Firebird 1.5 Language Reference Update

Everything new in Firebird SQL since InterBase 6

Paul Vinkenoog et al.

4 July 2008, document version 1.0 — covers Firebird 1.0–1.5.5

Firebird 1.5 Language Reference Update

Everything new in Firebird SQL since InterBase 6

4 July 2008, document version 1.0 — covers Firebird 1.0–1.5.5
Paul Vinkenoog et al.

Table of Contents

1. Introduction 1
    - Versions covered 1
    - Authorship 2
2. Reserved words 3
    - Added in 1.0 but removed in 1.5 3
    - Added in 1.0 and 1.5 3
    - To be added in future versions 4
3. Miscellaneous language elements 5
    - -- (single-line comment) 5
    - CASE construct 5
        - Simple CASE 6
        - Searched CASE 6
4. Data types and subtypes 8
    - BIGINT data type 8
    - New character sets 9
    - Character set NONE handling changed 10
    - New collations 10
5. DDL statements 11
    - ALTER TABLE 11
        - ALTER COLUMN: POSITION now 1-based 11
        - UNIQUE constraints now allow NULLs 11
        - USING INDEX subclause 12
    - ALTER TRIGGER 12
        - Multi-action triggers 12
        - ALTER TRIGGER no longer increments table change count 13
        - PLAN allowed in trigger code 13
    - CREATE DATABASE 13
        - 16 Kb page size supported 13
    - CREATE GENERATOR 14
        - Maximum number of generators significantly raised 14
    - CREATE INDEX 14
        - UNIQUE indices now allow NULLs 14
        - Maximum number of indices per table increased 14
    - CREATE TABLE 15
        - UNIQUE constraints now allow NULLs 15
        - USING INDEX subclause 16
    - CREATE TRIGGER 17
        - Multi-action triggers 17
        - CREATE TRIGGER no longer increments table change count 18
        - PLAN allowed in trigger code 18
    - CREATE VIEW 18
        - PLAN subclause disallowed 18
    - CREATE OR ALTER PROCEDURE 19
    - CREATE OR ALTER TRIGGER 19
    - DECLARE EXTERNAL FUNCTION 19
        - BY DESCRIPTOR parameter passing 20
        - RETURNS PARAMETER *n* 20
    - DROP GENERATOR 20

DROP TRIGGER 20
DROP TRIGGER no longer increments table change count 20
RECREATE PROCEDURE 21
RECREATE TABLE 21
RECREATE VIEW 21
6. DML statements 22
EXECUTE PROCEDURE 22
RELEASE SAVEPOINT 23
ROLLBACK TO SAVEPOINT 23
SAVEPOINT 24
Internal savepoints 25
Savepoints and PSQL 25
SELECT 26
Aggregate functions: Extended functionality 26
Ambiguous JOIN statements rejected 28
FIRST and SKIP 29
GROUP BY *UDF* 30
GROUP BY internal function, column position, and CASE 31
HAVING: Stricter rules 32
ORDER BY: Expressions and NULLs placement 32
ORDER BY: Stricter rules with aggregate statements 33
WITH LOCK 33
7. PSQL statements 35
BEGIN ... END blocks may be empty 35
BREAK 35
DECLARE [VARIABLE] with initialization 36
EXCEPTION 36
Rethrowing a caught exception 37
Providing a custom error message 37
EXECUTE PROCEDURE 37
EXECUTE STATEMENT 38
No data returned 38
One row of data returned 39
Any number of data rows returned 39
Caveats with EXECUTE STATEMENT 40
EXIT 40
FOR EXECUTE STATEMENT ... DO 40
LEAVE 41
PLAN allowed in trigger code 41
8. Context variables 42
CURRENT_CONNECTION 42
CURRENT_ROLE 42
CURRENT_TRANSACTION 43
CURRENT_USER 43
DELETING 44
GDSCODE 44
INSERTING 45
ROW_COUNT 45
SQLCODE 46
UPDATING 46
9. Internal functions 47
COALESCE() 47

EXTRACT() ... 47
NULLIF() ... 48
SUBSTRING() ... 49
10. External functions (UDFs) ... 50
addDay ... 50
addHour ... 50
addMilliSecond ... 51
addMinute ... 51
addMonth ... 52
addSecond ... 52
addWeek ... 52
addYear ... 53
ascii_char ... 53
dow ... 54
dpower ... 54
getExactTimestamp ... 55
i64round ... 55
i64truncate ... 55
log ... 55
lpad ... 56
ltrim ... 57
*nullif ... 57
*nvl ... 58
right ... 59
round, i64round ... 60
rpad ... 61
rtrim ... 61
sdow ... 62
string2blob ... 62
substr ... 63
substrlen ... 64
truncate, i64truncate ... 64
Appendix A: Notes ... 66
Character set NONE data accepted “as is” ... 66
Understanding the WITH LOCK clause ... 67
Syntax and behaviour ... 67
How the engine deals with WITH LOCK ... 68
The optional “OF *<column-names>*” sub-clause ... 69
Caveats using WITH LOCK ... 69
Examples using explicit locking ... 69
A note on CSTRING parameters ... 70
Appendix B: Document History ... 71
Appendix C: License notice ... 72

List of Tables

4.1. Character sets new in Firebird 9
4.2. Collations new in Firebird 10
5.1. Maximum indices per table 15
9.1. Ranges for EXTRACT results 48
A.1. How TPB settings affect explicit locking 68

Chapter 1

Introduction

This guide documents the **changes** made in Firebird 1.0 and 1.5 SQL since the fork from the open-sourced InterBase 6.0 codebase. It covers the following areas:

- Reserved words
- Data types and subtypes
- DDL statements (Data Definition Language)
- DML statements (Data Manipulation Language)
- PSQL statements (Procedural SQL, used in stored procedures and triggers)
- Context variables
- Internal functions
- UDFs (User Defined Functions, also known as external functions)

To have a complete Firebird 1.0 and 1.5 SQL reference, you need:

- The InterBase 6.0 beta SQL Reference (`LangRef.pdf` and/or `SQLRef.html`)
- This document

Topics **not** discussed in this document include:

- ODS versions
- Bug listings
- Installation and configuration
- Upgrade, migration and compatibility
- Server architectures
- API functions
- Connection protocols
- Tools and utilities

Consult the Release Notes for information on these subjects. You can find the Release Notes and other documentation via the Firebird Documentation Index at http://www.firebirdsql.org/index.php?op=doc.

Versions covered

This document covers all Firebird versions up to and including 1.5.5.

Authorship

Roughly 80–85% of the text in this document is new. The remainder was lifted from various Firebird Release Notes editions, which in turn contain material from preceding sources like the Whatsnew documents. Authors and editors of the included material are:

- J. Beesley
- Helen Borrie
- Arno Brinkman
- Alex Peshkov
- Nickolay Samofatov
- Dmitry Yemanov

Chapter 2

Reserved words

Reserved words are part of the Firebird SQL language. They cannot be used as identifiers, except when enclosed in double quotes. However, you should avoid this unless you really have no other option.

Added in 1.0 but removed in 1.5

The following reserved words were added in Firebird 1.0 but removed again in 1.5:

BREAK
DESCRIPTOR
FIRST
SKIP
SUBSTRING

The following non-reserved words were earmarked in Firebird 1.0 as "to be avoided because of future reservation", but no longer so in 1.5:

COALESCE
IIF
NULLIF

(of these three, COALESCE and NULLIF are non-reserved keywords in 1.5)

Added in 1.0 and 1.5

The following reserved words were added in Firebird 1.0 and are still reserved in 1.5:

CURRENT_ROLE
CURRENT_USER
RECREATE

The following reserved words were added in Firebird 1.5:

BIGINT
CASE
RELEASE
SAVEPOINT
CURRENT_CONNECTION
CURRENT_TRANSACTION
ROW_COUNT

The following words are not reserved, but recognized as keywords by Firebird 1.5 if used in the proper context:

COALESCE
DELETING
INSERTING
LAST
LEAVE
LOCK
NULLIF
NULLS
STATEMENT
UPDATING
USING

To be added in future versions

The following words are not reserved in Firebird 1.0 or 1.5, but should be avoided as identifiers because they will likely be reserved in future versions:

ABS
BOOLEAN
BOTH
CHAR_LENGTH
CHARACTER_LENGTH
FALSE
LEADING
OCTET_LENGTH
TRIM
TRAILING
TRUE
UNKNOWN

Chapter 3

Miscellaneous language elements

-- (single-line comment)

Available in: DSQL, PSQL

Added in: 1.0

Changed in: 1.5

Description: A line starting with “`--`” (two dashes) is a comment and will be ignored. This also makes it easy to quickly comment out a line of SQL.

In Firebird 1.5 and up, the “`--`” can be placed anywhere on the line, e.g. after an SQL statement. Everything from the double dash to the end of the line will be ignored.

Example:

```
-- a table to store our valued customers in:
create table Customers (
  name varchar(32),
  added_by varchar(24),
  custno varchar(8),
  purchases integer      -- number of purchases
)
```

Notice that the second comment is only allowed in Firebird 1.5 and up.

CASE construct

Available in: DSQL, ESQL, PSQL

Added in: 1.5

Description: A CASE construct returns exactly one value from a number of possibilities. There are two syntactic variants:

- The simple CASE, comparable to a Pascal `case` or a C `switch`.
- The searched CASE, which works like a series of “`if ... else if ... else if`” clauses.

Simple CASE

Syntax:

```
CASE <expression>
   WHEN <exp1> THEN result1
   WHEN <exp2> THEN result2
   ...
   [ELSE defaultresult]
END
```

When this variant is used, `<expression>` is compared to `<exp1>`, `<exp2>` etc., until a match is found, upon which the corresponding result is returned. If there is no match and there is an ELSE clause, `defaultresult` is returned. If there is no match and no ELSE clause, `NULL` is returned.

The match is determined with the "=" operator, so if `<expression>` is `NULL`, it won't match any of the `<expN>`s, not even those that are `NULL`.

The results don't have to be literal values: they may also be field or variable names, compound expressions, or `NULL` literals.

Example:

```
select name,
       age,
       case upper(sex)
         when 'M' then 'Male'
         when 'F' then 'Female'
         else 'Unknown'
       end,
       religion
from people
```

Searched CASE

Syntax:

```
CASE
   WHEN <bool_exp1> THEN result1
   WHEN <bool_exp2> THEN result2
   ...
   [ELSE defaultresult]
END
```

Here, the `<bool_expN>`s are tests that give a ternary boolean result: `true`, `false`, or `NULL`. The first expression evaluating to `TRUE` determines the result. If no expression is `TRUE` and there is an ELSE clause, `defaultresult` is returned. If no expression is `TRUE` and there is no ELSE clause, `NULL` is returned.

As with the simple CASE, the results don't have to be literal values: they may also be field or variable names, compound expressions, or `NULL` literals.

Example:

```
CanVote = case
            when Age >= 18 then 'Yes'
            when Age <  18 then 'No'
            else 'Unsure'
          end;
```

Chapter 4

Data types and subtypes

BIGINT data type

Added in: 1.5

Description: BIGINT is the SQL99-compliant 64-bit signed integer type. It is available in Dialect 3 only.

BIGINT numbers range from -2^{63} .. $2^{63}-1$, or -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807.

Example:

```
create table WholeLottaRecords (
  id bigint not null primary key,
  description varchar(32)
)
```

New character sets

Added in: 1.0, 1.5

The following table lists the character sets added in Firebird.

Table 4.1. Character sets new in Firebird

Name	Bytes/char.	Languages	Added in
DOS737	1	Greek	1.5
DOS775	1	Baltic	1.5
DOS858	1	= DOS850 plus € sign	1.5
DOS862	1	Hebrew	1.5
DOS864	1	Arabic	1.5
DOS866	1	Russian	1.5
DOS869	1	Modern Greek	1.5
ISO8859_2	1	Latin-2, Central European	1.0
ISO8859_3	1	Latin-3, Southern European	1.5
ISO8859_4	1	Latin-4, Northern European	1.5
ISO8859_5	1	Cyrillic	1.5
ISO8859_6	1	Arabic	1.5
ISO8859_7	1	Greek	1.5
ISO8859_8	1	Hebrew	1.5
ISO8859_9	1	Latin-5, Turkish	1.5
ISO8859_13	1	Latin-7, Baltic Rim	1.5
WIN1255	1	Hebrew	1.5
WIN1256	1	Arabic	1.5
WIN1257	1	Baltic	1.5

Character set NONE handling changed

Changed in: 1.5.1

Description: Firebird 1.5.1 has improved the way character set NONE data are moved to and from fields or variables with another character set, resulting in fewer transliteration errors. For more details, see the Note at the end of the book.

New collations

Added in: 1.0, 1.5, 1.5.1

The following table lists the collations added in Firebird.

Table 4.2. Collations new in Firebird

Character set	Collation	Language	Added in
ISO8859_2	CS_CZ	Czech	1.0
	ISO_HUN	Hungarian	1.5
ISO8859_13	LT_LT	Lithuanian	1.5.1
WIN1250	PXW_HUN	Hungarian	1.0
WIN1251	WIN1251_UA	Ukrainian and Russian	1.5

Chapter 5

DDL statements

ALTER TABLE

Available in: DSQL, ESQL

ALTER COLUMN: POSITION now 1-based

Changed in: 1.0

Description: When changing a column's position, the engine now interprets the new position as 1-based. This is in accordance with the SQL standard and the InterBase documentation, but in practice InterBase interpreted the position as 0-based.

Syntax:

```
ALTER TABLE tablename ALTER [COLUMN] colname POSITION <newpos>

<newpos>  ::=  an integer between 1 and the number of columns
```

Example:

```
alter table Stock alter Quantity position 3
```

Note

Don't confuse this with the POSITION in CREATE/ALTER TRIGGER. Trigger positions are and will remain 0-based.

UNIQUE constraints now allow NULLs

Changed in: 1.5

Description: In compliance with the SQL-99 standard, `NULL`s – even multiple – are now allowed in columns with a UNIQUE constraint. For a full discussion, see *CREATE TABLE :: UNIQUE constraints now allow `NULL`s*.

USING INDEX subclause

Added in: 1.5

Description: A USING INDEX subclause can be placed at the end of a primary, unique or foreign key definition. Its purpose is to

- provide a user-defined name for the automatically created index that enforces the constraint, and
- optionally define the index to be ascending or descending (the default being ascending).

Syntax:

```
[ADD] [CONSTRAINT constraint-name]
   <constraint-type> <constraint-definition>
   [USING [ASC[ENDING] | DESC[ENDING]] INDEX index_name]
```

For a full discussion and examples, see *CREATE TABLE :: USING INDEX subclause*.

ALTER TRIGGER

Available in: DSQL, ESQL

Multi-action triggers

Added in: 1.5

Description: The ALTER TRIGGER syntax has been extended to support multi-action triggers. For a full discussion of this feature, see *CREATE TRIGGER :: Multi-action triggers*.

Syntax:

```
ALTER TRIGGER trigger-name
   [ACTIVE | INACTIVE]
   {BEFORE | AFTER} <actions>
   [POSITION number]
   AS
   <trigger_body>

<actions>       ::= <single_action> [OR <single_action> [OR <single_action>]]
<single_action> ::= INSERT | UPDATE | DELETE
```

ALTER TRIGGER no longer increments table change count

Changed in: 1.0

Description: Each time you use CREATE, ALTER or DROP TRIGGER, InterBase increments the metadata change counter of the associated table. Once that counter reaches 255, no more metadata changes are possible on the table (you can still work with the data though). A backup-restore cycle is needed to reset the counter and perform metadata operations again.

While this obligatory cleanup after many metadata changes is in itself a useful feature, it also means that users who regularly use ALTER TRIGGER to deactivate triggers during e.g. bulk import operations are forced to backup and restore much more often then needed.

Since changes to triggers don't imply structural changes to the table itself, Firebird no longer increments the table change counter when CREATE, ALTER or DROP TRIGGER is used. One thing has remained though: once the counter is at 255, you can no longer create, alter or drop triggers for that table.

PLAN allowed in trigger code

Changed in: 1.5

Description: Before Firebird 1.5, a trigger containing a PLAN statement would be rejected by the compiler. Now a valid plan can be included and will be used.

CREATE DATABASE

Available in: DSQL, ESQL

16 Kb page size supported

Changed in: 1.0

Description: The maximum database page size has been raised from 8192 to 16384 bytes.

Syntax:

```
CREATE {DATABASE | SCHEMA}
   ...
   [PAGE_SIZE [=] <size>]
   ...

<size>  ::=  1024 | 2048 | 4096 | 8192 | 16384
```

CREATE GENERATOR

Available in: DSQL, ESQL

Maximum number of generators significantly raised

Changed in: 1.0

Description: InterBase reserved only one database page for generators, limiting the total number to 123 (on 1K pages) – 1019 (on 8K pages). Firebird has done away with that limit; you can now create more than 32,000 generators per database.

CREATE INDEX

Available in: DSQL, ESQL

UNIQUE indices now allow `NULL`s

Changed in: 1.5

Description: In compliance with the SQL-99 standard, `NULL`s – even multiple – are now allowed in columns that have a UNIQUE index defined on them. For a full discussion, see *CREATE TABLE :: UNIQUE constraints now allow `NULL`s*. As far as `NULL`s are concerned, the rules for unique indices are exactly the same as those for unique keys.

Maximum number of indices per table increased

Changed in: 1.0.3 and 1.5

Description: The maximum number of 64 indices per table has been removed in Firebird 1.0.3, and reintroduced at the higher level of 256 in Firebird 1.5.

Note

Probably due to an off-by-one error in the code, the effective ceiling is 65 indices in Firebird 1.0 and 1.0.2, and 257 indices in Firebird 1.5.

The number of indices attainable in practice is further limited by the database page size and the number of columns per index, as shown in the table below.

Table 5.1. Maximum indices per table

Page size	Firebird version(s)								
	1.0, 1.0.2			1.0.3			1.5.x		
	1 col	2 cols	3 cols	1 col	2 cols	3 cols	1 col	2 cols	3 cols
1024	62	50	41	62	50	41	62	50	41
2048	65	65	65	126	101	84	126	101	84
4096	65	65	65	254	203	169	254	203	169
8192	65	65	65	510	408	340	257	257	257
16384	65	65	65	1022	818	681	257	257	257

Please be aware that under normal circumstances, even 64 indices is way too many and will drastically reduce mutation speeds. The maximum was raised to accommodate data-warehousing applications and the like, that do lots of bulk operations during which indices are temporarily switched off.

CREATE TABLE

Available in: DSQL, ESQL

UNIQUE constraints now allow NULLs

Changed in: 1.5

Description: In compliance with the SQL-99 standard, NULLs – even multiple – are now allowed in columns with a UNIQUE constraint. It is therefore possible to define a UNIQUE key on a column that has no NOT NULL constraint.

For UNIQUE keys that span multiple columns, the logic is a little complicated:

- Multiple rows having *all* the UK columns NULL are allowed.
- Multiple rows having a *different subset* of UK colums NULL are allowed.
- Multiple rows having the *same subset* of UK columns NULL and the rest filled with regular values and those regular values *differ* in at least one column, are allowed.
- Multiple rows having the *same subset* of UK columns NULL and the rest filled with regular values and those regular values are the *same* in every column, are forbidden.

One way of summarizing this is as follows: In principle, all NULLs are considered distinct. But if two rows have exactly the same subset of UK columns filled with non-NULL values, the NULL columns are ignored and the non-NULL columns are decisive, just as if they constituted the entire unique key.

USING INDEX subclause

Added in: 1.5

Description: A USING INDEX subclause can be placed at the end of a primary, unique or foreign key definition. Its purpose is to

- provide a user-defined name for the automatically created index that enforces the constraint, and
- optionally define the index to be ascending or descending (the default being ascending).

Without USING INDEX, indices enforcing named constraints are named after the constraint (this is new behaviour in Firebird 1.5) and indices for unnamed constraints get names like RDB$FOREIGN13 or something equally romantic.

Note

You must always provide a *new* name for the index. It is not possible to use pre-existing indices to enforce constraints.

USING INDEX can be applied at field level, at table level, and (in ALTER TABLE) with ADD CONSTRAINT. It works with named as well as unnamed key constraints. It does *not* work with CHECK constraints, as these don't have their own enforcing index.

Syntax:

```
[CONSTRAINT constraint-name]
   <constraint-type> <constraint-definition>
   [USING [ASC[ENDING] | DESC[ENDING]] INDEX index_name]
```

Examples:

The first example creates a primary key constraint PK_CUST using an index named IX_CUSTNO:

```
create table customers (
  custno int not null constraint pk_cust primary key using index ix_custno,
  ...
```

This, however:

```
create table customers (
  custno int not null primary key using index ix_custno,
  ...
```

...will give you a PK constraint called INTEG_7 or something similar, and an index IX_CUSTNO.

Some more examples:

```
create table people (
  id int not null,
  nickname varchar(12) not null,
  country char(4),
  ..
  ..
  constraint pk_people primary key (id),
  constraint uk_nickname unique (nickname) using index ix_nick
)
```

```
alter table people
  add constraint fk_people_country
  foreign key (country) references countries(code)
  using desc index ix_people_country
```

Important

If you define a descending constraint-enforcing index on a primary or unique key, be sure to make any foreign keys referencing it descending as well.

CREATE TRIGGER

Available in: DSQL, ESQL

Multi-action triggers

Added in: 1.5

Description: Triggers can now be defined to fire upon multiple operations (INSERT and/or UPDATE and/or DELETE). Three new boolean context variables (`INSERTING`, `UPDATING` and `DELETING`) have been added so you can execute code conditionally within the trigger body depending on the type of operation.

Syntax:

```
CREATE TRIGGER trigger-name for table-name
   [ACTIVE | INACTIVE]
   {BEFORE | AFTER} <actions>
   [POSITION number]
   AS
   <trigger_body>

<actions>       ::= <single_action> [OR <single_action> [OR <single_action>]]
<single_action> ::= INSERT | UPDATE | DELETE
```

Example:

```
create trigger biu_parts for parts
  before insert or update
as
begin
  /* conditional code when inserting: */
  if (inserting and new.id is null)
    then new.id = gen_id(gen_partrec_id, 1);
  /* common code: */
  new.partname_upper = upper(new.partname);
end
```

> **Note**
>
> In multi-action triggers, both context variables OLD and NEW are always available. If you use them in the wrong situation (i.e. OLD while inserting or NEW while deleting), the following happens:
>
> - If you try to read their field values, `NULL` is returned.
> - If you try to assign values to them, a runtime exception is thrown.

CREATE TRIGGER no longer increments table change count

Changed in: 1.0

Description: In contrast to InterBase, Firebird does not increment the metadata change counter of the associated table when CREATE, ALTER or DROP TRIGGER is used. For a full discussion, see *ALTER TRIGGER no longer increments table change count*.

PLAN allowed in trigger code

Changed in: 1.5

Description: Before Firebird 1.5, a trigger containing a PLAN statement would be rejected by the compiler. Now a valid plan can be included and will be used.

CREATE VIEW

Available in: DSQL, ESQL

PLAN subclause disallowed

Changed in: 1.5

Description: You can no longer use a PLAN subclause in a view definition.

CREATE OR ALTER PROCEDURE

Available in: DSQL, ESQL

Added in: 1.5

Description: If the procedure does not yet exist, it is created just as if CREATE PROCEDURE were used. If it already exists, it is altered and recompiled. Existing permissions and dependencies are preserved.

Syntax: Exactly the same as for CREATE PROCEDURE.

CREATE OR ALTER TRIGGER

Available in: DSQL, ESQL

Added in: 1.5

Description: If the trigger does not yet exist, it is created just as if CREATE TRIGGER were used. If it already exists, it is altered and recompiled. Existing permissions and dependencies are preserved.

Syntax: Exactly the same as for CREATE TRIGGER.

DECLARE EXTERNAL FUNCTION

Available in: DSQL, ESQL

Description: This statement makes an external function (UDF) known to the database.

Syntax:

```
DECLARE EXTERNAL FUNCTION localname
   [<type_decl> [, <type_decl> ...]]
   RETURNS {<return_type_decl> | PARAMETER 1-based_pos} [FREE_IT]
   ENTRY_POINT 'function_name' MODULE_NAME 'library_name'

<type_decl>          ::=  sqltype [BY DESCRIPTOR] | CSTRING(length)
<return_type_decl>  ::=  sqltype [BY {DESCRIPTOR|VALUE}] | CSTRING(length)
```

You may choose `localname` freely; this is the name by which the function will be known to your database. You may also vary the `length` argument of CSTRING parameters (more about CSTRINGs in the note near the end of the book).

BY DESCRIPTOR parameter passing

Added in: 1.0

Description: Firebird introduces the possibility to pass parameters BY DESCRIPTOR; this mechanism facilitates the processing of `NULL`s in a meaningful way. Notice that this only works if the person who wrote the function has implemented it. Simply adding “BY DESCRIPTOR” to an existing declaration does not make it work – on the contrary! Always use the declaration block provided by the function designer.

RETURNS PARAMETER *n*

Added in: IB 6

Description: In order to return a BLOB, an extra input parameter must be declared and a “RETURNS PARAMETER *n*” subclause added – *n* being the position of said parameter. This subclause dates back to InterBase 6 beta, but somehow didn't make it into the *Language Reference* (it is documented in the *Developer's Guide* though).

DROP GENERATOR

Available in: DSQL, ESQL

Added in: 1.0

Description: Removes a generator. Its (very small) storage space will be freed for re-use after a backup-restore cycle.

Syntax:

```
DROP GENERATOR generator-name
```

DROP TRIGGER

Available in: DSQL, ESQL

DROP TRIGGER no longer increments table change count

Changed in: 1.0

Description: In contrast to InterBase, Firebird does not increment the metadata change counter of the associated table when CREATE, ALTER or DROP TRIGGER is used. For a full discussion, see *ALTER TRIGGER no longer increments table change count*.

RECREATE PROCEDURE

Available in: DSQL, ESQL

Added in: 1.0

Description: Creates or recreates a stored procedure. If a procedure with the same name already exists, RECREATE PROCEDURE will try to drop it and create a new procedure. RECREATE PROCEDURE will fail if the existing SP is in use.

Syntax: Exactly the same as CREATE PROCEDURE.

RECREATE TABLE

Available in: DSQL, ESQL

Added in: 1.0

Description: Creates or recreates a table. If a table with the same name already exists, RECREATE TABLE will try to drop it (destroying all its data in the process!) and create a new table. RECREATE TABLE will fail if the existing table is in use.

Syntax: Exactly the same as CREATE TABLE.

RECREATE VIEW

Available in: DSQL, ESQL

Added in: 1.5

Description: Creates or recreates a view. If a view with the same name already exists, RECREATE VIEW will try to drop it and create a new view. RECREATE VIEW will fail if the existing view is in use.

Syntax: Exactly the same as CREATE VIEW.

Chapter 6

DML statements

EXECUTE PROCEDURE

Available in: DSQL, ESQL, PSQL

Changed in: 1.5

Description: Executes a stored procedure. In Firebird 1.0.x as well as in InterBase, any input parameters for the SP must be supplied as literals, host language variables (in ESQL) or local variables (in PSQL). In Firebird 1.5 and above, input parameters may also be (compound) expressions, except in static ESQL.

Syntax:

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

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

Notes

- TRANSACTION clauses are not supported in PSQL.
- Expression parameters are not supported in static ESQL, and not in Firebird versions below 1.5.
- `NULL` indicators are only valid in ESQL code. They must be host language variables of type integer.
- 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, only possible in Firebird 1.5 and up:

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

RELEASE SAVEPOINT

Available in: DSQL

Added in: 1.5

Description: Deletes a named savepoint, freeing up all the resources it binds.

Syntax:

```
RELEASE SAVEPOINT name [ONLY]
```

Unless ONLY is added, all the savepoints created after the named savepoint are released as well.

For a full discussion of savepoints, see *SAVEPOINT*.

ROLLBACK TO SAVEPOINT

Available in: DSQL

Added in: 1.5

Description: Undoes everything that happened in a transaction since the creation of the savepoint.

Syntax:

```
ROLLBACK [WORK] TO [SAVEPOINT] name
```

ROLLBACK TO SAVEPOINT performs the following operations:

- All the mutations performed within the transaction since the savepoint was created are undone.
- All savepoints created after the one named are destroyed. All earlier savepoints are preserved, as is the savepoint itself. This means that you can rollback to the same savepoint several times.
- All implicit and explicit record locks acquired since the savepoint are released. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the unlocked rows immediately.

For a full discussion of savepoints, see *SAVEPOINT*.

SAVEPOINT

Available in: DSQL

Added in: 1.5

Description: Creates an SQL-99 compliant savepoint, to which you can later rollback your work without rolling back the entire transaction. Savepoint mechanisms are also known as “nested transactions”.

Syntax:

```
SAVEPOINT <name>

<name>  ::=  a user-chosen identifier, unique within the transaction
```

If the supplied name exists already within the same transaction, the existing savepoint is deleted and a new one is created with the same name.

If you later want to rollback your work to the point where the savepoint was created, use:

```
ROLLBACK [WORK] TO [SAVEPOINT] name
```

ROLLBACK TO SAVEPOINT performs the following operations:

- All the mutations performed within the transaction since the savepoint was created are undone.
- All savepoints created after the one named are destroyed. All earlier savepoints are preserved, as is the savepoint itself. This means that you can rollback to the same savepoint several times.
- All implicit and explicit record locks acquired since the savepoint are released. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the unlocked rows immediately.

The internal savepoint bookkeeping can consume huge amounts of memory, especially if you update the same records multiple times in one transaction. If you don't need a savepoint anymore but you're not yet ready to end the transaction, you can delete the savepoint and free the resources it uses with:

```
RELEASE SAVEPOINT name [ONLY]
```

With ONLY, the named savepoint is the only one that gets released. Without it, all savepoints created after it are released as well.

Example DSQL session using a savepoint:

```
create table test (id integer);
commit;
insert into test values (1);
commit;
insert into test values (2);
savepoint y;
delete from test;
select * from test;   -- returns no rows
rollback to y;
select * from test;   -- returns two rows
rollback;
select * from test;   -- returns one row
```

Internal savepoints

By default, the engine uses an automatic transaction-level system savepoint to perform transaction rollback. When you issue a ROLLBACK statement, all changes performed in this transaction are backed out via a transaction-level savepoint and the transaction is then committed. This logic reduces the amount of garbage collection caused by rolled back transactions.

When the volume of changes performed under a transaction-level savepoint is getting large (10^4–10^6 records affected), the engine releases the transaction-level savepoint and uses the TIP mechanism to roll back the transaction if needed.

Tip

If you expect the volume of changes in your transaction to be large, you can use the TPB flag `isc_tpb_no_auto_undo` to avoid the transaction-level savepoint being created.

Savepoints and PSQL

Transaction control statements are not allowed in PSQL, as that would break the atomicity of the statement that calls the procedure. But Firebird does support the raising and handling of exceptions in PSQL, so that actions performed in stored procedures and triggers can be selectively undone without the entire procedure failing. Internally, automatic savepoints are used to:

- undo all actions in a BEGIN...END block where an uncaught exception occurs;
- undo all actions performed by the SP/trigger (or, in the case of a selectable SP, all actions performed since the last SUSPEND) when it terminates prematurely due to an uncaught error or exception.

Each PSQL exception handling block is also bounded by automatic system savepoints.

SELECT

Available in: DSQL, ESQL, PSQL

Aggregate functions: Extended functionality

Changed in: 1.5

Description: Several types of mixing and nesting aggragate 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 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 built around a non-aggregate column that appears in the GROUP BY list (by name or position).
- The operand equals or is built around a subquery, whether or not it is also a GROUP BY item.

“Is built around” means that the operand need not be exactly the same as the column or subquery. 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 in the SELECT or GROUP BY list.

Ambiguous JOIN statements 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.

FIRST and SKIP

Added in: 1.0

Changed in: 1.5

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

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 MyTable)
  ```

  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!

- 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 )
  ```

GROUP BY *UDF*

Changed in: 1.0

Description: In Firebird, you can use the output of a user-defined function as a GROUP BY item.

Syntax:

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

<item>      ::=  column-name [COLLATE collation-name] | <udf-call>
<udf-call>  ::=  udf-name(arg1 [, argN ...])
```

UDF calls may be nested, but – as follows from the syntax – you cannot mix UDF calls and COLLATE in a single GROUP BY item.

Example:

```
select strlen(lastname), count(*)
  from people
  group by strlen(lastname)
  order by 2 desc
```

Warning

DSQL currently lacks a mechanism to check if GROUP BY *UDF* subclauses are formulated correctly. Always make sure that your GROUP BY item list correctly represents the *scalar* (i.e. non-aggregate) expression(s) in your SELECT list.

GROUP BY internal function, column position, and CASE

Changed in: 1.5

Description: Firebird 1.5 adds the following to the list of valid GROUP BY items:

- 1-based column position numbers (like in ORDER BY);
- The internal functions COALESCE, EXTRACT, NULLIF, SUBSTRING and UPPER;
- CASE constructs.

Syntax:

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

<item>           ::=  column-name [COLLATE collation-name]
                        | column-position
                        | <function-call>
                        | CASE-construct

<function-call>  ::=  COALESCE(arg1, arg2 [, argN ...])
                        | EXTRACT(part FROM date/time)
                        | NULLIF(arg1, arg2)
                        | SUBSTRING(str FROM pos [FOR count])
                        | UPPER(str)
                        | udf-name(arg1 [, argN ...])
```

Function calls may be nested. As in previous versions, COLLATE can only be used with column names.

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.

Important

- A GROUP BY item cannot be a reference to an aggregate function (including those that are buried inside an expression) from the same context.
- As before, every non-aggregate column **must** appear in the GROUP BY list, whether explicitly or by position.

Examples:

```
select
  case when price is null then 0 else price end,
  sum(number_sold)
from sales_per_article
group by
  case when price is null then 0 else price end
```

Of course this example is only to demonstrate the use of a CASE construct in the GROUP BY clause. In this particular case you should first use COALESCE:

```
select
  coalesce (price, 0),
  sum(number_sold)
from sales_per_article
group by
  coalesce (price, 0)
```

and second, you could save yourself some typing by using the column number:

```
select
  coalesce (price, 0),
  sum(number_sold)
from sales_per_article
group by 1
```

HAVING: Stricter rules

Changed in: 1.5

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

ORDER BY: Expressions and NULLs placement

Changed in: 1.5

Description: In addition to column names and positions, the ORDER BY clause can now also contain expressions to sort the output by. Furthermore, per-column NULLS FIRST and NULLS LAST subclauses can be used to specify where `NULL`s appear in the sorted column.

Syntax:

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

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

Expressions consisting of a single non-negative number will be interpreted as 1-based column numbers and will cause an exception if they're not in the range from 1 to the number of columns.

By default, `NULL`s will be placed at the end of the sort, regardless whether the order is ascending or descending. This is the same behaviour as in previous Firebird versions. No index will be used on columns for which the non-default NULLS FIRST placement is chosen.

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.

Examples:

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

```
select first 10 * from document
  order by strlen(description) desc
```

```
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
```

ORDER BY: Stricter rules with aggregate statements

Changed in: 1.5

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

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.

Chapter 7

PSQL statements

PSQL – Procedural SQL – is the Firebird stored procedure and trigger language.

BEGIN ... END blocks may be empty

Available in: PSQL

Changed in: 1.5

Description: BEGIN ... END blocks may be empty in Firebird 1.5 and up, allowing you to write stub code without having to resort to dummy statements.

Example:

```
create trigger bi_atable for atable
active before insert position 0
as
begin
end
```

BREAK

Available in: PSQL

Added in: 1.0

Deprecated in: 1.5

Description: BREAK immediately terminates a WHILE or FOR loop and continues with the first statement after the loop.

Example:

```
create procedure selphrase(num int)
returns (phrase varchar(40))
as
begin
  for select Phr from Phrases into phrase do
  begin
    if (num < 1) then break;
    suspend;
    num = num - 1;
  end
  phrase = '***  Ready!  ***';
  suspend;
end
```

This selectable SP returns at most *num* rows from the table Phrases. The variable *num* is decremented in each iteration; once it is smaller than 1, the loop is terminated with BREAK. The program then continues at the line “`phrase = '*** Ready! ***';`”.

Important

Since Firebird 1.5, BREAK is deprecated in favor of its SQL-99 compliant synonym *LEAVE*.

DECLARE [VARIABLE] with initialization

Available in: PSQL

Changed in: 1.5

Description: In Firebird 1.5 and above, a PSQL local variable can be initialized upon declaration. The VARIABLE keyword has become optional.

Syntax:

```
DECLARE [VARIABLE] varname datatype [{= | DEFAULT} value];
```

Example:

```
create procedure proccie (a int)
returns (b int)
as
declare p int;
declare q int = 8;
declare r int default 9;
declare variable s int;
declare variable t int = 10;
declare variable u int default 11;
begin
  <intelligent code here>
end
```

EXCEPTION

Available in: PSQL

Changed in: 1.5

Description: The EXCEPTION syntax has been extended so that the user can

a. Rethrow a caught exception or error.
b. Provide a custom message when throwing a user-defined exception.

Syntax:

```
EXCEPTION [<exception-name> [custom-message]]

<exception-name>  ::=  A previously defined exception name
```

Rethrowing a caught exception

Within the exception handling block only, you can rethrow the caught exception or error by giving the EXCEPTION command without any arguments. Outside such blocks, this “bare” command has no effect.

Example:

```
when any do
begin
  insert into error_log (...) values (sqlcode, ...);
  exception;
end
```

This example first logs some information about the exception or error, and then rethrows it.

Providing a custom error message

Firebird 1.5 and up allow you to override an exception's default error message by supplying an alternative one when throwing the exception.

Examples:

```
exception ex_data_error 'You just lost some valuable data';
```

```
exception ex_bad_type 'Wrong type for record with id ' || new.id;
```

EXECUTE PROCEDURE

Available in: DSQL, PSQL

Changed in: 1.5

Description: In Firebird 1.5 and above, (compound) expressions are allowed as input parameters for stored procedures called with EXECUTE PROCEDURE. See *DML statements :: EXECUTE PROCEDURE* for full info and examples.

EXECUTE STATEMENT

Available in: PSQL

Added in: 1.5

Description: EXECUTE STATEMENT takes a single string argument and executes it as if it had been submitted as a DSQL statement. The exact syntax depends on the number of data rows that the supplied statement may return.

No data returned

This form is used with INSERT, UPDATE, DELETE and EXECUTE PROCEDURE statements that return no data.

Syntax:

```
EXECUTE STATEMENT <statement>

<statement>  ::=  An SQL statement returning no data.
```

Example:

```
create procedure DynamicSampleOne (ProcName varchar(100))
as
declare variable stmt varchar(1024);
declare variable param int;
begin
   select min(SomeField) from SomeTable into param;
   stmt = 'execute procedure '
          || ProcName
          || '('
          || cast(param as varchar(20))
          || ')';
   execute statement stmt;
end
```

Warning

Although this form of EXECUTE STATEMENT can also be used with all kinds of DDL strings (except CREATE/DROP DATABASE), it is generally very, very unwise to use this trick in order to circumvent the no-DDL rule in PSQL.

One row of data returned

This form is used with singleton SELECT statements.

Syntax:

```
EXECUTE STATEMENT <select-statement> INTO <var> [, <var> ...]

<select-statement>  ::=  An SQL statement returning at most one row of data.
<var>                ::=  A PSQL variable, optionally preceded by “:”
```

Example:

```
create procedure DynamicSampleTwo (TableName varchar(100))
as
declare variable param int;
begin
  execute statement
    'select max(CheckField) from ' || TableName into :param;
  if (param > 100) then
    exception Ex_Overflow 'Overflow in ' || TableName;
end
```

Any number of data rows returned

This form – analogous to “FOR SELECT ... DO” – is used with SELECT statements that may return a multi-row dataset.

Syntax:

```
FOR EXECUTE STATEMENT <select-statement> INTO <var> [, <var> ...]
   DO <compound-statement>

<select-statement>  ::=  Any SELECT statement.
<var>                ::=  A PSQL variable, optionally preceded by “:”
```

Example:

```
create procedure DynamicSampleThree
  ( TextField varchar(100),
    TableName varchar(100) )
returns
  ( LongLine varchar(32000) )
as
declare variable Chunk varchar(100);
begin
  Chunk = '';
  for execute statement
    'select ' || TextField || ' from ' || TableName into :Chunk
  do
    if (Chunk is not null) then
      LongLine = LongLine || Chunk || ' ';
  suspend;
end
```

Caveats with EXECUTE STATEMENT

1. There is no way to validate the syntax of the enclosed statement.
2. There are no dependency checks to discover whether tables or columns have been dropped.
3. Operations will be slow because the embedded statement has to be prepared every time it is executed.
4. The argument string cannot contain any parameters. All variable substitution into the static part of the SQL statement should be performed before EXECUTE STATEMENT is called.
5. Return values are strictly checked for data type in order to avoid unpredictable type-casting exceptions. For example, the string `'1234'` would convert to an integer, 1234, but `'abc'` would give a conversion error.
6. If the stored procedure has special privileges on some objects, the dynamic statement submitted in the EXECUTE STATEMENT string does not inherit them. Privileges are restricted to those granted to the user who is executing the procedure.

All in all, this feature is intended only for very cautious use and you should always take the above factors into account. Bottom line: use EXECUTE STATEMENT only when other methods are impossible, or perform even worse than EXECUTE STATEMENT.

EXIT

Available in: PSQL

Changed in: 1.5

Description: In Firebird 1.5 and up, EXIT can be used in all PSQL. In earlier versions it is only supported in stored procedures, not in triggers.

FOR EXECUTE STATEMENT ... DO

Available in: PSQL

Added in: 1.5

Description: See *EXECUTE STATEMENT :: Any number of data rows returned*.

LEAVE

Available in: PSQL

Added in: 1.5

Description: LEAVE immediately terminates a WHILE or FOR loop and continues with the first statement after the loop.

Example:

```
while (b < 10) do
begin
  insert into Numbers(B) values (:b);
  b = b + 1;
  when any do
  begin
    execute procedure log_error (current_timestamp, 'Error in B loop');
    leave;
  end
end
c = 0;
while (c < 10) do
begin
  ...
  ...
end
```

If an error occurs during the insert, the event is logged and the loop terminated. The program continues at the line of code reading “`c = 0;`”

PLAN allowed in trigger code

Changed in: 1.5

Description: Before Firebird 1.5, a trigger containing a PLAN statement would be rejected by the compiler. Now a valid plan can be included and will be used.

Chapter 8

Context variables

CURRENT_CONNECTION

Available in: DSQL, PSQL

Added in: 1.5

Description: CURRENT_CONNECTION contains the system identifier of the active connection context.

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 upon restore. Since the engine itself is not interested in this value, it is only incremented if the client reads it during a session. Hence it is only useful as a unique identifier, not as an indicator of the number of connections since the creation or last restoration of the database.

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_TRANSACTION

Available in: DSQL, PSQL

Added in: 1.5

Description: CURRENT_TRANSACTION contains the system identifier of the current transaction context.

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 upon restore. Unlike CURRENT_CONNECTION, it is incremented with every new transaction, whether the client reads the value or not.

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

Description: In a WHEN GDSCODE handling block, the `GDSCODE` context variable contains a numerical representation of the current Firebird error code. It is 0 in WHEN SQLCODE, WHEN EXCEPTION and WHEN ANY handlers, as well as everywhere else in PSQL.

Type: INTEGER

Example:

```
when gdscode 335544551, gdscode 335544552,
     gdscode 335544553, gdscode 335544707
do
begin
  execute procedure log_grant_error(gdscode);
  exit;
end
```

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
```

ROW_COUNT

Available in: PSQL

Added in: 1.5

Description: `ROW_COUNT` is a context variable containing the number of rows affected by the last DML statement.

Type: INTEGER

Example:

```
update Figures set Number = 0 where id = :id;
if (row_count = 0) then
  insert into Figures (id, Number) values (:id, 0);
```

Notes

- For SELECT statements, `ROW_COUNT` currently returns 0.
- `ROW_COUNT` cannot be used to determine the number of rows affected by an EXECUTE STATEMENT command.

SQLCODE

Available in: PSQL

Added in: 1.5

Description: In a WHEN SQLCODE handling block, the `SQLCODE` context variable contains the current SQL error code. In a WHEN ANY block it contains the SQL error code if indeed an SQL error occurred; otherwise it contains 0. `SQLCODE` is also 0 in WHEN GDSCODE and WHEN EXCEPTION handlers, as well as everywhere else in PSQL.

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
```

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
```

Chapter 9

Internal functions

COALESCE()

Available in: DSQL, ESQL, 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`, `NULL` is returned.

Return 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.

EXTRACT()

Available in: DSQL, ESQL, PSQL

Added in: IB 6

Description: Extracts and returns an element from a DATE, TIME or TIMESTAMP expression. It was already added in InterBase 6, but not documented in the *Language Reference* at the time.

Return type: SMALLINT or DECIMAL(6,4)

Syntax:

```
EXTRACT (<part> FROM <datetime>)

<part>      ::=  YEAR | MONTH | DAY | WEEKDAY | YEARDAY
                   | HOUR | MINUTE | SECOND
<datetime>  ::=  An expression of type DATE, TIME or TIMESTAMP
```

The returned datatype is DECIMAL(6,4) for the SECOND part and SMALLINT for all others. The 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 9.1. Ranges for EXTRACT results

Part	Range	Comment
YEAR	1–9999	
MONTH	1–12	
DAY	1–31	
WEEKDAY	0–6	0 = Sunday
YEARDAY	0–365	0 = January 1
HOUR	0–23	
MINUTE	0–59	
SECOND	0.0000–59.999	

NULLIF()

Available in: DSQL, ESQL, 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.

Return 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.

SUBSTRING()

Available in: DSQL, ESQL, PSQL

Added in: 1.0

Description: Returns a string's substring starting at the given position, either to the end of the string or with a given length.

Return type: CHAR(*n*)

Syntax:

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

<str> := any expression evaluating to a string
startpos and length must be integer literals
```

SUBSTRING returns the stream of bytes starting at byte position *`startpos`* (the first byte position being 1). Without the FOR argument, it returns all the remaining bytes in the string. With FOR, it returns *`length`* bytes or the remainder of the string, whichever is shorter.

SUBSTRING can be used with:

- Any string or (var)char argument, regardless of its character set;
- Subtype 0 (binary) BLOBs;
- Subtype 1 (text) BLOBs, if the character set has 1 byte per character.

SUBSTRING can *not* be used with text BLOBs that have an underlying multi-byte character set.

Example:

```
insert into AbbrNames(AbbrName)
  select substring(LongName from 1 for 3) from LongNames
```

Chapter 10

External functions (UDFs)

External functions must be "declared" (made known) to the database before they can be used. Firebird ships with two external function libraries:

- `ib_udf` – inherited from InterBase;
- `fbudf` – a new library using descriptors, present as from Firebird 1.0 (Windows) and 1.5 (Linux).

Users can also create their own UDF libraries or acquire them from third parties.

addDay

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *`number`* days added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
adddday (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addDay
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addDay' MODULE_NAME 'fbudf'
```

addHour

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *`number`* hours added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addhour (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addHour
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addHour' MODULE_NAME 'fbudf'
```

addMilliSecond

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *number* milliseconds added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addmillisecond (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addMilliSecond
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addMilliSecond' MODULE_NAME 'fbudf'
```

addMinute

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *number* minutes added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addminute (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addMinute
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addMinute' MODULE_NAME 'fbudf'
```

addMonth

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *number* months added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addmonth (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addMonth
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addMonth' MODULE_NAME 'fbudf'
```

addSecond

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with *number* seconds added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addsecond (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addSecond
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addSecond' MODULE_NAME 'fbudf'
```

addWeek

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with `number` weeks added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addweek (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addWeek
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addWeek' MODULE_NAME 'fbudf'
```

addYear

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the first argument with `number` years added. Use negative numbers to subtract.

Return type: TIMESTAMP

Syntax:

```
addyear (atimestamp, number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION addYear
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addYear' MODULE_NAME 'fbudf'
```

ascii_char

Library: ib_udf

Changed in: 1.0

Description: Returns the ASCII character corresponding to the integer value passed in.

Return type: CHAR(1)

Syntax (unchanged):

```
ascii_char (intval)
```

Declaration (changed):

```
DECLARE EXTERNAL FUNCTION ascii_char
   INTEGER
   RETURNS CSTRING(1) FREE_IT
   ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf'
```

The declaration has been changed to reflect the fact that the UDF as such returns a 1-character C string, not an SQL CHAR(1) as stated in the InterBase declaration. The engine will pass it on to the caller as a CHAR(1) though.

dow

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the day of the week from a timestamp argument. The returned name may be localized.

Return type: VARCHAR(15)

Syntax:

```
dow (atimestamp)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION dow
   TIMESTAMP,
   VARCHAR(15) RETURNS PARAMETER 2
   ENTRY_POINT 'DOW' MODULE_NAME 'fbudf'
```

See also: sdow

dpower

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns *x* to the *y*'th power.

Return type: DOUBLE PRECISION

Syntax:

```
dpower (x, y)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION dPower
   DOUBLE PRECISION BY DESCRIPTOR, DOUBLE PRECISION BY DESCRIPTOR,
   DOUBLE PRECISION BY DESCRIPTOR
   RETURNS PARAMETER 3
   ENTRY_POINT 'power' MODULE_NAME 'fbudf'
```

getExactTimestamp

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the system time with milliseconds precision. This function was added because CURRENT_TIMESTAMP always has `.0000` in the fractional part of the second.

Return type: TIMESTAMP

Syntax:

```
getexacttimestamp()
```

Declaration:

```
DECLARE EXTERNAL FUNCTION getExactTimestamp
   TIMESTAMP RETURNS PARAMETER 1
   ENTRY_POINT 'getExactTimestamp' MODULE_NAME 'fbudf'
```

i64round

See `round`.

i64truncate

See `truncate`.

log

Library: ib_udf

Changed in: 1.5

Description: In Firebird 1.5 and up, `log` returns the the base-*x* logarithm of *y*. In Firebird 1.0.x and InterBase, it erroneously returns the base-*y* logarithm of *x*.

Return type: DOUBLE PRECISION

Syntax (unchanged):

```
log (x, y)
```

Declaration (unchanged):

```
DECLARE EXTERNAL FUNCTION log
   DOUBLE PRECISION, DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_log' MODULE_NAME 'ib_udf'
```

Warning

If any of your pre-1.5 databases uses `log`, check your PSQL and application code. It may contain workarounds to return the right results. Under Firebird 1.5 and up, any such workarounds should be removed or you'll get the wrong results.

lpad

Library: ib_udf

Added in: 1.5

Changed in: 1.5.2

Description: Returns the input string left-padded with *padchar*s until *endlength* is reached.

Return type: CHAR(*n*)

Syntax:

```
lpad (str, endlength, padchar)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION lpad
   CSTRING(255), INTEGER, CSTRING(1)
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- When calling this function, make sure *endlength* does not exceed the declared result length.

ltrim

Library: ib_udf

Changed in: 1.5, 1.5.2

Description: Returns the input string with any leading space characters removed. In Firebird 1.0.x, this function returns `NULL` if the input string is empty or `NULL`. In 1.5 and above it returns `''` (an empty string) in these cases.

Return type: CHAR(*n*)

Syntax (unchanged):

```
ltrim (str)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION ltrim
   CSTRING(255)
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_ltrim' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.

*nullif

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Deprecated in: 1.5

Description: The four `*nullif` functions – for integers, bigints, doubles and strings, respectively – each return the first argument if it is not equal to the second. If the arguments are equal, the functions return `NULL`.

Return type: Varies, see declarations.

Syntax:

```
inullif   (int1, int2)
i64nullif (bigint1, bigint2)
dnullif   (double1, double2)
snullif   (string1, string2)
```

As from Firebird 1.5 these functions are all deprecated. Use the new internal function NULLIF instead.

Warnings

- These functions return `NULL` when the second argument is `NULL`, even if the first argument is a proper value. This is a wrong result. The NULLIF internal function doesn't have this bug.
- `i64nullif` and `dnullif` will return wrong and/or bizarre results if it is not 100% clear to the engine that each argument is of the intended type (NUMERIC(18,0) or DOUBLE PRECISION). If in doubt, cast them both explicitly to the declared type (see declarations below).

Declarations:

```
DECLARE EXTERNAL FUNCTION inullif
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS INT BY DESCRIPTOR
   ENTRY_POINT 'iNullIf' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64nullif
   NUMERIC(18,4) BY DESCRIPTOR, NUMERIC(18,4) BY DESCRIPTOR
   RETURNS NUMERIC(18,4) BY DESCRIPTOR
   ENTRY_POINT 'iNullIf' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION dnullif
   DOUBLE PRECISION BY DESCRIPTOR, DOUBLE PRECISION BY DESCRIPTOR
   RETURNS DOUBLE PRECISION BY DESCRIPTOR
   ENTRY_POINT 'dNullIf' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION snullif
   VARCHAR(100) BY DESCRIPTOR, VARCHAR(100) BY DESCRIPTOR,
   VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
   ENTRY_POINT 'sNullIf' MODULE_NAME 'fbudf'
```

*nvl

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Deprecated in: 1.5

Description: The four `nvl` functions – for integers, bigints, doubles and strings, respectively – are `NULL` replacers. They each return the first argument's value if it is not `NULL`. If the first argument is `NULL`, the value of the second argument is returned.

Return type: Varies, see declarations.

Syntax:

```
invl   (int1, int2)
i64nvl (bigint1, bigint2)
dnvl   (double1, double2)
snvl   (string1, string2)
```

As from Firebird 1.5 these functions are all deprecated. Use the new internal function COALESCE instead.

> **Warning**
>
> `i64nvl` and `dnvl` will return wrong and/or bizarre results if it is not absolutely clear to the engine that each argument is of the intended type (NUMERIC(18,0) or DOUBLE PRECISION). If in doubt, cast both arguments explicitly to the declared type (see declarations below).

Declarations:

```
DECLARE EXTERNAL FUNCTION invl
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS INT BY DESCRIPTOR
   ENTRY_POINT 'idNvl' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64nvl
   NUMERIC(18,0) BY DESCRIPTOR, NUMERIC(18,0) BY DESCRIPTOR
   RETURNS NUMERIC(18,0) BY DESCRIPTOR
   ENTRY_POINT 'idNvl' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION dnvl
   DOUBLE PRECISION BY DESCRIPTOR, DOUBLE PRECISION BY DESCRIPTOR
   RETURNS DOUBLE PRECISION BY DESCRIPTOR
   ENTRY_POINT 'idNvl' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION snvl
   VARCHAR(100) BY DESCRIPTOR, VARCHAR(100) BY DESCRIPTOR,
   VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
   ENTRY_POINT 'sNvl' MODULE_NAME 'fbudf'
```

right

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the rightmost *`numchars`* characters of the input string.

Return type: VARCHAR(100)

Syntax:

```
right (str, numchars)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION right
   VARCHAR(100) BY DESCRIPTOR, SMALLINT,
   VARCHAR(100) BY DESCRIPTOR RETURNS PARAMETER 3
   ENTRY_POINT 'right' MODULE_NAME 'fbudf'
```

round, i64round

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Changed in: 1.5

Description: These functions return the whole number that is nearest to their (scaled numeric/decimal) argument. They do not work with floats or doubles.

Return type: INTEGER / NUMERIC(18,4)

Syntax:

```
round    (number)
i64round (bignumber)
```

Bug warning

These functions are *broken* for negative numbers:

- Anything between 0 and -0.6 (that's right: -0.6, not -0.5) is rounded to 0.
- Anything between -0.6 and -1 is rounded to +1 (*plus* 1).
- Anything between -1 and -1.6 is rounded to -1.
- Anything between -1.6 and -2 is rounded to -2.
- Etcetera.

Declarations:

In Firebird 1.0.x, the entry point for both functions is `round`:

```
DECLARE EXTERNAL FUNCTION Round
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'round' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64Round
   NUMERIC(18,4) BY DESCRIPTOR, NUMERIC(18,4) BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'round' MODULE_NAME 'fbudf'
```

In Firebird 1.5, the entry point has been renamed to `fbround`:

```
DECLARE EXTERNAL FUNCTION Round
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'fbround' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64Round
   NUMERIC(18,4) BY DESCRIPTOR, NUMERIC(18,4) BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'fbround' MODULE_NAME 'fbudf'
```

If you move an existing database from Firebird 1.0.x to 1.5 or higher, drop any existing `*round` and `*truncate` declarations and declare them anew, using the updated entry point names.

rpad

Library: ib_udf

Added in: 1.5

Changed in: 1.5.2

Description: Returns the input string right-padded with *`padchar`*s until *`endlength`* is reached.

Return type: CHAR(*n*)

Syntax:

```
rpad (str, endlength, padchar)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION rpad
   CSTRING(255), INTEGER, CSTRING(1)
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_rpad' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- When calling this function, make sure *`endlength`* does not exceed the declared result length.

rtrim

Library: ib_udf

Changed in: 1.5, 1.5.2

Description: Returns the input string with any trailing space characters removed. In Firebird 1.0.x, this function returns `NULL` if the input string is empty or `NULL`. In 1.5 and above it returns `''` (an empty string) in these cases.

Return type: CHAR(*n*)

Syntax (unchanged):

```
rtrim (str)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION rtrim
   CSTRING(255)
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_rtrim' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.

sdow

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the abbreviated day of the week from a timestamp argument. The returned abbreviation may be localized.

Return type: VARCHAR(5)

Syntax:

```
sdow (atimestamp)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION sdow
   TIMESTAMP,
   VARCHAR(5) RETURNS PARAMETER 2
   ENTRY_POINT 'SDOW' MODULE_NAME 'fbudf'
```

See also: `dow`

string2blob

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Description: Returns the input string as a BLOB.

Return type: BLOB

Syntax:

```
string2blob (str)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION string2blob
   VARCHAR(300) BY DESCRIPTOR,
   BLOB RETURNS PARAMETER 2
   ENTRY_POINT 'string2blob' MODULE_NAME 'fbudf'
```

substr

Library: ib_udf

Changed in: 1.0, 1.5.2

Description: Returns a string's substring from *`startpos`* to *`endpos`*, inclusively. Positions are 1-based. If *`endpos`* is past the end of the string, Firebird's `substr` returns all the characters from *`startpos`* to the end of the string. InterBase's `substr` returned `NULL` in this case.

Return type: CHAR(*n*)

Syntax (unchanged):

```
substr (str, startpos, endpos)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION substr
   CSTRING(255), SMALLINT, SMALLINT
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_substr' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.

Tip

Although the function arguments are slightly different, consider using the internal SQL function SUBSTRING instead, for better compatibility.

substrlen

Library: ib_udf

Added in: 1.0

Changed in: 1.5.2

Description: Returns the substring starting at *startpos* and having *length* characters (or less, if the end of the string is reached first). Positions are 1-based. If either *startpos* or *length* is smaller than 1, an empty string is returned.

Return type: CHAR(*n*)

Syntax:

```
substrlen (str, startpos, length)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION substrlen
   CSTRING(255), SMALLINT, SMALLINT
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_substrlen' MODULE_NAME 'ib_udf'
```

Notes

- In Firebird 1.5.1 and below, the default declaration uses CSTRING(80) instead of CSTRING(255).
- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.

Tip

Firebird 1.0 has also implemented the internal SQL function SUBSTRING, rendering `substrlen` obsolete in the same version in which it was introduced. In new code, use SUBSTRING.

truncate, i64truncate

Library: fbudf

Added in: 1.0 (Win), 1.5 (Linux)

Changed in: 1.5

Description: These functions return the whole-number portion of their (scaled numeric/decimal) argument. They do not work with floats or doubles.

Return type: INTEGER / NUMERIC(18)

Syntax:

```
truncate    (number)
i64truncate (bignumber)
```

Warning

Both functions round to the nearest whole number that is lower than or equal to the argument. This means that negative numbers are “truncated” downward. For instance, `truncate(-2.37)` returns `-3`. A rather peculiar exception is formed by the numbers between -1 and 0, which are all truncated to 0. The only number that truncates to -1 is -1 itself.

Declarations:

In Firebird 1.0.x, the entry point for both functions is `truncate`:

```
DECLARE EXTERNAL FUNCTION Truncate
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'truncate' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64Truncate
   NUMERIC(18) BY DESCRIPTOR, NUMERIC(18) BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'truncate' MODULE_NAME 'fbudf'
```

In Firebird 1.5, the entry point has been renamed to `fbtruncate`:

```
DECLARE EXTERNAL FUNCTION Truncate
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'fbtruncate' MODULE_NAME 'fbudf'
```

```
DECLARE EXTERNAL FUNCTION i64Truncate
   NUMERIC(18) BY DESCRIPTOR, NUMERIC(18) BY DESCRIPTOR
   RETURNS PARAMETER 2
   ENTRY_POINT 'fbtruncate' MODULE_NAME 'fbudf'
```

If you move an existing database from Firebird 1.0.x to 1.5 or higher, drop any existing `*round` and `*truncate` declarations and declare them anew, using the updated entry point names.

Appendix A: Notes

Character set NONE data accepted "as is"

In Firebird 1.5.1 and up

Firebird 1.5.1 has improved the way character set NONE data are moved to and from fields or variables with another character set, resulting in fewer transliteration errors.

In Firebird 1.5.0, from a client connected with character set NONE, you could read data in two incompatible character sets – such as SJIS (Japanese) and WIN1251 (Russian) – even though you could not read one of those character sets while connected from a client with the other character set. Data would be received "as is" and be stored without raising an exception.

However, from this character set NONE client connection, an attempt to update any Russian or Japanese data columns using either parameterized queries or literal strings without introducer syntax would fail with transliteration errors; and subsequent queries on the stored "NONE" data would similarly fail.

In Firebird 1.5.1, both problems have been circumvented. Data received from the client in character set NONE are still stored "as is" but what is stored is an exact, binary copy of the received string. In the reverse case, when stored data are read into this client from columns with specific character sets, there will be no transliteration error. When the connection character set is NONE, no attempt is made in either case to resolve the string to well-formed characters, so neither the write nor the read will throw a transliteration error.

This opens the possibility for working with data from multiple character sets in a single database, as long as the connection character set is NONE. The client has full responsibility for submitting strings in the appropriate character set and converting strings returned by the engine, as needed.

Abstraction layers that have to manage this can read the low byte of the *`sqlsubtype`* field in the XSQLVAR structure, which contains the character set identifier.

While character set NONE literals are accepted and implicitly stored in the character set of their context, the use of introducer syntax to coerce the character sets of literals is highly recommended when the application is handling literals in a mixture of character sets. This should avoid the string's being misinterpreted when the application shifts the context for literal usage to a different character set.

Note

Coercion of the character set, using the introducer syntax or casting, is still required when handling heterogeneous character sets from a client context that is anything other than NONE. Both methods are shown below, using character set ISO8859_1 as an example target. Notice the "_" prefix in the introducer syntax.

Introducer syntax:

```
_ISO8859_1 mystring
```

Casting:

```
CAST (mystring AS VARCHAR(n) CHARACTER SET ISO8859_1)
```

Understanding the WITH LOCK clause

This note looks a little deeper into explicit locking and its ramifications. The WITH LOCK feature, added in Firebird 1.5, 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.

Pessimistic locks are rarely needed in Firebird. This is an expert feature, intended for use by those who thoroughly understand its consequences. Knowledge of the various levels of transaction isolation is essential. WITH LOCK is available in DSQL and PSQL, and only for top-level, single-table SELECTs. As stated in the reference part of this guide, WITH LOCK 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.

Syntax and behaviour

```
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.

As the engine considers, in turn, each record falling under an explicit lock statement, it returns either the record version that is the most currently committed, regardless of database state when the statement was submitted, or an exception.

Wait behaviour and conflict reporting depend on the transaction parameters specified in the TPB block:

Table A.1. How TPB settings affect explicit locking

TPB mode	Behaviour
isc_tpb_consistency	Explicit locks are overridden by implicit or explicit table-level locks and are ignored
isc_tpb_concurrency + isc_tpb_nowait	If a record is modified by any transaction that was committed since the transaction attempting to get explicit lock started, or an active transaction has performed a modification of this record, an update conflict exception is raised immediately
isc_tpb_concurrency + isc_tpb_wait	If the record is modified by any transaction that has committed since the transaction attempting to get explicit lock started, an update conflict exception is raised immediately. If an active transaction is holding ownership on this record (via explicit locking or by a normal optimistic write-lock) the transaction attempting the explicit lock waits for the outcome of the blocking transaction and, when it finishes, attempts to get the lock on the record again. This means that, if the blocking transaction committed a modified version of this record, an update conflict exception will be raised.
isc_tpb_read_committed + isc_tpb_nowait	If there is an active transaction holding ownership on this record (via explicit locking or normal update), an update conflict exception is raised immediately.
isc_tpb_read_committed + isc_tpb_wait	If there is an active transaction holding ownership on this record (via explicit locking or by a normal optimistic write-lock), the transaction attempting the explicit lock waits for the outcome of blocking transation and when it finishes, attempts to get the lock on the record again. Update conflict exceptions can never be raised by an explicit lock statement in this TPB mode.

How the engine deals with WITH LOCK

When an UPDATE statement tries to access a record that is locked by another transaction, it either raises an update conflict exception or waits for the locking transaction to finish, depending on TPB mode. Engine behaviour here is the same as if this record had already been modified by the locking transaction.

No special gdscodes are returned from conflicts involving pessimistic locks.

The engine guarantees that all records returned by an explicit lock statement are actually locked and *do* meet the search conditions specified in WHERE clause, as long as the search conditions do not depend on any other tables, via joins, subqueries, etc. It also guarantees that rows not meeting the search conditions will not be locked by the statement. It can *not* guarantee that there are no rows which, though meeting the search conditions, are not locked.

Note

This situation can arise if other, parallel transactions commit their changes during the course of the locking statement's execution.

The engine locks rows at fetch time. This has important consequences if you lock several rows at once. Many access methods for Firebird databases default to fetching output in packets of a few hundred rows ("buffered fetches"). Most data access components cannot bring you the rows contained in the last-fetched packet, where an error occurred.

The optional "OF `<column-names>`" sub-clause

The FOR UPDATE clause provides a technique to prevent usage of buffered fetches, optionally with the "OF *<column-names>*" subclause to enable positioned updates.

> **Tip**
>
> Alternatively, it may be possible in your access components to set the size of the fetch buffer to 1. This would enable you to process the currently-locked row before the next is fetched and locked, or to handle errors without rolling back your transaction.

Caveats using WITH LOCK

- Rolling back of an implicit or explicit savepoint releases record locks that were taken under that savepoint, but it doesn't notify waiting transactions. Applications should not depend on this behaviour as it may get changed in the future.
- While explicit locks can be used to prevent and/or handle unusual update conflict errors, the volume of deadlock errors will grow unless you design your locking strategy carefully and control it rigorously.
- Most applications do not need explicit locks at all. The main purposes of explicit locks are (1) to prevent expensive handling of update conflict errors in heavily loaded applications and (2) to maintain integrity of objects mapped to a relational database in a clustered environment. If your use of explicit locking doesn't fall in one of these two categories, then it's the wrong way to do the task in Firebird.
- Explicit locking is an advanced feature; do not misuse it! While solutions for these kinds of problems may be very important for web sites handling thousands of concurrent writers, or for ERP/CRM systems operating in large corporations, most application programs do not need to work in such conditions.

Examples using explicit locking

i. Simple:

```
SELECT * FROM DOCUMENT WHERE ID=? WITH LOCK
```

ii. Multiple rows, one-by-one processing with DSQL cursor:

```
SELECT * FROM DOCUMENT WHERE PARENT_ID=?
   FOR UPDATE WITH LOCK
```

A note on CSTRING parameters

External functions involving strings often use the type CSTRING(*n*) in their declarations. This type represents a zero-terminated string of maximum length *n*. Most of the functions handling CSTRINGs are programmed in such a way that they can accept and return zero-terminated strings of any length. So why the *n*? Because the Firebird engine has to set up space to process the input an output parameters, and convert them to and from SQL data types. Most strings used in databases are only dozens to hundreds of bytes long; it would be a waste to reserve 32 KB of memory each time such a string is processed. Therefore, the *standard* declarations of most CSTRING functions – as found in the file `ib_udf.sql` – specifiy a length of 255 bytes. (In Firebird 1.5.1 and below, this default length is 80 bytes.) As an example, here's the SQL declaration of `lpad`:

```
DECLARE EXTERNAL FUNCTION lpad
   CSTRING(255), INTEGER, CSTRING(1)
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf'
```

Once you've declared a CSTRING parameter with a certain length, you cannot call the function with a longer input string, or cause it to return a string longer than the declared output length. But the standard declarations are just reasonable defaults; they're not cast in concrete, and you can change them if you want to. If you have to left-pad strings of up to 500 bytes long, then it's perfectly OK to change both 255's in the declaration to 500 or more.

A special case is when you usually operate on short strings (say less then 100 bytes) but occasionally have to call the function with a huge (VAR)CHAR argument. Declaring CSTRING(32000) makes sure that all the calls will be successful, but it will also cause 32000 bytes per parameter to be reserved, even in that majority of cases where the strings are under 100 bytes. In that situation you may consider declaring the function twice, with different names and different string lengths:

```
DECLARE EXTERNAL FUNCTION lpad
   CSTRING(100), INTEGER, CSTRING(1)
   RETURNS CSTRING(100) FREE_IT
   ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION lpadbig
   CSTRING(32000), INTEGER, CSTRING(1)
   RETURNS CSTRING(32000) FREE_IT
   ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf';
```

Now you can call `lpad()` for all the small strings and `lpadbig()` for the occasional monster. Notice how the declared names in the first line differ (they determine how you call the functions from within your SQL), but the entry point (the function name in the library) is the same in both cases.

Appendix B: 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

1.0	4 Jul 2008	PV	First publication, using 15–20% material from the Firebird Release Notes.

Appendix C: License notice