. etc.
- Every column in a derived table *must* have a name. If it doesn't have one by nature (e.g. because it's a constant) it must either be given an alias in the usual way, or a column aliases list must be added to the derived table specification.
- The column aliases list is optional, but if it is used it must be complete. That is: it must contain an alias for every column in the derived table.
- The optimizer can handle a derived table very efficiently. However, if the derived table is involved in an inner join and contains a subquery, then no join order can be made.

FROM LRU: FIRST and SKIP

Available in. DSQL, PSQL

Added in. 1.0

Changed in. 1.5

Better alternative. ROWS

Description. FIRST limits the output of a query to the first so-many rows. SKIP will suppress the given number of rows before starting to return output.

Tip

In Firebird 2.0 and up, use the SQL-compliant ROWS syntax instead.

Syntax.

```
SELECT [FIRST (<int-expr>)] [SKIP (<int-expr>)] <columns> FROM ...

<int-expr>  ::=  Any expression evaluating to an integer.
<columns>   ::=  The usual output column specifications.
```

Note

If *<int-expr>* is an integer literal or a query parameter, the "()" may be omitted. Subselects on the other hand require an extra pair of parentheses.

FIRST and SKIP are both optional. When used together as in "FIRST *m* SKIP *n*", the *n* topmost rows of the output set are discarded and the first *m* rows of the remainder are returned.

SKIP 0 is allowed, but of course rather pointless. FIRST 0 is allowed in version 1.5 and up, where it returns an empty set. In 1.0.x, FIRST 0 causes an error. Negative SKIP and/or FIRST values always result in an error.

If a SKIP lands past the end of the dataset, an empty set is returned. If the number of rows in the dataset (or the remainder after a SKIP) is less than the value given after FIRST, that smaller number of rows is returned. These are valid results, not error situations.

Examples.

The following query will return the first 10 names from the People table:

```
select first 10 id, name from People
  order by name asc
```

The following query will return everything *but* the first 10 names:

```
select skip 10 id, name from People
  order by name asc
```

And this one returns the last 10 rows. Notice the double parentheses:

```
select skip ((select count(*) - 10 from People))
  id, name from People
  order by name asc
```

This query returns rows 81–100 of the People table:

```
select first 20 skip 80 id, name from People
  order by name asc
```

Two Gotchas with FIRST in subselects

- This:

  ```
  delete from MyTable where ID in (select first 10 ID from MyTabl
  ```

  will delete all of the rows in the table. Ouch! The sub-select is evaluating each 10 candidate rows for deletion, deleting them, slipping forward 10 more... ad infinitum, until there are no rows left. Beware! Or better: use the ROWS syntax, available since Firebird 2.0.

- Queries like:

  ```
  ...where F1 in (select first 5 F2 from Table2 order by 1 desc)
  ```

  won't work as expected, because the optimization performed by the engine transforms the IN predicate to the correlated EXISTS predicate shown below. It's obvious that in this case FIRST *N* doesn't make any sense:

  ```
  ...where exists
     ( select first 5 F2 from Table2
       where Table2.F2 = Table1.F1
       order by 1 desc )
  ```

FROM LRU: GROUP BY

Description. GROUP BY merges rows that have the same combination of values and/or NULLs in the item list into a single row. Any aggregate functions in the select list are applied to each group individually instead of to the dataset as a whole.

Syntax.

```
SELECT ... FROM ...
   GROUP BY <item> [, <item> ...]
   ...

<item>  ::=  column-name [COLLATE collation-name]
               | column-alias
               | column-position
               | expression
```

- Only non-negative integer *literals* will be interpreted as column positions. If they are outside the range from 1 to the number of columns, an error is raised. Integer values resulting from expressions or parameter substitutions are simply invariables and will be used as such in the grouping. They will have no effect though, as their value is the same for each row.
- A GROUP BY item cannot be a reference to an aggregate function (including one that is buried inside an expression) from the same context.
- The select list may not contain expressions that can have different values within a group. To avoid this, the rule of thumb is to include each non-aggregate item from the select list in the GROUP BY list (whether by copying, alias or position).

Note. If you group by a column position, the expression at that position is copied internally from the select list. If it concerns a subquery, that subquery will be executed at least twice.

Grouping by alias, position and expressions

Changed in. 1.0, 1.5, 2.0

Description. In addition to column names, Firebird 2 allows column aliases, column positions and arbitrary valid expressions as GROUP BY items.

Examples.

These three queries all achieve the same result:

```
select strlen(lastname) as len_name, count(*)
  from people
  group by len_name

select strlen(lastname) as len_name, count(*)
  from people
  group by 1

select strlen(lastname) as len_name, count(*)
  from people
  group by strlen(lastname)
```

History. Grouping by UDF results was added in Firebird 1. Grouping by column positions, CASE outcomes and a limited number of internal functions in Firebird 1.5. Firebird 2 added column aliases and expressions in general as valid GROUP BY items (“expressions in general” absorbing the UDF, CASE and internal functions lot).

FROM LRU: HAVING: Stricter rules

Changed in. 1.5

Description. See *Aggregate statements: Stricter HAVING and ORDER BY*.

FROM LRU: JOIN

Ambiguous field names rejected

Changed in. 1.0

Description. InterBase 6 accepts and executes statements like the one below, which refers to an unqualified column name even though that name exists in both tables participating in the JOIN:

```
select buses.name, garages.name
  from buses join garages on buses.garage_id = garage.id
  where name = 'Phideaux III'
```

The results of such a query are unpredictable. Firebird Dialect 3 returns an error if there are ambiguous field names in JOIN statements. Dialect 1 gives a warning but will execute the query anyway.

CROSS JOIN

Added in. 2.0

Description. Firebird 2.0 and up support CROSS JOIN, which performs a full set multiplication on the tables involved. Previously you had to achieve this by joining on a tautology (a condition that is always true) or by using the comma syntax, now deprecated.

Syntax.

```
SELECT ...
   FROM <relation> CROSS JOIN <relation>
   ...

<relation>  ::=  {table | view | cte | (select_stmt)} [[AS] alias]
```

Note: If you use CROSS JOIN, you can't use ON.

Example.

```
select * from Men cross join Women
order by Men.age, Women.age

-- old syntax:
--   select * from Men join Women on 1 = 1
--   order by Men.age, Women.age

-- comma syntax:
--   select * from Men, Women
--   order by Men.age, Women.age
```

Named columns JOIN

Added in. 2.1

Description. A named columns join is an equi-join on the columns named in the USING clause. These columns must exist in both relations.

Syntax.

```
SELECT ...
   FROM <relation> [<join_type>] JOIN <relation>
   USING (colname [, colname ...])
```

```
   ...

<relation>   ::=  {table | view | cte | (select_stmt)} [[AS] alias]
<join_type>  ::=  INNER | {LEFT | RIGHT | FULL} [OUTER]
```

Example.

```
select *
  from books join shelves
  using (shelf, bookcase)
```

The equivalent in traditional syntax:

```
select *
  from books b join shelves s
  on b.shelf = s.shelf and b.bookcase = s.bookcase
```

Notes.

- The columns in the USING clause can be selected without qualifier. Be aware, however, that doing so in outer joins doesn't always give the same result as selecting *left.colname* or *right.colname*. One of the latter may be NULL while the other isn't; plain *colname* always returns the non-NULL alternative in such cases.
- SELECT * from a named columns join returns each USING column only once. In outer joins, such a column always contains the non-NULL alternative except for rows where the field is NULL in both tables.

Natural JOIN

Added in. 2.1

Description. A natural join is an automatic equi-join on all the columns that exist in both relations. If there are no common column names, a CROSS JOIN is produced.

Syntax.

```
SELECT ...
   FROM <relation> NATURAL [<join_type>] JOIN <relation>
   ...

<relation>   ::=  {table | view | cte | (select_stmt)} [[AS] alias]
<join_type>  ::=  INNER | {LEFT | RIGHT | FULL} [OUTER]
```

Example.

```
select * from Pupils natural left join Tutors
```

Assuming that the Pupils and Tutors tables have two field names in common: TUTOR and CLASS, the equivalent traditional syntax is:

```
select * from Pupils p left join Tutors t
  on p.tutor = t.tutor and p.class = t.class
```

Notes.

- Common columns can be selected from a natural join without qualifier. Beware, however, that doing so in outer joins doesn't always gives the same result as selecting *left.colname* or *right.colname*. One of the latter may be NULL while the other isn't; plain *colname* always returns the non-NULL alternative in such cases.

- SELECT * from a natural join returns each common column only once. In outer joins, such a column always contains the non-NULL alternative except for rows where the field is NULL in both tables.

FROM LRU: ORDER BY

Syntax.

```
SELECT ... FROM ...
   ...
   ORDER BY <ordering-item> [, <ordering-item> ...]

<ordering-item>  ::=  {col-name | col-alias | col-position | expression}
                         [COLLATE collation-name]
                         [ASC[ENDING] | DESC[ENDING]]
                         [NULLS {FIRST|LAST}]
```

Order by column alias

Added in. 2.0

Description. Firebird 2.0 and above support ordering by column alias.

Example.

```
select rdb$character_set_id as charset_id,
       rdb$collation_id as coll_id,
       rdb$collation_name as name
from rdb$collations
order by charset_id, coll_id
```

Ordering by column position causes * expansion

Changed in. 2.0

Description. If you order by column position in a “SELECT *” query, the engine will now expand the * to determine the sort column(s).

Examples.

The following wasn't possible in pre-2.0 versions:

```
select * from rdb$collations
order by 3, 2
```

The following would sort the output set on `Films.Director` in previous versions. In Firebird 2 and up, it will sort on the second column of `Books`:

```
select Books.*, Films.Director from Books, Films
order by 2
```

Ordering by expressions

Added in. 1.5

Description. Firebird 1.5 introduced the possibility to use expressions as ordering items. Please note that expressions consisting of a single non-negative whole number will be interpreted as column positions and cause an exception if they're not in the range from 1 to the number of columns.

Example.

```
select x, y, note from Pairs
```

```
order by x+y desc
```

Note

The number of function or procedure invocations resulting from a sort based on a UDF or stored procedure is unpredictable, regardless whether the ordering is specified by the expression itself or by the column position number.

Notes.

- The number of function or procedure invocations resulting from a sort based on a UDF or stored procedure is unpredictable, regardless whether the ordering is specified by the expression itself or by the column position number.
- Only non-negative whole number *literals* are interpreted as column positions. A whole number resulting from an expression evaluation or parameter substitution is seen as an integer invariable and will lead to a dummy sort, since its value is the same for each row.

NULLs placement

Changed in. 1.5, 2.0

Description. Firebird 1.5 has introduced the per-column NULLS FIRST and NULLS LAST directives to specify where NULLs appear in the sorted column. Firebird 2.0 has changed the default placement of NULLs.

Unless overridden by NULLS FIRST or NULLS LAST, NULLs in ordered columns are placed as follows:

- In Firebird 1.0 and 1.5: at the end of the sort, regardless whether the order is ascending or descending.
- In Firebird 2.0 and up: at the *start* of ascending orderings and at the *end* of descending orderings.

See also the table below for an overview of the different versions.

Table 6.1. NULLs placement in ordered columns

Ordering	NULLs placement		
	Firebird 1	Firebird 1.5	Firebird 2
order by Field [asc]	bottom	bottom	top
order by Field desc	bottom	bottom	bottom
order by Field [asc | desc] nulls first	—	top	top
order by Field [asc | desc] nulls last	—	bottom	bottom

Notes

- Pre-existing databases may need a backup-restore cycle before they show the correct NULL ordering behaviour under Firebird 2.0 and up.
- No index will be used on columns for which a non-default NULLS placement is chosen. In Firebird 1.5, that is the case with NULLS FIRST. In 2.0 and higher, with NULLS LAST on ascending and NULLS FIRST on descending sorts.

Examples.

```
select * from msg
  order by process_time desc nulls first

select * from document
```

```
  order by strlen(description) desc
  rows 10

select doc_number, doc_date from payorder
union all
select doc_number, doc_date from budgorder
  order by 2 desc nulls last, 1 asc nulls first
```

Stricter ordering rules with aggregate statements

Changed in. 1.5

Description. See *Aggregate statements: Stricter HAVING and ORDER BY*.

FROM LRU: PLAN

Available in. DSQL, ESQL, PSQL

Description. Specifies a user plan for the data retrieval, overriding the plan that the optimizer would have generated automatically.

Syntax.

```
PLAN <plan-expr>

<plan-expr>    ::=  (<plan-item> [, <plan-item> ...])
                  | <sorted-item>
                  | <joined-item>
                  | <merged-item>

<sorted-item>  ::=  SORT (<plan-item>)

<joined-item>  ::=  JOIN (<plan-item>, <plan-item> [, <plan-item> ...])

<merged-item>  ::=  [SORT] MERGE (<sorted-item>, <sorted-item> [, <sorted

<plan-item>    ::=  <basic-item> | <plan-expr>

<basic-item>   ::=  <relation>
                    {NATURAL                                  -- natural s
                     | INDEX (<indexlist>)                    -- indexed r
                     | ORDER index [INDEX (<indexlist>)]}     -- navigatio

<relation>     ::=  table
                  | view [table]                              -- en een SP

<indexlist>    ::=  index [, index ...]

table, view    ::=  name or alias
```

Handling of user PLANs improved

Changed in. 2.0

Description. Firebird 2 has implemented the following improvements in the handling of user-specified PLANs:

- Plan fragments are propagated to nested levels of joins, enabling manual optimization of complex outer joins.

- User-supplied plans will be checked for correctness in outer joins.
- Short-circuit optimization for user-supplied plans has been added.
- A user-specified access path can be supplied for any SELECT-based statement or clause.

ORDER with INDEX

Changed in. 2.0

Description. A single plan item can now contain both an ORDER and an INDEX directive (in that order).

Example.

```
plan (MyTable order ix_myfield index (ix_this, ix_that))
```

PLAN must include all tables

Changed in. 2.0

Description. In Firebird 2 and up, a PLAN clause must handle all the tables in the query. Previous versions sometimes accepted incomplete plans, but this is no longer the case.

FROM LRU: Relation alias makes real name unavailable

Changed in. 2.0

Description. If you give a table or view an alias in a Firebird 2.0 or above statement, you *must* use the alias, not the table name, if you want to qualify fields from that relation.

Examples.

Correct usage:

```
select pears from Fruit

select Fruit.pears from Fruit

select pears from Fruit F

select F.pears from Fruit F
```

No longer possible:

```
select Fruit.pears from Fruit F
```

FROM LRU: ROWS

Available in. DSQL, PSQL

Added in. 2.0

Description. Limits the amount of rows returned by the SELECT statement to a specified number or range.

Syntax.

With a single SELECT:

```
SELECT <columns> FROM ...
   [WHERE ...]
```

```
   [ORDER BY ...]
   ROWS <m> [TO <n>]

<columns>  ::=  The usual output column specifications.
<m>, <n>   ::=  Any expression evaluating to an integer.
```

With a UNION:

```
SELECT [FIRST p] [SKIP q] <columns> FROM ... [WHERE ...]
UNION [ALL | DISTINCT]
SELECT [FIRST r] [SKIP s] <columns> FROM ... [WHERE ...]
[ORDER BY ...]
ROWS <m> [TO <n>]
```

With a single argument *m*, the first *m* rows of the dataset are returned.

Points to note:

- If *m* > the total number of rows in the dataset, the entire set is returned.
- If *m* = 0, an empty set is returned.
- If *m* < 0, an error is raised.

With two arguments *m* and *n*, rows *m* to *n* of the dataset are returned, inclusively. Row numbers are 1-based.

Points to note when using two arguments:

- If *m* > the total number of rows in the dataset, an empty set is returned.
- If *m* lies within the set but *n* doesn't, the rows from *m* to the end of the set are returned.
- If *m* < 1 or *n* < 1, an error is raised.
- If *n* = *m*-1, an empty set is returned.
- If *n* < *m*-1, an error is raised.

The SQL-compliant ROWS syntax obviates the need for FIRST and SKIP, except in one case: a SKIP without FIRST, which returns the entire remainder of the set after skipping a given number of rows. (You can often “fake it” though, by supplying a second argument that you know to be bigger than the number of rows in the set.)

You cannot use ROWS together with FIRST and/or SKIP in a single SELECT statement, but is it valid to use one form in the top-level statement and the other in subselects, or to use the two syntaxes in different subselects.

When used with a UNION, the ROWS subclause applies to the UNION as a whole and must be placed after the last SELECT. If you want to limit the output of one or more individual SELECTs within the UNION, you have two options: either use FIRST/SKIP on those SELECT statements (probably of limited use, as you can't use ORDER BY on individual selects within a union), or convert them to derived tables with ROWS clauses.

ROWS can also be used with the UPDATE and DELETE statements.

FROM LRU: UNION

Available in. DSQL, ESQL, PSQL

UNIONs in subqueries

Changed in. 2.0

Description. UNIONs are now allowed in subqueries. This applies not only to column-level subqueries in a SELECT list, but also to subqueries in ANY|SOME, ALL and IN predicates, as well as the optional SELECT expression that feeds an INSERT.

Example.

```
select name, phone, hourly_rate from clowns
where hourly_rate < all
  (select hourly_rate from jugglers
     union
   select hourly_rate from acrobats)
order by hourly_rate
```

UNION DISTINCT

Added in. 2.0

Description. You can now use the optional DISTINCT keyword when defining a UNION. This will show duplicate rows only once instead of every time they occur in one of the tables. Since DISTINCT, being the opposite of ALL, is the default mode anyway, this doesn't add any new functionality.

Syntax.

```
SELECT (...) FROM (...)
UNION [DISTINCT | ALL]
SELECT (...) FROM (...)
```

Example.

```
select name, phone from translators
  union distinct
select name, phone from proofreaders
```

Translators who also work as proofreaders (a not uncommon combination) will show up only once in the result set, provided their phone number is the same in both tables. The same result would have been obtained without DISTINCT. With ALL, they would appear twice.

FROM LRU: WITH LOCK

Available in. DSQL, PSQL

Added in. 1.5

Description: WITH LOCK provides a limited explicit pessimistic locking capability for cautious use in conditions where the affected row set is:

a. extremely small (ideally, a singleton), *and*
b. precisely controlled by the application code.

This is for experts only!

The need for a pessimistic lock in Firebird is very rare indeed and should be well understood before use of this extension is considered.

It is essential to understand the effects of transaction isolation and other transaction attributes before attempting to implement explicit locking in your application.

Syntax.

```
SELECT ... FROM single_table
   [WHERE ...]
   [FOR UPDATE [OF ...]]
```

```
WITH LOCK
```

If the WITH LOCK clause succeeds, it will secure a lock on the selected rows and prevent any other transaction from obtaining write access to any of those rows, or their dependants, until your transaction ends.

If the FOR UPDATE clause is included, the lock will be applied to each row, one by one, as it is fetched into the server-side row cache. It becomes possible, then, that a lock which appeared to succeed when requested will nevertheless *fail subsequently*, when an attempt is made to fetch a row which becomes locked by another transaction.

WITH LOCK can only be used with a top-level, single-table SELECT statement. It is *not* available:

- in a subquery specification;
- for joined sets;
- with the DISTINCT operator, a GROUP BY clause or any other aggregating operation;
- with a view;
- with the output of a selectable stored procedure;
- with an external table.

A lengthier, more in-depth discussion of “SELECT ... WITH LOCK” is included in the Notes. It is a must-read for everybody who considers using this feature.

UPDATE

Available in. DSQL, ESQL, PSQL

Description. Changes values in a table or in one or more tables underlying a view. The columns affected are specified in the SET clause; the rows affected may be limited by the WHERE and ROWS clauses. If neither WHERE nor ROWS is present, all the records in the table will be updated.

Syntax.

```
UPDATE [TRANSACTION name] target [[AS] alias]
   SET col = newval [, col = newval ...]
   [WHERE {search-conditions | CURRENT OF cursorname}]
   [PLAN plan_items]
   [ORDER BY sort_items]
   [ROWS <m> [TO <n>]]
   [RETURNING <values> [INTO <variables>]]

target       ::=  A table or updatable view
<m>, <n>     ::=  Any expression evaluating to an integer.
<values>     ::=  value_expression [, value_expression ...]
<variables>  ::=  :varname [, :varname ...]
```

Restrictions

- The TRANSACTION directive is only available in ESQL.
- In a pure DSQL session, WHERE CURRENT OF isn't of much use, since there exists no DSQL statement to create a cursor.
- No column may be SET more than once in the same UPDATE statement.
- The “INTO <*variables*>” subclause is only available in PSQL.
- When returning values into the context variable NEW, this name must not be preceded by a colon (“:”).
- The PLAN, ORDER BY, ROWS and RETURNING clauses are not available in ESQL.

Using an alias

If you give a table or view an alias, you *must* use the alias, not the table name, if you want to qualify fields from that relation.

Examples.

Correct usage:

```
update Fruit set soort = 'pisang' where ...

update Fruit set Fruit.soort = 'pisang' where ...

update Fruit F set soort = 'pisang' where ...

update Fruit F set F.soort = 'pisang' where ...
```

Not possible:

```
update Fruit F set Fruit.soort = 'pisang' where ...
```

The SET clause

The SET clause specifies the values to be written to the affected row(s). It is a comma-separated list of assignments, each with a column name on the left hand side and a value expression on the right hand side.

String literals may optionally be preceded by a character set name, using *introducer syntax*, in order to let the engine know how to interpret the input.

Examples.

```
update addresses
  set city = 'Saint Petersburg', citycode = 'PET'
  where city = 'Leningrad'

update employees
  set salary = 2.5 * salary
  where title = 'CEO'

update People
  set name = _ISO8859_1 'Hans-Jörg Schäfer'  -- notice the '_' prefix
  where id = 53662
```

It is perfectly legal to include target columns in the value expressions on the right hand side, providing values for themselves and/or other columns. If this is done, the value used on the right hand side will always be the *old* column value, even if an assignment has already been made to the same column earlier in the list. The following example illustrates this.

Given table TSET:

```
A  B
----
1  0
2  0
```

the following statement:

```
update tset set a=5, b=a
```

will change its state to

```
A  B
----
5  1
5  2
```

Notice how the old values of a (1 and 2) are used for the update of b, even though a itself has already been assigned a new value (5).

Note

This has not always been the case. In pre-2.5 versions of Firebird, new column values became immediately available for subsequent assignments in the list. This is non-standard behaviour; hence it has been changed.

However, if the *OldSetClauseSemantics* parameter in firebird.conf has been set to 1, Firebird will continue to show the old behaviour. This parameter will be deprecated and removed somewhere in the future.

The WHERE clause

A WHERE clause limits the update action to the rows matching the search condition, or – in ESQL and PSQL only – to the current row of a named cursor.

Examples.

```
update People set firstname = 'Boris' where lastname = 'Johnson'

update Cities set name = :arg_name where current of Cur_Cities;  -- ESQL
```

An update using WHERE CURRENT OF is called a *positioned update*, because it updates the record at the current position. An update using "WHERE *<condition>*" is called a *searched update*, because the engine has to search for the record(s) meeting the condition.

ORDER BY and ROWS

If at all, ORDER BY and ROWS only make sense when used together. However, they are also valid separately.

With a single argument *m*, ROWS limits the update to the first *m* rows of the dataset defined by the table or view and the optional WHERE and ORDER BY clauses.

With two arguments *m* and *n*, the update is limited to rows *m* to *n* inclusively. Row numbers are 1-based.

Example.

```
-- give the 20 poorest guys a break:
update employees
  set salary = salary + 50
  order by salary asc
  rows 20
```

Points to note when using ROWS with one argument:

- If *m* > the total number of rows in the dataset, the entire set is updated.
- If *m* = 0, no rows are updated.
- If *m* < 0, an error is raised.

Points to note when using ROWS with two arguments:

- If *m* > the total number of rows in the dataset, no rows are updated.
- If *m* lies within the set but *n* doesn't, the rows from *m* to the end of the set are updated.
- If *m* < 1 or *n* < 1, an error is raised.
- If *n* = *m*-1, no rows are updated.
- If *n* < *m*-1, an error is raised.

RETURNING

An UPDATE statement modifying *at most one row* may optionally include a RETURNING clause in order to return values from the updated row. The clause, if present, need not contain all the modified columns and may also contain other columns or expressions. The returned values reflect any changes that may have been made in BEFORE triggers, but not those in AFTER triggers. OLD.*fieldname* and NEW.*fieldname* may both be used in the list of columns to return; for field names not preceded by either of these, the new value is returned.

Example.

```
update Scholars
  set firstname = 'Hugh', lastname = 'Pickering'
  where firstname = 'Henry' and lastname = 'Higgins'
  returning id, old.lastname, new.lastname
```

In DSQL, a statement with a RETURNING clause *always* returns exactly one row. If no record was actually updated, the fields in this row are all `NULL`. This behaviour may change in a later version of Firebird. In PSQL, if no row was updated, nothing is returned, and the target variables keep their existing values.

Updating BLOB columns

Updating a BLOB column always replaces the entire contents. Even the BLOB ID, the “handle” that is stored directly in the column, is changed. BLOBs can be updated if:

1. The client application has made special provisions for this operation, using the Firebird API. In this case, the modus operandi is application-specific and outside the scope of this manual.

2. The new value is a text string of at most 32767 bytes. Please notice: if the value is not a string literal, beware of concatenations, as these may exceed the maximum length.

3. The source is itself a BLOB column or, more generally, an expression that returns a BLOB.

4. You use the INSERT CURSOR statement (ESQL only).

UPDATE OR INSERT

Available in. DSQL, PSQL

UPDATE OR INSERT inserts a new record or updates one or more existing records. The action taken depends on the values provided for the columns in the MATCHING clause (or, if the latter is absent, in the primary key). If there are records found matching those values, they are updated. If not, a new record is inserted.

A match only counts if all the values in the MATCHING or PK columns are equal. Matching is done with the IS NOT DISTINCT operator, so one `NULL` matches another.

Syntax.

```
UPDATE OR INSERT INTO
   {tablename | viewname} [(<columns>)]
```

```
    VALUES (<values>)
    [MATCHING (<columns>)]
    [RETURNING <values> [INTO <variables>]]

<columns>      ::=  colname  [, colname  ...]
<values>       ::=  value    [, value    ...]
<variables>    ::=  :varname [, :varname ...]
```

Restrictions

- If the table has no PK, the MATCHING clause becomes mandatory.
- In the MATCHING list as well as in the update/insert column list, each column name may occur only once.
- The "INTO *<variables>*" subclause is only available in PSQL.
- When values are returned into the context variable NEW, this name must not be preceded by a colon (":").

Example.

```
update or insert into Cows (Name, Number, Location)
  values ('Suzy Creamcheese', 3278823, 'Green Pastures')
  matching (Number)
  returning rec_id into :id;
```

The RETURNING clause

The optional RETURNING clause, if present, need not contain all the columns mentioned in the statement and may also contain other columns or expressions. The returned values reflect any changes that may have been made in BEFORE triggers, but not those in AFTER triggers. OLD.*fieldname* and NEW.*fieldname* may both be used in the list of columns to return; for field names not preceded by either of these, the new value is returned.

In DSQL, a statement with a RETURNING clause *always* returns exactly one row. If a RETURNING clause is present and more than one matching record is found, an error is raised. This behaviour may change in a later version of Firebird.

Chapter 7. Built-in functions and variables

Context variables

CURRENT_CONNECTION

Available in. DSQL, PSQL

Added in. 1.5

Changed in. 2.1

Description. `CURRENT_CONNECTION` contains the unique identifier of the current connection.

Type. INTEGER

Examples.

```
select current_connection from rdb$database

execute procedure P_Login(current_connection)
```

The value of `CURRENT_CONNECTION` is stored on the database header page and reset to 0 upon restore. Since version 2.1, it is incremented upon every new connection. (In previous versions, it was only incremented if the client read it during a session.) As a result, `CURRENT_CONNECTION` now indicates the number of connections since the creation – or most recent restoration – of the database.

CURRENT_DATE

Available in. DSQL, PSQL, ESQL

Description. `CURRENT_DATE` returns the current server date.

Type. DATE

Syntax.

```
CURRENT_DATE
```

Examples.

```
select current_date from rdb$database
-- returns e.g. 2011-10-03
```

Notes.

- Within a PSQL module (procedure, trigger or executable block), the value of `CURRENT_DATE` will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use `'TODAY'`.

CURRENT_ROLE

Available in. DSQL, PSQL

Added in. 1.0

Description. `CURRENT_ROLE` is a context variable containing the role of the currently connected user. If there is no active role, `CURRENT_ROLE` is NONE.

Type. VARCHAR(31)

Example.

```
if (current_role <> 'MANAGER')
  then exception only_managers_may_delete;
else
  delete from Customers where custno = :custno;
```

`CURRENT_ROLE` always represents a valid role or NONE. If a user connects with a non-existing role, the engine silently resets it to NONE without returning an error.

CURRENT_TIME

Available in. DSQL, PSQL, ESQL

Changed in. 2.0

Description. `CURRENT_TIME` returns the current server time. In versions prior to 2.0, the fractional part used to be always "`.0000`", giving an effective precision of 0 decimals. From Firebird 2.0 onward you can specify a precision when polling this variable. The default is still 0 decimals, i.e. seconds precision.

Type. TIME

Syntax.

```
CURRENT_TIME [(precision)]

precision  ::=  0 | 1 | 2 | 3
```

The optional *`precision`* argument is not supported in ESQL.

Examples.

```
select current_time from rdb$database
-- returns e.g. 14:20:19.6170

select current_time(2) from rdb$database
-- returns e.g. 14:20:23.1200
```

Notes.

- Unlike `CURRENT_TIME`, the default precision of `CURRENT_TIMESTAMP` has changed to 3 decimals. As a result, `CURRENT_TIMESTAMP` is no longer the exact sum of `CURRENT_DATE` and `CURRENT_TIME`, unless you explicitly specify a precision.
- Within a PSQL module (procedure, trigger or executable block), the value of `CURRENT_TIME` will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use `'NOW'`.

CURRENT_TIMESTAMP

Available in. DSQL, PSQL, ESQL

Changed in. 2.0

Description. CURRENT_TIMESTAMP returns the current server date and time. In versions prior to 2.0, the fractional part used to be always ".0000", giving an effective precision of 0 decimals. From Firebird 2.0 onward you can specify a precision when polling this variable. The default is 3 decimals, i.e. milliseconds precision.

Type. TIMESTAMP

Syntax.

```
CURRENT_TIMESTAMP [(precision)]

precision  ::=  0 | 1 | 2 | 3
```

The optional *precision* argument is not supported in ESQL.

Examples.

```
select current_timestamp from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170

select current_timestamp(2) from rdb$database
-- returns e.g. 2008-08-13 14:20:23.1200
```

Notes.

- The default precision of CURRENT_TIME is still 0 decimals, so in Firebird 2.0 and up CURRENT_TIMESTAMP is no longer the exact sum of CURRENT_DATE and CURRENT_TIME, unless you explicitly specify a precision.
- Within a PSQL module (procedure, trigger or executable block), the value of CURRENT_TIMESTAMP will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use 'NOW'.

CURRENT_TRANSACTION

Available in. DSQL, PSQL

Added in. 1.5

Description. CURRENT_TRANSACTION contains the unique identifier of the current transaction.

Type. INTEGER

Examples.

```
select current_transaction from rdb$database

New.Txn_ID = current_transaction;
```

The value of CURRENT_TRANSACTION is stored on the database header page and reset to 0 upon restore. It is incremented with every new transaction.

CURRENT_USER

Available in. DSQL, PSQL

Added in. 1.0

Description. CURRENT_USER is a context variable containing the name of the currently connected user. It is fully equivalent to USER.

Type. VARCHAR(31)

Example.

```
create trigger bi_customers for customers before insert as
begin
  New.added_by  = CURRENT_USER;
  New.purchases = 0;
end
```

DELETING

Available in. PSQL

Added in. 1.5

Description. Available in triggers only, DELETING indicates if the trigger fired because of a DELETE operation. Intended for use in multi-action triggers.

Type. boolean

Example.

```
if (deleting) then
begin
  insert into Removed_Cars (id, make, model, removed)
    values (old.id, old.make, old.model, current_timestamp);
end
```

GDSCODE

Available in. PSQL

Added in. 1.5

Changed in. 2.0

Description. In a “WHEN ... DO” error handling block, the GDSCODE context variable contains the numerical representation of the current Firebird error code. Prior to Firebird 2.0, GDSCODE was only set in WHEN GDSCODE handlers. Now it may also be non-zero in WHEN ANY, WHEN SQLCODE and WHEN EXCEPTION blocks, provided that the condition raising the error corresponds with a Firebird error code. Outside error handlers, GDSCODE is always 0. Outside PSQL it doesn't exist at all.

Type. INTEGER

Example.

```
when gdscode grant_obj_notfound, gdscode grant_fld_notfound,
     gdscode grant_nopriv, gdscode grant_nopriv_on_base
do
begin
  execute procedure log_grant_error(gdscode);
  exit;
end
```

Please notice: After WHEN GDSCODE, you must use symbolic names like grant_obj_notfound etc. But the GDSCODE context variable is an INTEGER. If you want to compare it against a certain error, you have to use the numeric value, e.g. 335544551 for grant_obj_notfound.

INSERTING

Available in. PSQL

Added in. 1.5

Description. Available in triggers only, `INSERTING` indicates if the trigger fired because of an INSERT operation. Intended for use in multi-action triggers.

Type. boolean

Example.

```
if (inserting or updating) then
begin
  if (new.serial_num is null) then
    new.serial_num = gen_id(gen_serials, 1);
end
```

NEW

Available in. PSQL, triggers only

Changed in. 1.5, 2.0

Description. `NEW` contains the new version of a database record that has just been inserted or updated. Starting with Firebird 2.0 it is read-only in AFTER triggers.

Type. Data row

Note

In multi-action triggers – introduced in Firebird 1.5 – `NEW` is always available. But if the trigger is fired by a DELETE, there will be no new version of the record. In that situation, reading from `NEW` will always return `NULL`; writing to it will cause a runtime exception.

'NOW'

Available in. DSQL, PSQL, ESQL

Changed in. 2.0

Description. `'NOW'` is not a variable but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the current date and/or time. The fractional part of the time used to be always "`.0000`", giving an effective seconds precision. Since Firebird 2.0 the precision is 3 decimals, i.e. milliseconds. `'NOW'` is case-insensitive, and the engine ignores leading or trailing spaces when casting.

Note. Please be advised that these shorthand expressions are evaluated immediately at parse time and stay the same as long as the statement remains prepared. Thus, even if a query is executed multiple times, the value for e.g. "timestamp 'now'" won't change, no matter how much time passes. If you need the value to progress (i.e. be evaluated upon every call), use a full cast.

Type. CHAR(3)

Examples.

```
select 'Now' from rdb$database
-- returns 'Now'
```

```
select cast('Now' as date) from rdb$database
-- returns e.g. 2008-08-13

select cast('now' as time) from rdb$database
-- returns e.g. 14:20:19.6170

select cast('NOW' as timestamp) from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170
```

Shorthand syntax for the last three statements:

```
select date 'Now' from rdb$database
select time 'now' from rdb$database
select timestamp 'NOW' from rdb$database
```

Notes.

- `'NOW'` always returns the actual date/time, even in PSQL modules, where `CURRENT_DATE`, `CURRENT_TIME` and `CURRENT_TIMESTAMP` return the same value throughout the duration of the outermost routine. This makes `'NOW'` useful for measuring time intervals in triggers, procedures and executable blocks.
- Except in the situation mentioned above, reading `CURRENT_DATE`, `CURRENT_TIME` and `CURRENT_TIMESTAMP` is generally preferable to casting `'NOW'`. Be aware though that `CURRENT_TIME` defaults to seconds precision; to get milliseconds precision, use `CURRENT_TIME(3)`.

OLD

Available in. PSQL, triggers only

Changed in. 1.5, 2.0

Description. `OLD` contains the existing version of a database record just before a deletion or update. Starting with Firebird 2.0 it is read-only.

Type. Data row

Note

In multi-action triggers – introduced in Firebird 1.5 – `OLD` is always available. But if the trigger is fired by an INSERT, there is obviously no pre-existing version of the record. In that situation, reading from `OLD` will always return `NULL`; writing to it will cause a runtime exception.

ROW_COUNT

Available in. PSQL

Added in. 1.5

Changed in. 2.0

Description. The `ROW_COUNT` context variable contains the number of rows affected by the most recent DML statement (INSERT, UPDATE, DELETE, SELECT or FETCH) in the current trigger, stored procedure or executable block.

Type. INTEGER

Example.

```
update Figures set Number = 0 where id = :id;
if (row_count = 0) then
  insert into Figures (id, Number) values (:id, 0);
```

Behaviour with SELECT and FETCH.

- After a singleton SELECT, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise.
- In a FOR SELECT loop, `ROW_COUNT` is incremented with every iteration (starting at 0 before the first).
- After a FETCH from a cursor, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise. Fetching more records from the same cursor does *not* increment `ROW_COUNT` beyond 1.
- In Firebird 1.5.*x*, `ROW_COUNT` is 0 after any type of SELECT statement.

Note

`ROW_COUNT` cannot be used to determine the number of rows affected by an EXECUTE STATEMENT or EXECUTE PROCEDURE command.

SQLCODE

Available in. PSQL

Added in. 1.5

Changed in. 2.0

Deprecated in. 2.5.1

Description. In a "WHEN ... DO" error handling block, the `SQLCODE` context variable contains the current SQL error code. Prior to Firebird 2.0, `SQLCODE` was only set in WHEN SQLCODE and WHEN ANY handlers. Now it may also be non-zero in WHEN GDSCODE and WHEN EXCEPTION blocks, provided that the condition raising the error corresponds with an SQL error code. Outside error handlers, `SQLCODE` is always 0. Outside PSQL it doesn't exist at all.

Type. INTEGER

Example.

```
when any
do
begin
  if (sqlcode <> 0) then
    Msg = 'An SQL error occurred!';
  else
    Msg = 'Something bad happened!';
  exception ex_custom Msg;
end
```

Important notice. `SQLCODE` is now deprecated in favour of the SQL-2003-compliant *`SQLSTATE`* status code. Support for `SQLCODE` and WHEN SQLCODE will be discontinued in some future version of Firebird.

SQLSTATE

Available in. PSQL

Added in. 2.5.1

Description. In a “WHEN ... DO” error handler, the SQLSTATE context variable contains the 5-character, SQL-2003-compliant status code resulting from the statement that raised the error. Outside error handlers, SQLSTATE is always '00000'. Outside PSQL it is not available at all.

Type. CHAR(5)

Example.

```
when any
do
begin
  Msg = case sqlstate
          when '22003' then 'Numeric value out of range.'
          when '22012' then 'Division by zero.'
          when '23000' then 'Integrity constraint violation.'
          else 'Something bad happened! SQLSTATE = ' || sqlstate
        end;
  exception ex_custom Msg;
end
```

Notes.

- SQLSTATE is destined to replace SQLCODE. The latter is now deprecated in Firebird and will disappear in some future version.
- Firebird does not (yet) support the syntax “WHEN SQLSTATE ... DO”. You have to use WHEN ANY and test the SQLSTATE variable within the handler.
- Each SQLSTATE code is the concatenation of a 2-character class and a 3-character subclass. Classes 00 (successful completion), 01 (warning) and 02 (no data) represent *completion conditions*. Every status code outside these classes is an *exception*. Because classes 00, 01 and 02 don't raise an error, they won't ever show up in the SQLSTATE variable.
- For a complete listing of SQLSTATE codes, consult the *Appendix to the Firebird 2.5 Release Notes* [http://www.firebirdsql.org/rlsnotesh/rlsnotes25.html#rnfb25-appx-sqlstates].

'TODAY'

Available in. DSQL, PSQL, ESQL

Description. 'TODAY' is not a variable but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the current date. 'TODAY' is case-insensitive, and the engine ignores leading or trailing spaces when casting.

Type. CHAR(5)

Examples.

```
select 'Today' from rdb$database
  -- returns 'Today'

select cast('Today' as date) from rdb$database
  -- returns e.g. 2011-10-03

select cast('TODAY' as timestamp) from rdb$database
  -- returns e.g. 2011-10-03 00:00:00.0000
```

Shorthand syntax for the last two statements:

```
select date 'Today' from rdb$database
  select timestamp 'TODAY' from rdb$database
```

Notes.

- `'TODAY'` always returns the actual date, even in PSQL modules, where `CURRENT_DATE`, `CURRENT_TIME` and `CURRENT_TIMESTAMP` return the same value throughout the duration of the outermost routine. This makes `'TODAY'` useful for measuring time intervals in triggers, procedures and executable blocks (at least if your procedures are running for days).
- Except in the situation mentioned above, reading `CURRENT_DATE`, is generally preferable to casting `'NOW'`.

'TOMORROW'

Available in. DSQL, PSQL, ESQL

Description. `'TOMORROW'` is not a variable but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the date of the next day. See also `'TODAY'`.

Type. CHAR(8)

Examples.

```
select 'Tomorrow' from rdb$database
  -- returns 'Tomorrow'

select cast('Tomorrow' as date) from rdb$database
  -- returns e.g. 2011-10-04

select cast('TOMORROW' as timestamp) from rdb$database
  -- returns e.g. 2011-10-04 00:00:00.0000
```

Shorthand syntax for the last two statements:

```
select date 'Tomorrow' from rdb$database
  select timestamp 'TOMORROW' from rdb$database
```

UPDATING

Available in. PSQL

Added in. 1.5

Description. Available in triggers only, `UPDATING` indicates if the trigger fired because of an UPDATE operation. Intended for use in multi-action triggers.

Type. boolean

Example.

```
if (inserting or updating) then
begin
  if (new.serial_num is null) then
    new.serial_num = gen_id(gen_serials, 1);
end
```

'YESTERDAY'

Available in. DSQL, PSQL, ESQL

Description. `'YESTERDAY'` is not a variable but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the date of the day before. See also `'TODAY'`.

Type. CHAR(9)

Examples.

```
select 'Yesterday' from rdb$database
  -- returns 'Tomorrow'

select cast('Yesterday as date) from rdb$database
  -- returns e.g. 2011-10-02

select cast('YESTERDAY' as timestamp) from rdb$database
  -- returns e.g. 2011-10-02 00:00:00.0000
```

Shorthand syntax for the last two statements:

```
select date 'Yesterday' from rdb$database
  select timestamp 'YESTERDAY' from rdb$database
```

USER

Available in. DSQL, PSQL

Added in. InterBase

Description. USER is a context variable containing the name of the currently connected user. It is fully equivalent to CURRENT_USER.

Type. VARCHAR(31)

Example.

```
create trigger bi_customers for customers before insert as
begin
  New.added_by  = USER;
  New.purchases = 0;
end
```

Scalar functions

ABS()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the absolute value of the argument.

Result type. Numerical

Syntax.

```
ABS (number)
```

Important

If the external function ABS is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ACOS()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the arc cosine of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
ACOS (number)
```

- The result is an angle in the range [0, #].
- If the argument is outside the range [-1, 1], NaN is returned.

Important

If the external function `ACOS` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ASCII_CHAR()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the ASCII character corresponding to the number passed in the argument.

Result type. [VAR]CHAR(1) CHARACTER SET NONE

Syntax.

```
ASCII_CHAR (<code>)

  <code>  ::=  an integer in the range [0..255]
```

Important

- If the external function `ASCII_CHAR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).
- If you are used to the behaviour of the `ASCII_CHAR` UDF, which returns an empty string if the argument is 0, please notice that the internal function correctly returns a character with ASCII code 0 here.

ASCII_VAL()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the ASCII code of the character passed in.

Result type. SMALLINT

Syntax.

```
ASCII_VAL (ch)

  ch  ::=  a [VAR]CHAR or text BLOB of max. 32767 bytes
```

- If the argument is a string with more than one character, the ASCII code of the first character is returned.
- If the argument is an empty string, 0 is returned.
- If the argument is `NULL`, `NULL` is returned.
- If the first character of the argument string is multi-byte, an error is raised. (A bug in Firebird 2.1–2.1.3 and 2.5 causes an error to be raised if *any* character in the string is multi-byte. This is fixed in versions 2.1.4 and 2.5.1.)

Important

If the external function `ASCII_VAL` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ASIN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the arc sine of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
ASIN (number)
```

- The result is an angle in the range [-#/2, #/2].
- If the argument is outside the range [-1, 1], NaN is returned.

Important

If the external function `ASIN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ATAN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the arc tangent of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
ATAN (number)
```

- The result is an angle in the range <-#/2, #/2>.

Important

If the external function ATAN is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ATAN2()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the angle whose sine-to-cosine *ratio* is given by the two arguments, and whose sine and cosine *signs* correspond to the signs of the arguments. This allows results across the entire circle, including the angles -#/2 and #/2.

Result type. DOUBLE PRECISION

Syntax.

```
ATAN2 (y, x)
```

- The result is an angle in the range [-#, #].
- If *x* is negative, the result is # if *y* is 0, and -# if *y* is -0.
- If both *y* and *x* are 0, the result is meaningless. Starting with Firebird 3, an error will be raised if both arguments are 0.

Important

If the external function ATAN2 is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Notes.

- A fully equivalent description of this function is the following: ATAN2(*y*, *x*) is the angle between the positive X-axis and the line from the origin to the point (*x*, *y*). This also makes it obvious that ATAN2(0, 0) is undefined.
- If *x* is greater than 0, ATAN2(*y*, *x*) is the same as ATAN(*y*/*x*).
- If both sine and cosine of the angle are already known, ATAN2(*sin*, *cos*) gives the angle.

BIN_AND()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the result of the bitwise AND operation on the argument(s).

Result type. INTEGER or BIGINT

Syntax.

```
BIN_AND (number [, number ...])
```

Important

If the external function `BIN_AND` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIN_OR()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the result of the bitwise OR operation on the argument(s).

Result type. INTEGER or BIGINT

Syntax.

```
BIN_OR (number [, number ...])
```

Important

If the external function `BIN_OR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIN_SHL()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the first argument bitwise left-shifted by the second argument, i.e. $a << b$ or $a \cdot 2^{\wedge} b$.

Result type. BIGINT

Syntax.

```
BIN_SHL (number, shift)
```

BIN_SHR()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the first argument bitwise right-shifted by the second argument, i.e. $a >> b$ or $a/2^{\wedge} b$.

Result type. BIGINT

Syntax.

```
BIN_SHR (number, shift)
```

- The operation performed is an arithmetic right shift (SAR), meaning that the sign of the first operand is always preserved.

BIN_XOR()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the result of the bitwise XOR operation on the argument(s).

Result type. INTEGER or BIGINT

Syntax.

```
BIN_XOR (number [, number ...])
```

Important

If the external function `BIN_XOR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIT_LENGTH()

Available in. DSQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Gives the length in bits of the input string. For multi-byte character sets, this may be less than the number of characters times 8 times the “formal” number of bytes per character as found in RDB$CHARACTER_SETS.

Note

With arguments of type CHAR, this function takes the entire formal string length (e.g. the declared length of a field or variable) into account. If you want to obtain the “logical” bit length, not counting the trailing spaces, right-TRIM the argument before passing it to BIT_LENGTH.

Result type. INTEGER

Syntax.

```
BIT_LENGTH (str)
```

BLOB support. Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples.

```
select bit_length('Hello!') from rdb$database
  -- returns 48

select bit_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 64: ü and ß take up one byte each in ISO8859_1

select bit_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
  from rdb$database
  -- returns 80: ü and ß take up two bytes each in UTF8
```

```
select bit_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
  from rdb$database
  -- returns 208: all 24 CHAR positions count, and two of them are 16-bit
```

See also. OCTET_LENGTH(), CHARACTER_LENGTH()

CAST()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in. 2.0, 2.1, 2.5

Description. CAST converts an expression to the desired datatype or domain. If the conversion is not possible, an error is raised.

Result type. User-chosen.

Syntax.

```
CAST (expression AS <target_type>)

  <target_type>  ::=  sql_datatype
                    | [TYPE OF] domain
                    | TYPE OF COLUMN relname.colname
```

Shorthand syntax.

Alternative syntax, supported only when casting a string literal to a DATE, TIME or TIMESTAMP:

```
datatype 'date/timestring'
```

This syntax was already available in InterBase, but was never properly documented. *Please notice:* The shorthand syntax is evaluated immediately at parse time, causing the value to stay the same until the statement is unprepared. For datetime literals like `'12-Oct-2012'` this doesn't make any difference. But for the pseudo-variables `'NOW'`, `'YESTERDAY'`, `'TODAY'` and `'TOMORROW'` this may not be what you want. If you need the value to be evaluated at every call, use CAST().

Examples.

A full-syntax cast:

```
select cast ('12' || '-June-' || '1959' as date) from rdb$database
```

A shorthand string-to-date cast:

```
update People set AgeCat = 'Old'
  where BirthDate < date '1-Jan-1943'
```

Notice that you can drop even the shorthand cast from the example above, as the engine will understand from the context (comparison to a DATE field) how to interpret the string:

```
update People set AgeCat = 'Old'
  where BirthDate < '1-Jan-1943'
```

But this is not always possible. The cast below cannot be dropped, otherwise the engine would find itself with an integer to be subtracted from a string:

```
select date 'today' - 7 from rdb$database
```

The following table shows the type conversions possible with CAST.

Table 7.1. Possible CASTs

From	To
Numeric types	Numeric types [VAR]CHAR BLOB
[VAR]CHAR BLOB	[VAR]CHAR BLOB Numeric types DATE TIME TIMESTAMP
DATE TIME	[VAR]CHAR BLOB TIMESTAMP
TIMESTAMP	[VAR]CHAR BLOB DATE TIME

Keep in mind that sometimes information is lost, for instance when you cast a TIMESTAMP to a DATE. Also, the fact that types are CAST-compatible is in itself no guarantee that a conversion will succeed. “CAST(123456789 as SMALLINT)” will definitely result in an error, as will “CAST('Judgement Day' as DATE)”.

Casting input fields. Since Firebird 2.0, you can cast statement parameters to a datatype:

```
cast (? as integer)
```

This gives you control over the type of input field set up by the engine. Please notice that with statement parameters, you always need a full-syntax cast – shorthand casts are not supported.

Casting to a domain or its type. Firebird 2.1 and above support casting to a domain or its base type. When casting to a domain, any constraints (NOT NULL and/or CHECK) declared for the domain must be satisfied or the cast will fail. Please be aware that a CHECK passes if it evaluates to `TRUE` *or* `NULL`! So, given the following statements:

```
create domain quint as int check (value >= 5000)
  select cast (2000 as quint) from rdb$database     -- (1)
  select cast (8000 as quint) from rdb$database     -- (2)
  select cast (null as quint) from rdb$database     -- (3)
```

only cast number (1) will result in an error.

When the TYPE OF modifier is used, the expression is cast to the base type of the domain, ignoring any constraints. With domain quint defined as above, the following two casts are equivalent and will both succeed:

```
select cast (2000 as type of quint) from rdb$database
  select cast (2000 as int) from rdb$database
```

If TYPE OF is used with a (VAR)CHAR type, its character set and collation are retained:

```
create domain iso20 varchar(20) character set iso8859_1;
```

```
create domain dunl20 varchar(20) character set iso8859_1 collate du_nl;
create table zinnen (zin varchar(20));
commit;
insert into zinnen values ('Deze');
insert into zinnen values ('Die');
insert into zinnen values ('die');
insert into zinnen values ('deze');

select cast(zin as type of iso20) from zinnen order by 1;
-- returns Deze -> Die -> deze -> die

select cast(zin as type of dunl20) from zinnen order by 1;
-- returns deze -> Deze -> die -> Die
```

Warning

If a domain's definition is changed, existing CASTs to that domain or its type may become invalid. If these CASTs occur in PSQL modules, their invalidation may be detected. See the note *The RDB$VALID_BLR field*, near the end of this document.

Casting to a column's type. In Firebird 2.5 and above, it is possible to cast expressions to the type of an existing table or view column. Only the type itself is used; in the case of string types, this includes the character set but not the collation. Constraints and default values of the source column are not applied.

```
create table ttt (
  s varchar(40) character set utf8 collate unicode_ci_ai
  );
  commit;

  select cast ('Jag har många vänner' as type of column ttt.s) from rdb$da
```

Warnings

- For text types, character set and collation are preserved by the cast – just as when casting to a domain. However, due to a bug, the collation is not always taken into consideration when comparisons (e.g. equality tests) are made. In cases where the collation is of importance, test your code thoroughly before deploying! This bug is fixed for Firebird 3.
- If a column's definition is altered, existing CASTs to that column's type may become invalid. If these CASTs occur in PSQL modules, their invalidation may be detected. See the note *The RDB$VALID_BLR field*, near the end of this document.

Casting BLOBs. Successful casting to and from BLOBs is possible since Firebird 2.1.

CEIL(), CEILING()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the smallest whole number greater than or equal to the argument.

Result type. BIGINT or DOUBLE PRECISION

Syntax.

```
CEIL[ING] (number)
```

Important

If the external function `CEILING` is declared in your database, it will override the internal function CEILING (but not CEIL). To make the internal function available, DROP or ALTER the external function (UDF).

See also. FLOOR()

CHAR_LENGTH(), CHARACTER_LENGTH()

Available in. DSQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Gives the length in characters of the input string.

Note

With arguments of type CHAR, this function returns the formal string length (i.e. the declared length of a field or variable). If you want to obtain the "logical" length, not counting the trailing spaces, right-TRIM the argument before passing it to CHAR[ACTER]_LENGTH.

Result type. INTEGER

Syntax.

```
CHAR_LENGTH (str)
  CHARACTER_LENGTH (str)
```

BLOB support. Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples.

```
select char_length('Hello!') from rdb$database
  -- returns 6

select char_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 8

select char_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
  from rdb$database
  -- returns 8; the fact that ü and ß take up two bytes each is irrelevan

select char_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
  from rdb$database
  -- returns 24: all 24 CHAR positions count
```

See also. BIT_LENGTH(), OCTET_LENGTH()

CHAR_TO_UUID()

Available in. DSQL, PSQL

Added in. 2.5

Description. Converts a human-readable 36-char UUID string to the corresponding 16-byte UUID.

Result type. CHAR(16) CHARACTER SET OCTETS

Syntax.

```
CHAR_TO_UUID (ascii_uuid)

  ascii_uuid  ::=  a string of length 36 with:
                   * '-' (hyphen) at positions 9, 14, 19 and 24;
                   * valid hex digits at every other position.
```

Examples.

```
select char_to_uuid('A0bF4E45-3029-2a44-D493-4998c9b439A3') from rdb$datab
  -- returns A0BF4E4530292A44D4934998C9B439A3 (16-byte string)

  select char_to_uuid('A0bF4E45-3029-2A44-X493-4998c9b439A3') from rdb$dat
  -- error: -Human readable UUID argument for CHAR_TO_UUID must
  --         have hex digit at position 20 instead of "X (ASCII 88)"
```

See also. UUID_TO_CHAR(), GEN_UUID()

COALESCE()

Available in. DSQL, PSQL

Added in. 1.5

Description. The COALESCE function takes two or more arguments and returns the value of the first non-`NULL` argument. If all the arguments evaluate to `NULL`, the result is `NULL`.

Result type. Depends on input.

Syntax.

```
COALESCE (<exp1>, <exp2> [, <expN> ... ])
```

Example.

```
select
  coalesce (Nickname, FirstName, 'Mr./Mrs.') || ' ' || LastName
  as FullName
  from Persons
```

This example picks the Nickname from the Persons table. If it happens to be `NULL`, it goes on to FirstName. If that too is `NULL`, “Mr./Mrs.” is used. Finally, it adds the family name. All in all, it tries to use the available data to compose a full name that is as informal as possible. Notice that this scheme only works if absent nicknames and first names are really `NULL`: if one of them is an empty string instead, COALESCE will happily return that to the caller.

Note

In Firebird 1.0.x, where COALESCE is not available, you can accomplish the same with the `*nvl` external functions.

COS()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns an angle's cosine. The argument must be given in radians.

Result type. DOUBLE PRECISION

Syntax.

```
COS (angle)
```

- Any non-NULL result is – obviously – in the range [-1, 1].

Important

If the external function COS is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

COSH()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the hyperbolic cosine of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
COSH (number)
```

- Any non-NULL result is in the range [1, INF].

Important

If the external function COSH is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

COT()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns an angle's cotangent. The argument must be given in radians.

Result type. DOUBLE PRECISION

Syntax.

```
COT (angle)
```

Important

If the external function COT is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

DATEADD()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5

Description. Adds the specified number of years, months, weeks, days, hours, minutes, seconds or milliseconds to a date/time value. (The WEEK unit is new in 2.5.)

Result type. DATE, TIME or TIMESTAMP

Syntax.

```
DATEADD (<args>)

  <args>      ::=  <amount> <unit> TO <datetime>
                 | <unit>, <amount>, <datetime>

  <amount>    ::=  an integer expression (negative to subtract)
  <unit>      ::=  YEAR | MONTH | WEEK | DAY
                   | HOUR | MINUTE | SECOND | MILLISECOND
  <datetime>  ::=  a DATE, TIME or TIMESTAMP expression
```

- The result type is determined by the third argument.
- With TIMESTAMP and DATE arguments, all units can be used. (Prior to Firebird 2.5, units smaller than DAY were disallowed for DATEs.)
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Examples.

```
dateadd (28 day to current_date)
  dateadd (-6 hour to current_time)
  dateadd (month, 9, DateOfConception)
  dateadd (-38 week to DateOfBirth)
  dateadd (minute, 90, time 'now')
  dateadd (? year to date '11-Sep-1973')
```

DATEDIFF()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5

Description. Returns the number of years, months, weeks, days, hours, minutes, seconds or milliseconds elapsed between two date/time values. (The WEEK unit is new in 2.5.)

Result type. BIGINT

Syntax.

```
DATEDIFF (<args>)

  <args>     ::=  <unit> FROM <moment1> TO <moment2>
```

```
                | <unit>, <moment1>, <moment2>

<unit>     ::=  YEAR | MONTH | WEEK | DAY
              | HOUR | MINUTE | SECOND | MILLISECOND
<momentN>  ::=  a DATE, TIME or TIMESTAMP expression
```

- DATE and TIMESTAMP arguments can be combined. No other mixes are allowed.
- With TIMESTAMP and DATE arguments, all units can be used. (Prior to Firebird 2.5, units smaller than DAY were disallowed for DATEs.)
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Computation.

- DATEDIFF doesn't look at any smaller units than the one specified in the first argument. As a result,
  - "`datediff (year, date '1-Jan-2009', date '31-Dec-2009')`" returns 0, but
  - "`datediff (year, date '31-Dec-2009', date '1-Jan-2010')`" returns 1
- It does, however, look at all the *bigger* units. So:
  - "`datediff (day, date '26-Jun-1908', date '11-Sep-1973')`" returns 23818
- A negative result value indicates that *`moment2`* lies before *`moment1`*.

Examples.

```
datediff (hour from current_timestamp to timestamp '12-Jun-2059 06:00')
  datediff (minute from time '0:00' to current_time)
  datediff (month, current_date, date '1-1-1900')
  datediff (day from current_date to cast(? as date))
```

DECODE()

Available in. DSQL, PSQL

Added in. 2.1

Description. DECODE is a shortcut for the so-called "simple CASE" construct, in which a given expression is compared to a number of other expressions until a match is found. The result is determined by the value listed after the matching expression. If no match is found, the default result is returned, if present. Otherwise, `NULL` is returned.

Result type. Varies

Syntax.

```
DECODE ( <test-expr>,
       <expr>, result
       [, <expr>, result ...]
       [, defaultresult] )
```

The equivalent CASE construct:

```
CASE <test-expr>
  WHEN <expr> THEN result
  [WHEN <expr> THEN result ...]
  [ELSE defaultresult]
  END
```

Caution

Matching is done with the "=" operator, so if *<test-expr>* is NULL, it won't match any of the *<expr>*s, not even those that are NULL.

Example.

```
select name,
     age,
     decode( upper(sex),
             'M', 'Male',
             'F', 'Female',
             'Unknown' ),
     religion
  from people
```

See also. CASE, Simple CASE

EXP()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the natural exponential, e^{number}

Result type. DOUBLE PRECISION

Syntax.

```
EXP (number)
```

See also. LN()

EXTRACT()

Available in. DSQL, ESQL, PSQL

Added in. IB 6

Changed in. 2.1

Description. Extracts and returns an element from a DATE, TIME or TIMESTAMP expression. This function was already added in InterBase 6, but not documented in the *Language Reference* at the time.

Result type. SMALLINT or NUMERIC

Syntax.

```
EXTRACT (<part> FROM <datetime>)

  <part>      ::=  YEAR | MONTH | WEEK
                 | DAY | WEEKDAY | YEARDAY
                 | HOUR | MINUTE | SECOND | MILLISECOND
  <datetime>  ::=  a DATE, TIME or TIMESTAMP expression
```

The returned data types and possible ranges are shown in the table below. If you try to extract a part that isn't present in the date/time argument (e.g. SECOND from a DATE or YEAR from a TIME), an error occurs.

Table 7.2. Types and ranges of EXTRACT results

Part	Type	Range	Comment
YEAR	SMALLINT	1–9999	
MONTH	SMALLINT	1–12	
WEEK	SMALLINT	1–53	
DAY	SMALLINT	1–31	
WEEKDAY	SMALLINT	0–6	0 = Sunday
YEARDAY	SMALLINT	0–365	0 = January 1
HOUR	SMALLINT	0–23	
MINUTE	SMALLINT	0–59	
SECOND	NUMERIC(9,4)	0.0000–59.9999	includes millisecond as fraction
MILLISECOND	NUMERIC(9,1)	0.0–999.9	broken in 2.1, 2.1.1

MILLISECOND

Added in. 2.1 (with bug)

Fixed in. 2.1.2

Description. Firebird 2.1 and up support extraction of the millisecond from a TIME or TIMESTAMP. The datatype returned is NUMERIC(9,1).

Note

If you extract the millisecond from CURRENT_TIME, be aware that this variable defaults to seconds precision, so the result will always be 0. Extract from CURRENT_TIME(3) or CURRENT_TIMESTAMP to get milliseconds precision.

WEEK

Added in. 2.1

Description. Firebird 2.1 and up support extraction of the ISO-8601 week number from a DATE or TIMESTAMP. ISO-8601 weeks start on a Monday and always have the full seven days. Week 1 is the first week that has a majority (at least 4) of its days in the new year. The first 1–3 days of the year may belong to the last week (52 or 53) of the previous year. Likewise, a year's final 1–3 days may belong to week 1 of the following year.

Caution

Be careful when combining WEEK and YEAR results. For instance, 30 December 2008 lies in week 1 of 2009, so “`extract (week from date '30 Dec 2008')`” returns 1. However, extracting YEAR always gives the calendar year, which is 2008. In this case, WEEK and YEAR are at odds with each other. The same happens when the first days of January belong to the last week of the previous year.

Please also notice that WEEKDAY is *not* ISO-8601 compliant: it returns 0 for Sunday, whereas ISO-8601 specifies 7.

FLOOR()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the largest whole number smaller than or equal to the argument.

Result type. BIGINT or DOUBLE PRECISION

Syntax.

```
FLOOR (number)
```

Important

If the external function FLOOR is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also. CEIL() / CEILING()

GEN_ID()

Available in. DSQL, ESQL, PSQL

Added in. IB

Description. Increments a generator or sequence and returns its new value. From Firebird 2.0 onward, the SQL-compliant NEXT VALUE FOR syntax is preferred, except when an increment other than 1 is needed.

Result type. BIGINT

Syntax.

```
GEN_ID (generator-name, <step>)

  <step>  ::=  An integer expression.
```

Example.

```
new.rec_id = gen_id(gen_recnum, 1);
```

Warning

Unless you know very well what you are doing, using GEN_ID() with step values lower than 1 may compromise your data's integrity.

See also. NEXT VALUE FOR, CREATE GENERATOR

GEN_UUID()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns a universally unique ID as a 16-byte character string.

Result type. CHAR(16) CHARACTER SET OCTETS

Syntax.

```
GEN_UUID ()
```

Example.

```
select gen_uuid() from rdb$database
  -- returns e.g. 017347BFE212B2479C00FA4323B36320 (16-byte string)
```

See also. UUID_TO_CHAR(), CHAR_TO_UUID()

HASH()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns a hash value for the input string. This function fully supports text BLOBs of any length and character set.

Result type. BIGINT

Syntax.

```
HASH (string)
```

IIF()

Available in. DSQL, PSQL

Added in. 2.0

Description. IIF takes three arguments. If the first evaluates to `true`, the second argument is returned; otherwise the third is returned.

Result type. Depends on input.

Syntax.

```
IIF (<condition>, ResultT, ResultF)

  <condition>  ::=  A boolean expression.
```

Example.

```
select iif( sex = 'M', 'Sir', 'Madam' ) from Customers
```

IIF(*Cond*, *Result1*, *Result2*) is a shortcut for "CASE WHEN *Cond* THEN *Result1* ELSE *Result2* END". You can also compare IIF to the ternary "`? :`" operator in C-like languages.

LEFT()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the leftmost part of the argument string. The number of characters is given in the second argument.

Result type. VARCHAR or BLOB

Syntax.

```
LEFT (string, length)
```

- This function fully supports text BLOBs of any length, including those with a multi-byte character set.
- If *string* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the length of the input string.
- If the *length* argument exceeds the string length, the input string is returned unchanged.
- If the *length* argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

See also. RIGHT()

LN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the natural logarithm of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
LN (number)
```

- An error is raised if the argument is negative or 0.

Important

If the external function LN is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also. EXP()

LOG()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5

Description. Returns the *x*-based logarithm of *y*.

Result type. DOUBLE PRECISION

Syntax.

```
LOG (x, y)
```

- If either argument is 0 or below, an error is raised. (Before 2.5, this would result in NaN, ±INF or 0, depending on the exact values of the arguments.)
- If both arguments are 1, NaN is returned.
- If *x* = 1 and *y* < 1, -INF is returned.

- If *x* = 1 and *y* > 1, `INF` is returned.

Important

If the external function `LOG` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

LOG10()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5

Description. Returns the 10-based logarithm of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
LOG10 (number)
```

- An error is raised if the argument is negative or 0. (In versions prior to 2.5, such values would result in `NaN` and `-INF`, respectively.)

Important

If the external function `LOG10` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

LOWER()

Available in. DSQL, ESQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Returns the lower-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are lowercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

Result type. (VAR)CHAR or BLOB

Syntax.

```
LOWER (str)
```

Note

Because LOWER is a reserved word, the internal function will take precedence even if the external function by that name has also been declared. To call the (inferior!) external function, use double-quotes and the exact capitalisation, as in `"LOWER"`(*str*).

Example.

```
select Sheriff from Towns
  where lower(Name) = 'cooper''s valley'
```

See also. UPPER

LPAD()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5 (backported to 2.1.4)

Description. Left-pads a string with spaces or with a user-supplied string until a given length is reached.

Result type. VARCHAR or BLOB

Syntax.

```
LPAD (str, endlen [, padstr])
```

- This function fully supports text BLOBs of any length and character set.
- If *str* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*endlen*).
- If *padstr* is given and equals `''` (empty string), no padding takes place.
- If *endlen* is less than the current string length, the string is truncated to *endlen*, even if *padstr* is the empty string.

Important

If the external function LPAD is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Note

In Firebird 2.1–2.1.3, all non-BLOB results were of type VARCHAR(32765), which made it advisable to cast them to a more modest size. This is no longer the case.

Examples.

```
lpad ('Hello', 12)                 -- returns '       Hello'
  lpad ('Hello', 12, '-')            -- returns '-------Hello'
  lpad ('Hello', 12, '')             -- returns 'Hello'
  lpad ('Hello', 12, 'abc')          -- returns 'abcabcaHello'
  lpad ('Hello', 12, 'abcdefghij')   -- returns 'abcdefgHello'
  lpad ('Hello', 2)                  -- returns 'He'
  lpad ('Hello', 2, '-')             -- returns 'He'
  lpad ('Hello', 2, '')              -- returns 'He'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

See also. RPAD()

MAXVALUE()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the maximum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

Result type. Varies

Syntax.

```
MAXVALUE (expr [, expr ...])
```

- If one or more expressions resolve to `NULL`, MAXVALUE returns `NULL`. This behaviour differs from the aggregate function MAX.

See also. MINVALUE()

MINVALUE()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the minimum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

Result type. Varies

Syntax.

```
MINVALUE (expr [, expr ...])
```

- If one or more expressions resolve to `NULL`, MINVALUE returns `NULL`. This behaviour differs from the aggregate function MIN.

See also. MAXVALUE()

MOD()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the remainder of an integer division.

Result type. INTEGER or BIGINT

Syntax.

```
MOD (a, b)
```

- Non-integer arguments are rounded before the division takes place. So, “7.5 mod 2.5” gives 2 (8 mod 3), not 0.

Important

If the external function `MOD` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

NULLIF()

Available in. DSQL, PSQL

Added in. 1.5

Description. NULLIF returns the value of the first argument, unless it is equal to the second. In that case, `NULL` is returned.

Result type. Depends on input.

Syntax.

```
NULLIF (<exp1>, <exp2>)
```

Example.

```
select avg( nullif(Weight, -1) ) from FatPeople
```

This will return the average weight of the persons listed in FatPeople, excluding those having a weight of -1, since AVG skips `NULL` data. Presumably, -1 indicates "weight unknown" in this table. A plain AVG(Weight) would include the -1 weights, thus skewing the result.

Note

In Firebird 1.0.x, where NULLIF is not available, you can accomplish the same with the `*nullif` external functions.

OCTET_LENGTH()

Available in. DSQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Gives the length in bytes (octets) of the input string. For multi-byte character sets, this may be less than the number of characters times the "formal" number of bytes per character as found in RDB$CHARACTER_SETS.

Note

With arguments of type CHAR, this function takes the entire formal string length (e.g. the declared length of a field or variable) into account. If you want to obtain the "logical" byte length, not counting the trailing spaces, right-TRIM the argument before passing it to OCTET_LENGTH.

Result type. INTEGER

Syntax.

```
OCTET_LENGTH (str)
```

BLOB support. Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples.

```
select octet_length('Hello!') from rdb$database
  -- returns 6
```

```
select octet_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 8: ü and ß take up one byte each in ISO8859_1

select octet_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
  from rdb$database
  -- returns 10: ü and ß take up two bytes each in UTF8

select octet_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
  from rdb$database
  -- returns 26: all 24 CHAR positions count, and two of them are 2-byte
```

See also. BIT_LENGTH(), CHARACTER_LENGTH()

OVERLAY()

Available in. DSQL, PSQL

Added in. 2.1

Description. Overwrites part of a string with another string. By default, the number of characters removed from the host string equals the length of the replacement string. With the optional fourth argument, the user can specify a different number of characters to be removed.

Result type. VARCHAR or BLOB

Syntax.

```
OVERLAY (string PLACING replacement FROM pos [FOR length])
```

- This function supports BLOBs of any length.
- If *string* or *replacement* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the sum of the lengths of *string* and *replacement*.
- As usual in SQL string functions, *pos* is 1-based.
- If *pos* is beyond the end of *string*, *replacement* is placed directly after *string*.
- If the number of characters from *pos* to the end of *string* is smaller than the length of *replacement* (or than the *length* argument, if present), *string* is truncated at *pos* and *replacement* placed after it.
- The effect of a “FOR 0” clause is that *replacement* is simply inserted into *string*.
- If any argument is NULL, the result is NULL.
- If *pos* or *length* is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

Examples.

```
overlay ('Goodbye' placing 'Hello' from 2)    -- returns 'GHelloe'
  overlay ('Goodbye' placing 'Hello' from 5)    -- returns 'GoodHello'
  overlay ('Goodbye' placing 'Hello' from 8)    -- returns 'GoodbyeHello'
  overlay ('Goodbye' placing 'Hello' from 20)   -- returns 'GoodbyeHello'

  overlay ('Goodbye' placing 'Hello' from 2 for 0)   -- r. 'GHellooodbye'
```

```
overlay ('Goodbye' placing 'Hello' from 2 for 3)   -- r. 'GHellobye'
overlay ('Goodbye' placing 'Hello' from 2 for 6)   -- r. 'GHello'
overlay ('Goodbye' placing 'Hello' from 2 for 9)   -- r. 'GHello'

overlay ('Goodbye' placing '' from 4)         -- returns 'Goodbye'
overlay ('Goodbye' placing '' from 4 for 3)   -- returns 'Gooe'
overlay ('Goodbye' placing '' from 4 for 20)  -- returns 'Goo'

overlay ('' placing 'Hello' from 4)           -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 0)     -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 20)    -- returns 'Hello'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

See also. REPLACE()

PI()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns an approximation of the value of #.

Result type. DOUBLE PRECISION

Syntax.

```
PI ()
```

Important

If the external function `PI` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

POSITION()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the (1-based) position of the first occurrence of a substring in a host string. With the optional third argument, the search starts at a given offset, disregarding any matches that may occur earlier in the string. If no match is found, the result is 0.

Result type. INTEGER

Syntax.

```
POSITION (<args>)

  <args>  ::=  substr IN string
             | substr, string [, startpos]
```

- The optional third argument is only supported in the second syntax (comma syntax).

- The empty string is considered a substring of every string. Therefore, if *substr* is '' (empty string) and *string* is not NULL, the result is:
  - 1 if *startpos* is not given;
  - *startpos* if *startpos* lies within *string*;
  - 0 if *startpos* lies beyond the end of *string*.

  Notice: A bug in Firebird 2.1–2.1.3 and 2.5 causes POSITION to *always* return 1 if *substr* is the empty string. This is fixed in 2.1.4 and 2.5.1.

- This function fully supports text BLOBs of any size and character set.

Examples.

```
position ('be' in 'To be or not to be')      -- returns 4
  position ('be', 'To be or not to be')        -- returns 4
  position ('be', 'To be or not to be', 4)     -- returns 4
  position ('be', 'To be or not to be', 8)     -- returns 17
  position ('be', 'To be or not to be', 18)    -- returns 0
  position ('be' in 'Alas, poor Yorick!')      -- returns 0
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

POWER()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns *x* to the *y*'th power.

Result type. DOUBLE PRECISION

Syntax.

```
POWER (x, y)
```

- If *x* negative, an error is raised.

Important

If the external function POWER is declared in your database as power instead of the default dPower, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

RAND()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns a random number between 0 and 1.

Result type. DOUBLE PRECISION

Syntax.

```
RAND ()
```

Important

If the external function RAND is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

RDB$GET_CONTEXT()

Note

RDB$GET_CONTEXT and its counterpart RDB$SET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present – the user doesn't have to do anything to make them available.

Available in. DSQL, ESQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Retrieves the value of a context variable from one of the namespaces SYSTEM, USER_SESSION and USER_TRANSACTION.

Result type. VARCHAR(255)

Syntax.

```
RDB$GET_CONTEXT ('<namespace>', '<varname>')

  <namespace>  ::=  SYSTEM | USER_SESSION | USER_TRANSACTION
  <varname>    ::=  A case-sensitive string of max. 80 characters
```

The namespaces. The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The SYSTEM namespace is read-only. It contains a number of predefined variables, shown in the table below.

Table 7.3. Context variables in the SYSTEM namespace

DB_NAME	Either the full path to the database or – if connecting via the path is disallowed – its alias.
NETWORK_PROTOCOL	The protocol used for the connection: 'TCPv4', 'WNET', 'XNET' or NULL.
CLIENT_ADDRESS	For TCPv4, this is the IP address. For XNET, the local process ID. For all other protocols this variable is NULL.
CURRENT_USER	Same as global CURRENT_USER variable.
CURRENT_ROLE	Same as global CURRENT_ROLE variable.
SESSION_ID	Same as global CURRENT_CONNECTION variable.
TRANSACTION_ID	Same as global CURRENT_TRANSACTION variable.
ISOLATION_LEVEL	The isolation level of the current transaction: 'READ COMMITTED', 'SNAPSHOT' or 'CONSISTENCY'.
ENGINE_VERSION	The Firebird engine (server) version. Added in 2.1.

Return values and error behaviour. If the polled variable exists in the given namespace, its value will be returned as a string of max. 255 characters. If the namespace doesn't exist or if you try to access a non-existing variable in the SYSTEM namespace, an error is raised. If you poll a non-existing

variable in one of the other namespaces, NULL is returned. Both namespace and variable names must be given as single-quoted, case-sensitive, non-NULL strings.

Examples.

```
select rdb$get_context('SYSTEM', 'DB_NAME') from rdb$database

New.UserAddr = rdb$get_context('SYSTEM', 'CLIENT_ADDRESS');

insert into MyTable (TestField)
  values (rdb$get_context('USER_SESSION', 'MyVar'))
```

See also. RDB$SET_CONTEXT()

RDB$SET_CONTEXT()

Note

RDB$SET_CONTEXT and its counterpart RDB$GET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present – the user doesn't have to do anything to make them available.

Available in. DSQL, ESQL, PSQL

Added in. 2.0

Description. Creates, sets or unsets a variable in one of the user-writable namespaces USER_SESSION and USER_TRANSACTION.

Result type. INTEGER

Syntax.

```
RDB$SET_CONTEXT ('<namespace>', '<varname>', <value> | NULL)

  <namespace>  ::=  USER_SESSION | USER_TRANSACTION
  <varname>    ::=  A case-sensitive string of max. 80 characters
  <value>      ::=  A value of any type, as long as it's castable
                    to a VARCHAR(255)
```

The namespaces. The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The USER_SESSION context is bound to the current connection. Variables in USER_TRANSACTION only exist in the transaction in which they have been set. When the transaction ends, the context and all the variables defined in it are destroyed.

Return values and error behaviour. The function returns 1 if the variable already existed before the call and 0 if it didn't. To remove a variable from a context, set it to NULL. If the given namespace doesn't exist, an error is raised. Both namespace and variable names must be entered as single-quoted, case-sensitive, non-NULL strings.

Examples.

```
select rdb$set_context('USER_SESSION', 'MyVar', 493) from rdb$database

rdb$set_context('USER_SESSION', 'RecordsFound', RecCounter);

select rdb$set_context('USER_TRANSACTION', 'Savepoints', 'Yes')
  from rdb$database
```

Notes.

- The maximum number of variables in any single context is 1000.
- All USER_TRANSACTION variables will survive a ROLLBACK RETAIN or ROLLBACK TO SAVEPOINT unaltered, no matter at which point during the transaction they were set.
- Due to its UDF-like nature, RDB$SET_CONTEXT can – in PSQL only – be called like a void function, without assigning the result, as in the second example above. Regular internal functions don't allow this type of use.

See also. RDB$GET_CONTEXT()

REPLACE()

Available in. DSQL, PSQL

Added in. 2.1

Description. Replaces all occurrences of a substring in a string.

Result type. VARCHAR or BLOB

Syntax.

```
REPLACE (str, find, repl)
```

- This function fully supports text BLOBs of any length and character set.
- If any argument is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* calculated from the lengths of *str*, *find* and *repl* in such a way that even the maximum possible number of replacements won't overflow the field.
- If *find* is the empty string, *str* is returned unchanged.
- If *repl* is the empty string, all occurrences of *find* are deleted from *str*.
- If any argument is NULL, the result is always NULL, even if nothing would have been replaced.

Examples.

```
replace ('Billy Wilder',  'il', 'oog')     -- returns 'Boogly Woogder'
  replace ('Billy Wilder',  'il',    '')     -- returns 'Bly Wder'
  replace ('Billy Wilder',  null, 'oog')     -- returns NULL
  replace ('Billy Wilder',  'il',  null)     -- returns NULL
  replace ('Billy Wilder', 'xyz',  null)     -- returns NULL (!)
  replace ('Billy Wilder', 'xyz', 'abc')     -- returns 'Billy Wilder'
  replace ('Billy Wilder',    '', 'abc')     -- returns 'Billy Wilder'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

See also. OVERLAY()

REVERSE()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns a string backwards.

Result type. VARCHAR

Syntax.

```
REVERSE (str)
```

Examples.

```
reverse ('spoonful')                 -- returns 'lufnoops'
  reverse ('Was it a cat I saw?')      -- returns '?was I tac a ti saW'
```

Tip

This function comes in very handy if you want to group, search or order on string endings, e.g. when dealing with domain names or email addresses:

```
create index ix_people_email on people
  computed by (reverse(email));

  select * from people
  where reverse(email) starting with reverse('.br');
```

RIGHT()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the rightmost part of the argument string. The number of characters is given in the second argument.

Result type. VARCHAR or BLOB

Syntax.

```
RIGHT (string, length)
```

- This function supports text BLOBs of any length, but has a bug in versions 2.1–2.1.3 and 2.5 that makes it fail with text BLOBs larger than 1024 bytes that have a multi-byte character set. This has been fixed in versions 2.1.4 and 2.5.1.
- If *string* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the length of the input string.
- If the *length* argument exceeds the string length, the input string is returned unchanged.
- If the *length* argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

Important

If the external function `RIGHT` is declared in your database as `right` instead of the default `sright`, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also. LEFT()

ROUND()

Available in. DSQL, PSQL

Added in. 2.1

Description. Rounds a number to the nearest integer. If the fractional part is exactly `0.5`, rounding is upward for positive numbers and downward for negative numbers. With the optional *`scale`* argument, the number can be rounded to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

Result type. INTEGER, (scaled) BIGINT or DOUBLE

Syntax.

```
ROUND (<number> [, <scale>])

  <number>  ::=  a numerical expression
  <scale>   ::=  an integer specifying the number of decimal places
                 toward which should be rounded, e.g.:
                    2 for rounding to the nearest multiple of 0.01
                    1 for rounding to the nearest multiple of 0.1
                    0 for rounding to the nearest whole number
                   -1 for rounding to the nearest multiple of 10
                   -2 for rounding to the nearest multiple of 100
```

Notes.

- If the *`scale`* argument is present, the result usually has the same scale as the first argument, e.g.
  - ROUND(123.654, 1) returns 123.700 (not 123.7)
  - ROUND(8341.7, -3) returns 8000.0 (not 8000)
  - ROUND(45.1212, 0) returns 45.0000 (not 45)

  Otherwise, the result scale is 0:
  - ROUND(45.1212) returns 45

Important

- If the external function `ROUND` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).
- If you are used to the behaviour of the external function `ROUND`, please notice that the *internal* function always rounds halves away from zero, i.e. downward for negative numbers.

RPAD()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5 (backported to 2.1.4)

Description. Right-pads a string with spaces or with a user-supplied string until a given length is reached.

Result type. VARCHAR or BLOB

Syntax.

```
RPAD (str, endlen [, padstr])
```

- This function fully supports text BLOBs of any length and character set.
- If *str* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*endlen*).
- If *padstr* is given and equals `''` (empty string), no padding takes place.
- If *endlen* is less than the current string length, the string is truncated to *endlen*, even if *padstr* is the empty string.

Important

If the external function RPAD is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Note

In Firebird 2.1–2.1.3, all non-BLOB results were of type VARCHAR(32765), which made it advisable to cast them to a more modest size. This is no longer the case.

Examples.

```
rpad ('Hello', 12)                  -- returns 'Hello       '
  rpad ('Hello', 12, '-')             -- returns 'Hello-------'
  rpad ('Hello', 12, '')              -- returns 'Hello'
  rpad ('Hello', 12, 'abc')           -- returns 'Helloabcabca'
  rpad ('Hello', 12, 'abcdefghij')    -- returns 'Helloabcdefg'
  rpad ('Hello', 2)                   -- returns 'He'
  rpad ('Hello', 2, '-')              -- returns 'He'
  rpad ('Hello', 2, '')               -- returns 'He'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

See also. LPAD()

SIGN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the sign of the argument: -1, 0 or 1.

Result type. SMALLINT

Syntax.

```
SIGN (number)
```

Important

If the external function `SIGN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SIN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns an angle's sine. The argument must be given in radians.

Result type. DOUBLE PRECISION

Syntax.

```
SIN (angle)
```

- Any non-`NULL` result is – obviously – in the range [-1, 1].

Important

If the external function `SIN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SINH()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the hyperbolic sine of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
SINH (number)
```

Important

If the external function `SINH` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SQRT()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the square root of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
SQRT (number)
```

Important

If the external function SQRT is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SUBSTRING()

Available in. DSQL, PSQL

Added in. 1.0

Changed in. 2.0, 2.1, 2.1.5, 2.5.1

Description. Returns a string's substring starting at the given position, either to the end of the string or with a given length.

Result type. VARCHAR(*n*) or BLOB

Syntax.

```
SUBSTRING (str FROM startpos [FOR length])
```

This function returns the substring starting at character position *startpos* (the first position being 1). Without the FOR argument, it returns all the remaining characters in the string. With FOR, it returns *length* characters or the remainder of the string, whichever is shorter.

In Firebird 1.x, *startpos* and *length* must be integer literals. In 2.0 and above they can be any valid integer expression.

Starting with Firebird 2.1, this function fully supports binary and text BLOBs of any length and character set. If *str* is a BLOB, the result is also a BLOB. For any other argument type, the result is a VARCHAR(*n*). Previously, the result type used to be CHAR(*n*) if the argument was a CHAR(*n*) or a string literal.

For non-BLOB arguments, the width of the result field is always equal to the length of *str*, regardless of *startpos* and *length*. So, `substring('pinhead' from 4 for 2)` will return a VARCHAR(7) containing the string `'he'`.

If any argument is NULL, the result is NULL.

Bugs

- If *str* is a BLOB and the *length* argument is not present, the output is limited to 32767 characters. Workaround: with long BLOBs, always specify char_length(*str*) – or a sufficiently high integer – as the third argument, unless you are sure that the requested substring fits within 32767 characters.

  This bug has been fixed in version 2.5.1; the fix was also backported to 2.1.5.

- A bug in Firebird 2.0 which caused the function to return “false emptystrings” if *startpos* or *length* was NULL, has been fixed.

Example.

```
insert into AbbrNames(AbbrName)
  select substring(LongName from 1 for 3) from LongNames
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

TAN()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns an angle's tangent. The argument must be given in radians.

Result type. DOUBLE PRECISION

Syntax.

```
TAN (angle)
```

Important

If the external function `TAN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

TANH()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the hyperbolic tangent of the argument.

Result type. DOUBLE PRECISION

Syntax.

```
TANH (number)
```

- Due to rounding, any non-`NULL` result is in the range [-1, 1] (mathematically, it's <-1, 1>).

Important

If the external function `TANH` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

TRIM()

Available in. DSQL, PSQL

Added in. 2.0

Changed in. 2.1

Description. Removes leading and/or trailing spaces (or optionally other strings) from the input string. Since Firebird 2.1 this function fully supports text BLOBs of any length and character set.

Result type. VARCHAR(*n*) or BLOB

Syntax.

```
TRIM ([<adjust>] str)

  <adjust>  ::=  {[where] [what]} FROM

  where     ::=  BOTH | LEADING | TRAILING       /* default is BOTH */

  what      ::=  The substring to be removed (repeatedly if necessary)
                 from str's head and/or tail. Default is ' ' (space).
```

Examples.

```
select trim ('  Waste no space   ') from rdb$database
  -- returns 'Waste no space'

select trim (leading from '  Waste no space   ') from rdb$database
  -- returns 'Waste no space   '

select trim (leading '.' from '  Waste no space   ') from rdb$database
  -- returns '  Waste no space   '

select trim (trailing '!' from 'Help!!!!') from rdb$database
  -- returns 'Help'

select trim ('la' from 'lalala I love you Ella') from rdb$database
  -- returns ' I love you El'

select trim ('la' from 'Lalala I love you Ella') from rdb$database
  -- returns 'Lalala I love you El'
```

Notes.

- If `str` is a BLOB, the result is a BLOB. Otherwise, it is a VARCHAR(*n*) with *n* the formal length of `str`.
- The substring to be removed, if specified, may not be bigger than 32767 bytes. However, if this substring is *repeated* at `str`'s head or tail, the total number of bytes removed may be far greater. (The restriction on the size of the substring will be lifted in Firebird 3.)

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

TRUNC()

Available in. DSQL, PSQL

Added in. 2.1

Description. Returns the integer part of a number. With the optional `scale` argument, the number can be truncated to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

Result type. INTEGER, (scaled) BIGINT or DOUBLE

Syntax.

```
TRUNC (<number> [, <scale>])

  <number>  ::=  a numerical expression
  <scale>   ::=  an integer specifying the number of decimal places
                 toward which should be truncated, e.g.:
                    2 for truncating to a multiple of 0.01
                    1 for truncating to a multiple of 0.1
                    0 for truncating to a whole number
                   -1 for truncating to a multiple of 10
                   -2 for truncating to a multiple of 100
```

Notes.

- If the *scale* argument is present, the result usually has the same scale as the first argument, e.g.
  - TRUNC(789.2225, 2) returns 789.2200 (not 789.22)
  - TRUNC(345.4, -2) returns 300.0 (not 300)
  - TRUNC(-163.41, 0) returns -163.00 (not -163)

  Otherwise, the result scale is 0:
  - TRUNC(-163.41) returns -163

Important

If you are used to the behaviour of the external function TRUNCATE, please notice that the *internal* function TRUNC always truncates toward zero, i.e. upward for negative numbers.

UPPER()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in. 2.0, 2.1

Description. Returns the upper-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are uppercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

Result type. (VAR)CHAR or BLOB

Syntax.

```
UPPER (str)
```

Examples.

```
select upper(_iso8859_1 'Débâcle')
  from rdb$database
  -- returns 'DÉBÂCLE' (before Firebird 2.0: 'DéBâCLE')

select upper(_iso8859_1 'Débâcle' collate fr_fr)
  from rdb$database
  -- returns 'DEBACLE', following French uppercasing rules
```

See also. LOWER

UUID_TO_CHAR()

Available in. DSQL, PSQL

Added in. 2.5

Description. Converts a 16-byte UUID to its 36-character, human-readable ASCII representation.

Result type. CHAR(36)

Syntax.

```
UUID_TO_CHAR (uuid)

  uuid  ::=  a string consisting of 16 single-byte characters
```

Examples.

```
select uuid_to_char(x'876C45F4569B320DBCB4735AC3509E5F') from rdb$databas
  -- returns '876C45F4-569B-320D-BCB4-735AC3509E5F'

  select uuid_to_char(gen_uuid()) from rdb$database
  -- returns e.g. '680D946B-45FF-DB4E-B103-BB5711529B86'

  select uuid_to_char('Firebird swings!') from rdb$database
  -- returns '46697265-6269-7264-2073-77696E677321'
```

See also. CHAR_TO_UUID(), GEN_UUID()

Aggregate functions

Aggregate functions operate on groups of records, rather than on individual records or variables. They are often used in combination with a GROUP BY clause.

AVG()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in.

Description. AVG returns the average argument value in the group.

Result type. Integer

Syntax.

```
AVG (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.

COUNT()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in.

Description. COUNT returns the number of non-null values in the group.

Result type. Integer

Syntax.

```
COUNT (expression)
```

- If the group is empty or contains only NULLs, the result is 0.

LIST()

Available in. DSQL, PSQL

Added in. 2.1

Changed in. 2.5

Description. LIST returns a string consisting of the non-`NULL` argument values in the group, separated either by a comma or by a user-supplied delimiter. If there are no non-`NULL` values (this includes the case where the group is empty), `NULL` is returned.

Result type. BLOB

Syntax.

```
LIST ([ALL | DISTINCT] expression [, separator])
```

- ALL (the default) results in all non-`NULL` values to be listed. With DISTINCT, duplicates are removed, except if *`expression`* is a BLOB.
- In Firebird 2.5 and up, the optional *`separator`* argument may be any string expression. This makes it possible to specify e.g. *`ascii_char(13)`* as a separator. (This improvement has also been backported to 2.1.4.)
- The *`expression`* and *`separator`* arguments support BLOBs of any size and character set.
- Date/time and numerical arguments are implicitly converted to strings before concatenation.
- The result is a text BLOB, except when *`expression`* is a BLOB of another subtype.
- The ordering of the list values is undefined.

MAX()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in. 2.1

Description. MAX returns the maximum argument value in the group. If the argument is a string, this is the value that comes last when the active collation is applied.

Result type. Varies

Syntax.

```
MAX (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.
- Since Firebird 2.1, this function fully supports text BLOBs of any size and character set.

MIN()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in. 2.1

Description. MIN returns the minimum argument value in the group. If the argument is a string, this is the value that comes first when the active collation is applied.

Result type. Varies

Syntax.

```
MIN (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.
- Since Firebird 2.1, this function fully supports text BLOBs of any size and character set.

SUM()

Available in. DSQL, ESQL, PSQL

Added in. IB

Changed in.

Description. SUM calculates and returns the sum of non-null values in the group.

Result type. Integer

Syntax.

```
SUM (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.

Appendix A. Reserved words and keywords

Reserved words are part of the Firebird SQL language. They cannot be used as identifiers (e.g. as table or procedure names), except when enclosed in double quotes in Dialect 3. However, you should avoid this unless you have a compelling reason.

Keywords are also part of the language. They have a special meaning when used in the proper context, but they are not reserved for Firebird's own and exclusive use. You can use them as identifiers without double-quoting.

Reserved words

Full list of reserved words in Firebird 2.5:

ADD
ADMIN
ALL
ALTER
AND
ANY
AS
AT
AVG
BEGIN
BETWEEN
BIGINT
BIT_LENGTH
BLOB
BOTH
BY
CASE
CAST
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
COLLATE
COLUMN
COMMIT
CONNECT
CONSTRAINT
COUNT
CREATE
CROSS
CURRENT
CURRENT_CONNECTION
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSACTION
CURRENT_USER

CURSOR
DATE
DAY
DEC
DECIMAL
DECLARE
DEFAULT
DELETE
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
ESCAPE
EXECUTE
EXISTS
EXTERNAL
EXTRACT
FETCH
FILTER
FLOAT
FOR
FOREIGN
FROM
FULL
FUNCTION
GDSCODE
GLOBAL
GRANT
GROUP
HAVING
HOUR
IN
INDEX
INNER
INSENSITIVE
INSERT
INT
INTEGER
INTO
IS
JOIN
LEADING
LEFT
LIKE
LONG
LOWER
MAX
MAXIMUM_SEGMENT
MERGE
MIN
MINUTE
MONTH
NATIONAL
NATURAL
NCHAR
NO

NOT
NULL
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OR
ORDER
OUTER
PARAMETER
PLAN
POSITION
POST_EVENT
PRECISION
PRIMARY
PROCEDURE
RDB$DB_KEY
REAL
RECORD_VERSION
RECREATE
RECURSIVE
REFERENCES
RELEASE
RETURNING_VALUES
RETURNS
REVOKE
RIGHT
ROLLBACK
ROW_COUNT
ROWS
SAVEPOINT
SECOND
SELECT
SENSITIVE
SET
SIMILAR
SMALLINT
SOME
SQLCODE
SQLSTATE (2.5.1)
START
SUM
TABLE
THEN
TIME
TIMESTAMP
TO
TRAILING
TRIGGER
TRIM
UNION
UNIQUE
UPDATE
UPPER
USER
USING

VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VIEW
WHEN
WHERE
WHILE
WITH
YEAR

Keywords

The following terms have a special meaning in Firebird 2.5 DSQL. Some of them are also reserved words, others aren't.

!<
^<
^=
^>
,
:=
!=
!>
(
)
<
<=
<>
=
>
>=
||
~<
~=
~>
ABS
ACCENT
ACOS
ACTION
ACTIVE
ADD
ADMIN
AFTER
ALL
ALTER
ALWAYS
AND
ANY
AS
ASC
ASCENDING
ASCII_CHAR
ASCII_VAL
ASIN
AT
ATAN

ATAN2
AUTO
AUTONOMOUS
AVG
BACKUP
BEFORE
BEGIN
BETWEEN
BIGINT
BIN_AND
BIN_NOT
BIN_OR
BIN_SHL
BIN_SHR
BIN_XOR
BIT_LENGTH
BLOB
BLOCK
BOTH
BREAK
BY
CALLER
CASCADE
CASE
CAST
CEIL
CEILING
CHAR
CHAR_LENGTH
CHAR_TO_UUID
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMENT
COMMIT
COMMITTED
COMMON
COMPUTED
CONDITIONAL
CONNECT
CONSTRAINT
CONTAINING
COS
COSH
COT
COUNT
CREATE
CROSS
CSTRING
CURRENT
CURRENT_CONNECTION
CURRENT_DATE
CURRENT_ROLE

CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSACTION
CURRENT_USER
CURSOR
DATA
DATABASE
DATE
DATEADD
DATEDIFF
DAY
DEC
DECIMAL
DECLARE
DECODE
DEFAULT
DELETE
DELETING
DESC
DESCENDING
DESCRIPTOR
DIFFERENCE
DISCONNECT
DISTINCT
DO
DOMAIN
DOUBLE
DROP
ELSE
END
ENTRY_POINT
ESCAPE
EXCEPTION
EXECUTE
EXISTS
EXIT
EXP
EXTERNAL
EXTRACT
FETCH
FILE
FILTER
FIRST
FIRSTNAME
FLOAT
FLOOR
FOR
FOREIGN
FREE_IT
FROM
FULL
FUNCTION
GDSCODE
GEN_ID
GEN_UUID
GENERATED
GENERATOR
GLOBAL

GRANT
GRANTED
GROUP
HASH
HAVING
HOUR
IF
IGNORE
IIF
IN
INACTIVE
INDEX
INNER
INPUT_TYPE
INSENSITIVE
INSERT
INSERTING
INT
INTEGER
INTO
IS
ISOLATION
JOIN
KEY
LAST
LASTNAME
LEADING
LEAVE
LEFT
LENGTH
LEVEL
LIKE
LIMBO
LIST
LN
LOCK
LOG
LOG10
LONG
LOWER
LPAD
MANUAL
MAPPING
MATCHED
MATCHING
MAX
MAXIMUM_SEGMENT
MAXVALUE
MERGE
MIDDLENAME
MILLISECOND
MIN
MINUTE
MINVALUE
MOD
MODULE_NAME
MONTH
NAMES

NATIONAL
NATURAL
NCHAR
NEXT
NO
NOT
NULL
NULLIF
NULLS
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OS_NAME
OUTER
OUTPUT_TYPE
OVERFLOW
OVERLAY
PAD
PAGE
PAGE_SIZE
PAGES
PARAMETER
PASSWORD
PI
PLACING
PLAN
POSITION
POST_EVENT
POWER
PRECISION
PRESERVE
PRIMARY
PRIVILEGES
PROCEDURE
PROTECTED
RAND
RDB$DB_KEY
READ
REAL
RECORD_VERSION
RECREATE
RECURSIVE
REFERENCES
RELEASE
REPLACE
REQUESTS
RESERV
RESERVING
RESTART
RESTRICT
RETAIN
RETURNING

RETURNING_VALUES
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLLBACK
ROUND
ROW_COUNT
ROWS
RPAD
SAVEPOINT
SCALAR_ARRAY
SCHEMA
SECOND
SEGMENT
SELECT
SENSITIVE
SEQUENCE
SET
SHADOW
SHARED
SIGN
SIMILAR
SIN
SINGULAR
SINH
SIZE
SKIP
SMALLINT
SNAPSHOT
SOME
SORT
SOURCE
SPACE
SQLCODE
SQLSTATE (2.5.1)
SQRT
STABILITY
START
STARTING
STARTS
STATEMENT
STATISTICS
SUB_TYPE
SUBSTRING
SUM
SUSPEND
TABLE
TAN
TANH
TEMPORARY
THEN
TIME
TIMEOUT
TIMESTAMP
TO
TRAILING

TRANSACTION
TRIGGER
TRIM
TRUNC
TWO_PHASE
TYPE
UNCOMMITTED
UNDO
UNION
UNIQUE
UPDATE
UPDATING
UPPER
USER
USING
UUID_TO_CHAR
VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VIEW
WAIT
WEEK
WEEKDAY
WHEN
WHERE
WHILE
WITH
WORK
WRITE
YEAR
YEARDAY

Appendix B. Character sets and collations

To be written.

Appendix C. Error codes

To be written.

Appendix D. Document History

The exact file history is recorded in the `manual` module in our CVS tree; see http://sourceforge.net/cvs/?group_id=9028

Revision History		
Revision 1.0	?? Mon 2012	XX
First publication.		

Appendix E. License notice