Firebird 2.1 Language Reference Update

Everything new in Firebird SQL since InterBase 6

Paul Vinkenoog et al.

9 December 2010, document version 1.0 — covers Firebird 2.1–2.1.4

Firebird 2.1 Language Reference Update

Everything new in Firebird SQL since InterBase 6

9 December 2010, document version 1.0 — covers Firebird 2.1–2.1.4
Paul Vinkenoog et al.

Table of Contents

List of Tables

Chapter 1

Introduction

Subject matter

What's this book about?

This guide documents the **changes** made in the Firebird SQL language between InterBase 6 and Firebird 2.1.*x*. It covers the following areas:

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

To have a complete Firebird 2.1 SQL reference, you need:

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

Non-SQL topics are **not** discussed in this document. These 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 2.1.4.

Authorship

Most of this document was written by the main author. The remainder (3–4%) 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
- Frank Ingermann
- Vlad Khorsun
- Alex Peshkov
- Nickolay Samofatov
- Adriano dos Santos Fernandes
- Dmitry Yemanov

Chapter 2

Reserved words and keywords

Reserved words are part of the Firebird SQL language. They cannot be used as identifiers (e.g. as table or procedure names), except when enclosed in double quotes in Dialect 3. However, you should avoid this unless you have a compelling reason.

Keywords are also part of the language. They have a special meaning when used in the proper context, but they are not reserved for Firebird's own and exclusive use. You can use them as identifiers without double-quoting.

Added since InterBase 6

Newly reserved words

The following reserved words have been added to Firebird:

BIGINT
BIT_LENGTH
BOTH
CASE
CHAR_LENGTH
CHARACTER_LENGTH
CLOSE
CONNECT
CROSS
CURRENT_CONNECTION
CURRENT_ROLE
CURRENT_TRANSACTION
CURRENT_USER
DISCONNECT
FETCH
GLOBAL
INSENSITIVE
LEADING
LOWER
OCTET_LENGTH
OPEN
RECREATE
RECURSIVE
RELEASE
ROW_COUNT
ROWS
SAVEPOINT

SENSITIVE
START
TRAILING
TRIM
USING

New keywords

The following words have been added to Firebird as non-reserved keywords. Most of them are names of internal functions added between 2.0 and 2.1.

ABS
ACCENT
ACOS
ALWAYS
ASCII_CHAR
ASCII_VAL
ASIN
ATAN
ATAN2
BACKUP
BIN_AND
BIN_OR
BIN_SHL
BIN_SHR
BIN_XOR
BLOCK
CEIL
CEILING
COALESCE
COLLATION
COMMENT
COS
COSH
COT
DATEADD
DATEDIFF
DECODE
DELETING
DIFFERENCE
EXP
FLOOR
GEN_UUID
GENERATED
HASH
IIF
INSERTING
LAST
LEAVE
LIST
LN

LOCK
LOG
LOG10
LPAD
MATCHED
MATCHING
MAXVALUE
MILLISECOND
MINVALUE
MOD
NEXT
NULLIF
NULLS
OVERLAY
PAD
PI
PLACING
POWER
PRESERVE
RAND
REPLACE
RESTART
RETURNING
REVERSE
ROUND
RPAD
SCALAR_ARRAY
SEQUENCE
SIGN
SIN
SINH
SPACE
SQRT
STATEMENT
TAN
TANH
TEMPORARY
TRUNC
WEEK
UPDATING

Dropped since InterBase 6

No longer reserved

The following words are no longer reserved in Firebird 2.1, but are still recognized as keywords:

ACTION
CASCADE
FREE_IT
RESTRICT
ROLE
TYPE
WEEKDAY
YEARDAY

No longer keywords

The following are no longer keywords in Firebird 2.1:

BASENAME
CACHE
CHECK_POINT_LEN
GROUP_COMMIT_WAIT
LOG_BUF_SIZE
LOGFILE
NUM_LOG_BUFS
RAW_PARTITIONS

Possibly reserved in future versions

The following words are not reserved in Firebird 2.1, but are better avoided as identifiers because they will likely be reserved – or added as keywords – in future versions:

BOOLEAN
FALSE
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.

Shorthand casts

Available in: DSQL, ESQL, PSQL

Added in: IB

Description: When converting a string literal to a DATE, TIME or TIMESTAMP, Firebird allows the use of a shorthand "C-style" cast. This feature already existed in InterBase 6, but was never properly documented.

Syntax:

```
datatype 'date/timestring'
```

Examples:

```
update People set AgeCat = 'Old'
  where BirthDate < date '1-Jan-1943'
```

```
insert into Appointments
  (Employee_Id, Client_Id, App_date, App_time)
values
  (973, 8804, date 'today' + 2, time '16:00')
```

```
new.lastmod = timestamp 'now';
```

See also: CAST

CASE construct

Available in: DSQL, 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 <test-expr>
   WHEN <expr> THEN result
   [WHEN <expr> THEN result ...]
   [ELSE defaultresult]
END
```

When this variant is used, *`<test-expr>`* is compared to *`<expr>`* 1, *`<expr>`* 2 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 *`<test-expr>`* is `NULL`, it won't match any of the *`<expr>`*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.

A shorthand form of the simple CASE construct is the DECODE() function, available since Firebird 2.1.

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_expr> THEN result
   [WHEN <bool_expr> THEN result ...]
   [ELSE defaultresult]
END
```

Here, the *<bool_expr>*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)
)
```

BLOB data type

Text BLOB support in functions and operators

Changed in: 2.1

Description: Text BLOBs of any length and character set (including multi-byte sets) are now supported by practically every internal text function and operator. In a few cases there are limitations or bugs.

Level of support:

- Full support for:
  - = (assignment);
  - =, <>, <, <=, >, >= and synonyms (comparison);
  - || (concatenation);
  - BETWEEN, IS [NOT] DISTINCT FROM, IN, ANY|SOME and ALL.

- Partial support for STARTING [WITH], LIKE and CONTAINING: an error is raised if the second argument is 32 KB or longer.
- SELECT DISTINCT, ORDER BY and GROUP BY work on the BLOB ID, not the contents. This makes them as good as useless, except that SELECT DISTINCT weeds out multiple `NULL`s, if present. GROUP BY behaves oddly in that it groups together equal rows if they are adjacent, but not if they are apart.
- Any issues with BLOBs in internal functions and aggregate functions are discussed in their respective sections.

Various enhancements

Changed in: 2.0

Description: In Firebird 2.0, several enhancements have been implemented for text BLOBs:

- DML COLLATE clauses are now supported.
- Equality comparisons can be performed on the full BLOB contents.
- Character set conversions are possible when assigning a BLOB to a BLOB or a string to a BLOB.

When defining binary BLOBs, the mnemonic `binary` can now be used instead of the integer 0.

Examples:

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

```
create table MyPictures (
  id int not null primary key,
  title varchar(40),
  description varchar(200),
  picture blob sub_type binary
)
```

New character sets

Added in: 1.0, 1.5, 2.0, 2.1

The following table lists the character sets added in Firebird.

Table 4.1. Character sets new in Firebird

Name	Max bytes/ch.	Languages	Added in
CP943C	2	Japanese	2.1
DOS737	1	Greek	1.5
DOS775	1	Baltic	1.5
DOS858	1	= DOS850 plus € sign	1.5

Name	Max bytes/ch.	Languages	Added in
DOS862	1	Hebrew	1.5
DOS864	1	Arabic	1.5
DOS866	1	Russian	1.5
DOS869	1	Modern Greek	1.5
GBK	2	Chinese	2.1
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
KOI8R	1	Russian	2.0
KOI8U	1	Ukrainian	2.0
TIS620	1	Thai	2.1
UTF8 (*)	4	All	2.0
WIN1255	1	Hebrew	1.5
WIN1256	1	Arabic	1.5
WIN1257	1	Baltic	1.5
WIN1258	1	Vietnamese	2.0

(*)In Firebird 1.5, UTF8 is an alias for UNICODE_FSS. This character set has some inherent problems. In Firebird 2, UTF8 is a character set in its own right, without the drawbacks of UNICODE_FSS.

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, 2.0, 2.1

The following table lists the collations added in Firebird. The “Details” column is based on what has been reported in the Release Notes and other documents. The information in this column is probably incomplete; some collations with an empty Details field may still be case insensitive (ci), accent insensitive (ai) or dictionary-sorted (dic).

Please note that the default – binary – collations for new character sets are not listed here, as doing so would add no meaningful information.

Table 4.2. Collations new in Firebird

Character set	Collation	Language	Details	Added in
CP943C	CP943C_UNICODE	Japanese		2.1
GBK	GBK_UNICODE	Chinese		2.1
ISO8859_1	ES_ES_CI_AI	Spanish	ci, ai	2.0
	FR_FR_CI_AI	French	ci, ai	2.1
	PT_BR	Brazilian Portuguese	ci, ai	2.0
ISO8859_2	CS_CZ	Czech		1.0
	ISO_HUN	Hungarian		1.5
	ISO_PLK	Polish		2.0
ISO8859_13	LT_LT	Lithuanian		1.5.1
UTF8	UCS_BASIC	All		2.0
	UNICODE	All	dic	2.0
	UNICODE_CI	All	ci	2.1
WIN1250	BS_BA	Bosnian		2.0
	PXW_HUN	Hungarian	ci	1.0
	WIN_CZ	Czech	ci	2.0
	WIN_CZ_CI_AI	Czech	ci, ai	2.0
WIN1251	WIN1251_UA	Ukrainian and Russian		1.5
WIN1252	WIN_PTBR	Brazilian Portuguese	ci, ai	2.0
WIN1257	WIN1257_EE	Estonian	dic	2.0
	WIN1257_LT	Lithuanian	dic	2.0

Character set	Collation	Language	Details	Added in
	WIN1257_LV	Latvian	dic	2.0
KOI8R	KOI8R_RU	Russian	dic	2.0
KOI8U	KOI8U_UA	Ukrainian	dic	2.0
TIS620	TIS620_UNICODE	Thai		2.1

A note on the UTF8 collations

The UCS_BASIC collation sorts in Unicode code-point order: A, B, a, b, á... This is exactly the same as UTF8 with no collation specified. UCS_BASIC was added to comply with the SQL standard.

The UNICODE collation sorts using UCA (Unicode Collation Algorithm): a, A, á, b, B...

UNICODE_CI is truly case-insensitive. In a search for e.g. 'Apple', it will also find 'apple', 'APPLE' and 'aPPLe'.

Unicode collations for all character sets

Added in: 2.1

Firebird now comes with UNICODE collations for all the standard character sets. However, except for the ones listed in the new collations table in the previous section, these collations are not automatically available in your databases. Instead, they must be added with the CREATE COLLATION statement, like this:

```
create collation ISO8859_1_UNICODE for ISO8859_1
```

The new Unicode collations all have the name of their character set with _UNICODE added. (The built-in Unicode collations for UTF8 are the exception to the rule.) They are defined, along with the other collations, in the manifest file `fbintl.conf` in Firebird's `intl` subdirectory.

Collations may also be registered under a user-chosen name, e.g.:

```
create collation LAT_UNI for ISO8859_1 from external ('ISO8859_1_UNICODE')
```

See CREATE COLLATION for the full syntax.

Chapter 5

DDL statements

The statements in this chapter are grouped by the type of database object they operate on. For instance, ALTER DATABASE, CREATE DATABASE and DROP DATABASE are all found under *DATABASE*; DECLARE EXTERNAL FUNCTION and ALTER EXTERNAL FUNCTION are under *EXTERNAL FUNCTION*; etc.

GRANT and REVOKE, which can operate on a variety of object types, are together under *Privileges*.

COLLATION

CREATE COLLATION

Available in: DSQL

Added in: 2.1

Description: Adds a collation to the database. The collation must already be present on your system (typically in a library file) and must be properly registered in a `.conf` file in the `intl` subdirectory of your Firebird installation. You may also base the collation on one that is already present in the database.

Syntax:

```
CREATE COLLATION collname
   FOR charset
   [FROM basecoll | FROM EXTERNAL ('extname')]
   [NO PAD | PAD SPACE]
   [CASE [IN]SENSITIVE]
   [ACCENT [IN]SENSITIVE]
   ['<specific-attributes>']

collname                ::=  the name to use for the new collation
charset                 ::=  a character set present in the database
basecoll                ::=  a collation already present in the database
extname                 ::=  the collation name used in the .conf file
<specific-attributes>   ::=  <attribute> [; <attribute> ...]
<attribute>             ::=  attrname=attrvalue
```

- If no FROM clause is present, Firebird will scan the `.conf` file(s) in your `intl` subdirectory for a collation with the name specified after CREATE COLLATION. That is, omitting the FROM clause is the same as specifying "FROM EXTERNAL ('*collname*')".
- The single-quoted *extname* is case-sensitive and must be exactly equal to the collation name in the `.conf` file. The *collname*, *charset* and *basecoll* parameters are case-insensitive, unless surrounded by double-quotes.

Specific attributes: The table below lists the available specific attributes. Not all specific attributes apply to every collation, even if specifying them doesn't cause an error. Please note that specific attributes are case sensitive. In the table below, "*1 bpc*" indicates that an attribute is valid for collations of character sets using 1 byte per character (so-called *narrow character sets*). "*UNI*" stands for "UNICODE and UNICODE_CI".

Table 5.1. Specific collation attributes

Name	Values	Valid for	Comment
DISABLE-COMPRESSIONS	0, 1	1 bpc	Disables compressions (aka contractions). Compressions cause certain character sequences to be sorted as atomic units, e.g. Spanish `c+h` as a single character `ch`.
DISABLE-EXPANSIONS	0, 1	1 bpc	Disables expansions. Expansions cause certain characters (e.g. ligatures or umlauted vowels) to be treated as character sequences and sorted accordingly.
ICU-VERSION	`default` or *`M.m`*	UNI	Specifies the ICU library version to use. Valid values are the ones defined in the applicable `<intl_module>` element in `intl/fbintl.conf`. Format: either the string literal "`default`" or a major+minor version number like "3.0" (both unquoted).
LOCALE	*`xx_YY`*	UNI	Specifies the collation locale. Requires complete version of ICU libraries. Format: a locale string like "`du_NL`" (unquoted).
MULTI-LEVEL	0, 1	1 bpc	Uses more than one ordering level.
SPECIALS-FIRST	0, 1	1 bpc	Orders special characters (spaces, symbols etc.) before alphanumeric characters.

Examples:

Simplest form, using the name as found in the `.conf` file (case-insensitive):

```
create collation iso8859_1_unicode for iso8859_1
```

Using a custom name. Notice how the "external" name must now *exactly* match the name in the `.conf` file:

```
create collation lat_uni
  for iso8859_1
  from external ('ISO8859_1_UNICODE')
```

Based on a collation already present in the database:

```
create collation es_es_nopad_ci
  for iso8859_1
  from es_es
  no pad
  case insensitive
```

With a special attribute (case-sensitive!):

```
create collation es_es_ci_compr
  for iso8859_1
  from es_es
  case insensitive
  'DISABLE-COMPRESSIONS=0'
```

> **Tip**
>
> If you want to add a new character set with its default collation in your database, declare and run the stored procedure `sp_register_character_set(name, max_bytes_per_character)`, found in `misc/intl.sql` under your Firebird installation directory. Please note: in order for this to work, the character set must be present on your system and registered in a `.conf` file in the `intl` subdirectory.

DROP COLLATION

Available in: DSQL

Added in: 2.1

Description: Removes a collation from the database. Only user-added collations can be removed in this way.

Syntax:

```
DROP COLLATION name
```

> **Tip**
>
> If you want to remove an entire character set with all its collations from your database, declare and run the stored procedure `sp_unregister_character_set(name)`, found in `misc/intl.sql` under your Firebird installation directory.

COMMENT

Available in: DSQL

Added in: 2.0

Description: Allows you to enter comments for metadata objects. The comments will be stored in the various RDB$DESCRIPTION text BLOB fields in the system tables, from where client applications can pick them up.

Syntax:

```
COMMENT ON <object> IS {'sometext' | NULL}

<object>      ::=  DATABASE
                 | <basic-type> objectname
```

```
                    | COLUMN relationname.fieldname
                    | PARAMETER procname.paramname

<basic-type>  ::=  CHARACTER SET | COLLATION | DOMAIN | EXCEPTION
                    | EXTERNAL FUNCTION | FILTER | GENERATOR | INDEX
                    | PROCEDURE | ROLE | SEQUENCE | TABLE | TRIGGER | VIEW
```

> **Note**
>
> If you enter an empty comment (`''`), it will end up as `NULL` in the database.

Examples:

```
comment on database is 'Here''s where we keep all our customer records.'
```

```
comment on table Metals is 'Also for alloys'
```

```
comment on column Metals.IsAlloy is '0 = pure metal, 1 = alloy'
```

```
comment on index ix_sales is 'Set inactive during bulk inserts!'
```

DATABASE

CREATE DATABASE

Available in: DSQL, ESQL

Syntax (partial):

```
CREATE {DATABASE | SCHEMA}
   ...
   [PAGE_SIZE [=] size]
   ...
   [DIFFERENCE FILE 'filepath']

size  ::=  4096 | 8192 | 16384
```

- If the user supplies a size smaller than 4096, it will be silently converted to 4096. Other numbers not equal to any of the supported sizes will be silently converted to the next lower supported size.

16 Kb page size supported, 1 and 2 Kb deprecated

Changed in: 1.0, 2.1

Description: Firebird 1.0 has raised the maximum database page size from 8192 to 16384 bytes. In Firebird 2.1 and up, page sizes 1024 and 2048 are deprecated as inefficient. Firebird will no longer create databases with these page sizes, but it will connect to existing small-page databases without any problem.

DIFFERENCE FILE parameter

Added in: 2.0

Description: The DIFFERENCE FILE parameter was added in Firebird 2.0, but not documented at the time. For a full description, see *ALTER DATABASE :: ADD DIFFERENCE FILE*.

ALTER DATABASE

Available in: DSQL, ESQL

Description: Alters a database's file organisation or toggles its “safe-to-copy” state.

Syntax:

```
ALTER {DATABASE | SCHEMA}
   [<add_sec_clause> [<add_sec_clause> ...]]
   [ADD DIFFERENCE FILE 'filepath' | DROP DIFFERENCE FILE]
   [{BEGIN | END} BACKUP]

<add_sec_clause>  ::=  ADD <sec_file> [<sec_file> ...]

<sec_file>        ::=  FILE 'filepath'
                           [STARTING [AT [PAGE]] pagenum]
                           [LENGTH [=] num [PAGE[S]]
```

The DIFFERENCE FILE and BACKUP clauses, added in Firebird 2.0, are not available in ESQL.

BEGIN BACKUP

Available in: DSQL

Added in: 2.0

Description: Freezes the main database file so that it can be backed up safely by filesystem means, even while users are connected and perform operations on the data. Any mutations to the database will be written to a separate file, the *delta file*. Contrary to what the syntax suggests, this statement does *not* initiate the backup itself; it merely creates the conditions.

Example:

```
alter database begin backup
```

END BACKUP

Available in: DSQL

Added in: 2.0

Description: Merges the delta file back into the main database file and restores the normal state of operation, thus closing the time window during which safe backups could be made via the filesystem. (Safe backups with gbak are still possible.)

Example:

```
alter database end backup
```

> **Tip**
>
> Instead of BEGIN and END BACKUP, consider using Firebird's nbackup tool: it can freeze and unfreeze the main database file as well as make full and incremental backups. A manual for nbackup is available via the Firebird Documentation Index.

ADD DIFFERENCE FILE

Available in: DSQL

Added in: 2.0

Description: Presets path and name of the delta file to which mutations are written when the database goes into "copy-safe" mode after an ALTER DATABASE BEGIN BACKUP command.

Example:

```
alter database add difference file 'C:\Firebird\Databases\Fruitbase.delta'
```

Notes:

- This statement doesn't really add any file. It just overrides the default path and name for the delta file that will be created if and when the database enters copy-safe mode.
- If you provide a relative path or a bare filename here, it will be appended to the current directory as seen from the server. On Windows, this is often the system directory.
- If you want to change an existing setting, DROP the old one first and then ADD the new one.
- When not overridden, the delta file gets the same path and filename as the database itself, but with the extension `.delta`

DROP DIFFERENCE FILE

Available in: DSQL

Added in: 2.0

Description: Removes the delta file path and name that were previously set with ALTER DATABASE ADD DIFFERENCE FILE. This statement doesn't really drop a file. It only erases the preset path and/or filename that would otherwise have been used the next time the database went into copy-safe mode, and reverts to the default behaviour.

Example:

```
alter database drop difference file
```

DOMAIN

CREATE DOMAIN

Available in: DSQL, ESQL

Context variables as defaults

Changed in: IB

Description: Any context variable that is assignment-compatible to the new domain's datatype can be used as a default. This was already the case in InterBase 6, but the *Language Reference* only mentioned `USER`.

Example:

```
create domain DDate as
  date
  default current_date
  not null
```

ALTER DOMAIN

Available in: DSQL, ESQL

Warning

If you change a domain's definition, existing PSQL code using that domain may become invalid. If this happens, the system table field RDB$VALID_BLR will be set to 0 for any procedure or trigger whose code is no longer valid. If you have changed a domain, the following query will find the code modules that depend on it and report the state of RDB$VALID_BLR:

```
select * from (
  select 'Procedure', rdb$procedure_name, rdb$valid_blr from rdb$procedures
    union
  select 'Trigger', rdb$trigger_name, rdb$valid_blr from rdb$triggers
) (type, name, valid)
where exists
  (select * from rdb$dependencies
   where rdb$dependent_name = name and rdb$depended_on_name = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name. Use all-caps if the domain
   was created case-insensitively. Otherwise, use the exact capitalisation. */
```

Unfortunately, not all PSQL invalidations will be reflected in the RDB$VALID_BLR field. It is therefore advisable to look at all the procedures and triggers reported by the above query, even those having a 1 in the "VALID" column.

Please notice that for PSQL modules inherited from earlier Firebird versions (including a number of system triggers, even if the database was created under Firebird 2.1 or higher), RDB$VALID_BLR is `NULL`. This does *not* indicate that their BLR is invalid.

The isql commands SHOW PROCEDURES and SHOW TRIGGERS flag modules whose RDB$VALID_BLR field is zero with an asterisk. SHOW PROCEDURE ***`PROCNAME`*** and SHOW TRIGGER ***`TRIGNAME`***, which display individual PSQL modules, do not signal invalid BLR.

Rename domain

Added in: IB

Description: Renaming of a domain is possible with the TO clause. This feature was introduced in InterBase 6, but left out of the *Language Reference*.

Example:

```
alter domain posint to plusint
```

- The TO clause can be combined with other clauses and need not come first in that case.

SET DEFAULT to any context variable

Changed in: IB

Description: Any context variable that is assignment-compatible to the domain's datatype can be used as a default. This was already the case in InterBase 6, but the *Language Reference* only mentioned `USER`.

Example:

```
alter domain DDate
  set default current_date
```

EXCEPTION

CREATE EXCEPTION

Available in: DSQL, ESQL

Message length increased

Changed in: 2.0

Description: In Firebird 2.0 and higher, the maximum length of the exception message has been raised from 78 to 1021.

Example:

```
create exception Ex_TooManyManagers
  'Too many managers: An attempt was made to create more managers than the
   maximum defined in the Limits table. If you really need to create more
   managers than you have now, raise the limit first. However, please consult
   your department''s manager before doing so. Otherwise, your decision may
   be overturned later and the additional manager(s) removed.'
```

> **Note**
>
> The maximum exception message length depends on a certain system table field. Therefore, pre-2.0 databases need to be backed up and restored under Firebird 2.*x* before they can store exception messages of up to 1021 bytes.

CREATE OR ALTER EXCEPTION

Available in: DSQL

Added in: 2.0

Description: If the exception does not yet exist, it is created just as if CREATE EXCEPTION were used. If it already exists, it is altered. Existing dependencies are preserved.

Syntax: Exactly the same as for CREATE EXCEPTION.

RECREATE EXCEPTION

Available in: DSQL

Added in: 2.0

Description: Creates or recreates an exception. If an exception with the same name already exists, RECREATE EXCEPTION will try to drop it and create a new exception. This wlll fail if there are existing dependencies on the exception.

Syntax: Exactly the same as CREATE EXCEPTION.

> **Note**
>
> If you use RECREATE EXCEPTION on an exception that has dependent objects, you may not get an error message until you try to commit your transaction.

EXTERNAL FUNCTION

DECLARE EXTERNAL FUNCTION

Available in: DSQL, ESQL

Description: This statement makes an external function (UDF) available in the database.

Syntax:

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

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

Restrictions

- The BY DESCRIPTOR passing method is not supported in ESQL.

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

Available in: DSQL

Added in: 1.0

Description: Firebird introduces the possibility to pass parameters BY DESCRIPTOR; this mechanism facilitates the processing of NULLs 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*

Available in: DSQL, ESQL

Added in: IB 6

Description: In order to return a BLOB, an extra input parameter must be declared and a "RETURNS PARAMETER *n*" clause added – *n* being the position of said parameter. This clause 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).

ALTER EXTERNAL FUNCTION

Available in: DSQL

Added in: 2.0

Description: Alters an external function's module name and/or entry point. Existing dependencies are preserved.

Syntax:

```
ALTER EXTERNAL FUNCTION funcname
   <modification> [<modification>]

<modification>  ::=  ENTRY_POINT 'new-entry-point'
                       | MODULE_NAME 'new-module-name'
```

Example:

```
alter external function Phi module_name 'NewUdfLib'
```

FILTER

DECLARE FILTER

Available in: DSQL, ESQL

Changed in: 2.0

Description: Makes a BLOB filter available to the database.

Syntax:

```
DECLARE FILTER filtername
   INPUT_TYPE <sub_type> OUTPUT_TYPE <sub_type>
   ENTRY_POINT 'function_name' MODULE_NAME 'library_name'

<sub_type>  ::=  number | <mnemonic>
<mnemonic>  ::=  binary | text | blr | acl | ranges | summary | format
                  | transaction_description | external_file_description
                  | user_defined
```

- In Firebird 2 and up, no two BLOB filters in a database may have the same combination of input and output type. Declaring a filter with an already existing input-output type combination will fail. Restoring pre-2.0 databases that contain such “duplicate” filters will also fail.
- The possibility to indicate the BLOB types with their mnemonics instead of numbers was added in Firebird 2. The `binary` mnemonic for subtype 0 was also added in Firebird 2. The predefined mnemonics are case-insensitive.

Example:

```
declare filter Funnel
  input_type blr output_type text
  entry_point 'blr2asc' module_name 'myfilterlib'
```

User-defined mnemonics: If you want to define mnemonics for your own BLOB subtypes, you can add them to the RDB$TYPES system table as shown below. Once committed, the mnemonics can be used in subsequent filter declarations.

```
insert into rdb$types (rdb$field_name, rdb$type, rdb$type_name)
  values ('RDB$FIELD_SUB_TYPE', -33, 'MIDI')
```

The value for `rdb$field_name` must always be 'RDB$FIELD_SUB_TYPE'. If you define your mnemonics in all-uppercase, you can use them case-insensitively and unquoted in your filter declarations.

INDEX

CREATE INDEX

Available in: DSQL, ESQL

Description: Creates an index on a table for faster searching, sorting and/or grouping.

Syntax:

```
CREATE [UNIQUE] [ASC[ENDING] | [DESC[ENDING]] INDEX indexname
   ON tablename
   { (<col> [, <col> ...]) | COMPUTED BY (expression) }

<col>  ::=  a column not of type ARRAY, BLOB or COMPUTED BY
```

UNIQUE indices now allow NULLs

Changed in: 1.5

Description: In compliance with the SQL-99 standard, NULLs – 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 NULLs*. As far as NULLs are concerned, the rules for unique indices are exactly the same as those for unique keys.

Indexing on expressions

Added in: 2.0

Description: Instead of one or more columns, you can now also specify a single COMPUTED BY expression in an index definition. Expression indices will be used in appropriate queries, provided that the expression in the WHERE, ORDER BY or GROUP BY clause exactly matches the expression in the index definition. Multi-segment expression indices are not supported, but the expression itself may involve multiple columns.

Examples:

```
create index ix_upname on persons computed by (upper(name));
commit;
```

```
-- the following queries will use ix_upname:
select * from persons order by upper(name);
select * from persons where upper(name) starting with 'VAN';
delete from persons where upper(name) = 'BROWN';
delete from persons where upper(name) = 'BROWN' and age > 65;
```

```
create descending index ix_events_yt
  on MyEvents
  computed by (extract(year from StartDate) || Town);
commit;

-- the following query will use ix_events_yt:
select * from MyEvents
  order by extract(year from StartDate) || Town desc;
```

Maximum index key length increased

Changed in: 2.0

Description: The maximum length of index keys, which used to be fixed at 252 bytes, is now equal to 1/4 of the page size, i.e. varying from 256 to 4096. The maximum indexable string length in bytes is 9 less than the key length. The table below shows the indexable string lengths in characters for the various page sizes and character sets.

Table 5.2. Maximum indexable (VAR)CHAR length

Page size	Maximum indexable string length per charset type			
	1 byte/char	2 bytes/char	3 bytes/char	4 bytes/char
1024	247	123	82	61
2048	503	251	167	125
4096	1015	507	338	253
8192	2039	1019	679	509
16384	4087	2043	1362	1021

Maximum number of indices per table increased

Changed in: 1.0.3, 1.5, 2.0

Description: The maximum number of 65 indices per table has been removed in Firebird 1.0.3, reintroduced at the higher level of 257 in Firebird 1.5, and removed once again in Firebird 2.0.

Although there is no longer a “hard” ceiling, the number of indices creatable in practice is still limited by the database page size and the number of columns per index, as shown in the table below.

Table 5.3. Max. indices per table, Firebird 2.0

Page size	Number of indices depending on column count		
	1 col	2 cols	3 cols
1024	50	35	27
2048	101	72	56
4096	203	145	113
8192	408	291	227
16384	818	584	454

Please be aware that under normal circumstances, even 50 indices is way too many and will drastically reduce mutation speeds. The maximum was removed to accommodate data-warehousing applications and the like, which perform lots of bulk operations with the indices temporarily inactivated.

For a full table also including Firebird versions 1.0–1.5, see the Notes at the end of the book.

Privileges: GRANT and REVOKE

REVOKE ADMIN OPTION

Available in: DSQL

Added in: 2.0

Description: Revokes a previously granted admin option (the right to pass on a granted role to others) from the grantee, without revoking the role itself. Multiple roles and/or multiple grantees can be handled in one statement.

Syntax:

```
REVOKE ADMIN OPTION FOR <role-list> FROM <grantee-list>

<role-list>     ::=  role [, role ...]
<grantee-list>  ::=  [USER] <grantee> [, [USER] <grantee> ...]
<grantee>       ::=  username | PUBLIC
```

Example:

```
revoke admin option for manager from john, paul, george, ringo
```

If a user has received the admin option from several grantors, each of those grantors must revoke it or the user will still be able to grant the role(s) in question to others.

PROCEDURE

A stored procedure (SP) is a code module that can be called by the client, by another stored procedure or by a trigger. Stored procedures and triggers are written in Procedural SQL (PSQL). Most SQL statements are also available in PSQL, sometimes with restrictions or extensions. Notable exceptions are DDL and transaction control statements.

Stored procedures can accept and return multiple parameters.

CREATE PROCEDURE

Available in: DSQL, ESQL

Description: Creates a stored procedure.

Syntax:

```
CREATE PROCEDURE procname
   [(<inparam> [, <inparam> ...])]
   [RETURNS (<outparam> [, <outparam> ...])]
AS
   [<declarations>]
BEGIN
   [<PSQL statements>]
END

<inparam>       ::=  <param_decl> [{= | DEFAULT} value]
<outparam>      ::=  <param_decl>
<param_decl>    ::=  paramname <type> [NOT NULL] [COLLATE collation]
<type>          ::=  sql_datatype | [TYPE OF] domain
<declarations>  ::=  See PSQL::DECLARE for the exact syntax

/* If sql_datatype is a string type, it may include a character set */
```

Domains supported in parameter and variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up support the use of domains instead of SQL datatypes when declaring input/output parameters and local variables. With the "TYPE OF" modifier only the domain's type is used, not its NOT NULL setting, CHECK constraint and/or default value.

Example:

```
create domain bool3
  smallint
  check (value is null or value in (0,1));

create domain bigposnum
  bigint
  check (value >= 0);
```

```
/* Determines if A is a multiple of B: */
set term #;
create procedure ismultiple (a bigposnum, b bigposnum)
  returns (res bool3)
as
  declare ratio type of bigposnum;      -- ratio is a bigint
  declare remainder type of bigposnum;  -- so is remainder
begin
  if (a is null or b is null) then res = null;
  else if (b = 0) then
  begin
    if (a = 0) then res = 1; else res = 0;
  end
  else
  begin
    ratio = a / b;                      -- integer division!
    remainder = a - b*ratio;
    if (remainder = 0) then res = 1; else res = 0;
  end
end#
set term ;#
```

Warning

If you change a domain's definition, existing PSQL code using that domain may become invalid. If this happens, the system table field RDB$VALID_BLR will be set to 0 for any procedure or trigger whose code is no longer valid. If you have changed a domain, the following query will find the code modules that depend on it and report the state of RDB$VALID_BLR:

```
select * from (
  select 'Procedure', rdb$procedure_name, rdb$valid_blr from rdb$procedures
    union
  select 'Trigger', rdb$trigger_name, rdb$valid_blr from rdb$triggers
) (type, name, valid)
where exists
  (select * from rdb$dependencies
   where rdb$dependent_name = name and rdb$depended_on_name = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name. Use all-caps if the domain
   was created case-insensitively. Otherwise, use the exact capitalisation. */
```

Unfortunately, not all PSQL invalidations will be reflected in the RDB$VALID_BLR field. It is therefore advisable to look at all the procedures and triggers reported by the above query, even those having a 1 in the "VALID" column.

Please notice that for PSQL modules inherited from earlier Firebird versions (including a number of system triggers, even if the database was created under Firebird 2.1 or higher), RDB$VALID_BLR is `NULL`. This does *not* indicate that their BLR is invalid.

The isql commands SHOW PROCEDURES and SHOW TRIGGERS flag modules whose RDB$VALID_BLR field is zero with an asterisk. SHOW PROCEDURE *PROCNAME* and SHOW TRIGGER *TRIGNAME*, which display individual PSQL modules, do not signal invalid BLR.

COLLATE in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow COLLATE clauses in declarations of input/output parameters and local variables.

Example:

```
create procedure SpanishToDutch
  (es_1 varchar(20) character set iso8859_1 collate es_es,
   es_2 my_char_domain collate es_es)
returns
  (nl_1 varchar(20) character set iso8859_1 collate du_nl,
   nl_2 my_char_domain collate du_nl)
as
declare s_temp varchar(100) character set utf8 collate unicode;
begin
  ...
  ...
end
```

NOT NULL in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow NOT NULL constraints in declarations of input/output parameters and local variables.

Example:

```
create procedure RegisterOrder(order_no int not null, description varchar(200) not null
returns
  (ticket_no int not null)
as
declare temp int not null;
begin
  ...
  ...
end
```

Default argument values

Changed in: 2.0

Description: It is now possible to provide default values for stored procedure arguments, allowing the caller to omit one or more items (possibly even all) from the end of the argument list.

Syntax:

```
CREATE PROCEDURE procname (<inparam> [, <inparam> ...])
   ...

<inparam>  ::=  paramname datatype [{= | DEFAULT} value]
```

Important: If you provide a default value for a parameter, you must do the same for any and all parameters following it.

BEGIN ... END blocks may be empty

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 procedure grab_ints (a integer, b integer)
as
begin
end
```

ALTER PROCEDURE

Available in: DSQL, ESQL

Default argument values

Added in: 2.0

Description: You can now provide default values for stored procedure arguments, allowing the caller to omit one or more items from the end of the argument list. See *CREATE PROCEDURE* for syntax and details.

Example:

```
alter procedure TestProc
  (a int, b int default 1007, s varchar(12) = '-')
  ...
```

COLLATE in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow COLLATE clauses in declarations of input/output parameters and local variables. See *CREATE PROCEDURE* for syntax and details.

Domains supported in parameter and variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up support the use of domains instead of SQL datatypes when declaring input/output parameters and local variables. See *CREATE PROCEDURE* for syntax and details.

NOT NULL in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow NOT NULL constraints in declarations of input/output parameters and local variables. See *CREATE PROCEDURE* for syntax and details.

Restriction on altering used procedures

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

CREATE OR ALTER PROCEDURE

Available in: DSQL

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.

DROP PROCEDURE

Available in: DSQL, ESQL

Restriction on dropping used procedures

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

RECREATE PROCEDURE

Available in: DSQL

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.

Restriction on recreating used procedures

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

SEQUENCE or GENERATOR

CREATE SEQUENCE

Available in: DSQL

Added in: 2.0

Description: Creates a new sequence or generator. SEQUENCE is the SQL-compliant term for what InterBase and Firebird have always called a generator. CREATE SEQUENCE is fully equivalent to CREATE GENERATOR and is the recommended syntax from Firebird 2.0 onward.

Syntax:

```
CREATE SEQUENCE sequence-name
```

Example:

```
create sequence seqtest
```

Because internally sequences and generators are the same thing, you can freely mix the generator and sequence syntaxes, even when operating on the same object. This is not recommended however.

Sequences (or generators) are always stored as 64-bit integer values, regardless of the database dialect. However:

- If the *client* dialect is set to 1, the server passes generator values as truncated 32-bit values to the client.
- If generator values are fed into a 32-bit field or variable, all goes well until the actual value exceeds the 32-bit range. At that point, a dialect 3 database will raise an error whereas a dialect 1 database will silently truncate the value (which could also lead to an error, e.g. if the receiving field has a unique key defined on it).

See also: ALTER SEQUENCE, NEXT VALUE FOR, DROP SEQUENCE

CREATE GENERATOR

Available in: DSQL, ESQL

Better alternative: CREATE SEQUENCE

CREATE SEQUENCE preferred

Changed in: 2.0

Description: From Firebird 2.0 onward, the SQL-compliant CREATE SEQUENCE syntax is preferred.

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.

ALTER SEQUENCE

Available in: DSQL

Added in: 2.0

Description: (Re)initializes a sequence or generator to the given value. SEQUENCE is the SQL-compliant term for what InterBase and Firebird have always called a generator. "ALTER SEQUENCE ... RESTART WITH" is fully equivalent to "SET GENERATOR ... TO" and is the recommended syntax from Firebird 2.0 onward.

Syntax:

```
ALTER SEQUENCE sequence-name RESTART WITH <newval>

<newval>  ::=  A signed 64-bit integer value.
```

Example:

```
alter sequence seqtest restart with 0
```

> **Warning**
>
> Careless use of ALTER SEQUENCE is a mighty fine way of screwing up your database! Under normal circumstances you should only use it right after CREATE SEQUENCE, to set the initial value.

See also: CREATE SEQUENCE

SET GENERATOR

Available in: DSQL, ESQL

Better alternative: ALTER SEQUENCE

Description: (Re)initializes a generator or sequence to the given value. From Firebird 2 onward, the SQL-compliant ALTER SEQUENCE syntax is preferred.

Syntax:

```
SET GENERATOR generator-name TO <new-value>

<new-value>  ::=  A 64-bit integer.
```

> **Warning**
>
> Once a generator or sequence is up and running, you should not tamper with its value (other than retrieving next values with GEN_ID or NEXT VALUE FOR) unless you know exactly what you are doing.

DROP SEQUENCE

Available in: DSQL

Added in: 2.0

Description: Removes a sequence or generator from the database. Its (very small) storage space will be freed for re-use after a backup-restore cycle. SEQUENCE is the SQL-compliant term for what InterBase and Firebird have always called a generator. DROP SEQUENCE is fully equivalent to DROP GENERATOR and is the recommended syntax from Firebird 2.0 onward.

Syntax:

```
DROP SEQUENCE sequence-name
```

Example:

```
drop sequence seqtest
```

See also: CREATE SEQUENCE

DROP GENERATOR

Available in: DSQL

Added in: 1.0

Better alternative: DROP SEQUENCE

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

Syntax:

```
DROP GENERATOR generator-name
```

From Firebird 2.0 onward, the SQL-compliant DROP SEQUENCE syntax is preferred.

TABLE

CREATE TABLE

Available in: DSQL, ESQL

Global Temporary Tables (GTTs)

Added in: 2.1

Description: Global temporary tables have persistent metadata, but their contents are transaction-bound (the default) or connection-bound. Every transaction or connection has its own private instance of a GTT, isolated from all the others. Instances are only created if and when the GTT is referenced, and destroyed upon transaction end or disconnection. To modify or remove a GTT's metadata, ALTER TABLE and DROP TABLE can be used.

Syntax:

```
CREATE GLOBAL TEMPORARY TABLE name
   (column_def [, column_def | table_constraint ...])
   [ON COMMIT {DELETE | PRESERVE} ROWS]
```

- ON COMMIT DELETE ROWS creates a transaction-level GTT (the default), ON COMMIT PRESERVE ROWS a connection-level GTT.
- An EXTERNAL [FILE] clause is not allowed on a global temporary table.

Restrictions: GTTs can be “dressed up” with all the features and paraphernalia of ordinary tables (keys, references, indices, triggers...) but there are a few restrictions:

- GTTs and regular tables cannot reference one another.
- A connection-bound (“PRESERVE ROWS”) GTT cannot reference a transaction-bound (“DELETE ROWS”) GTT.
- Domain constraints cannot reference any GTT.
- The destruction of a GTT instance at the end of its life cycle does *not* cause any before/after delete triggers to fire.

Example:

```
create global temporary table MyConnGTT (
  id int not null primary key,
  txt varchar(32),
  ts timestamp default current_timestamp
)
on commit preserve rows;

commit;

create global temporary table MyTxGTT (
  id int not null primary key,
  parent_id int not null references MyConnGTT(id),
  txt varchar(32),
  ts timestamp default current_timestamp
);

commit;
```

Tip

In an existing database, it's not always easy to tell a regular table from a GTT, or a transaction-level GTT from a connection-level GTT. Use this query to find out a table's type:

```
select t.rdb$type_name
  from rdb$relations r
  join rdb$types t on r.rdb$relation_type = t.rdb$type
  where t.rdb$field_name = 'RDB$RELATION_TYPE'
  and r.rdb$relation_name = 'TABLENAME'
```

Or, for an overview of all your relations:

```
select r.rdb$relation_name, t.rdb$type_name
  from rdb$relations r
  join rdb$types t on r.rdb$relation_type = t.rdb$type
  where t.rdb$field_name = 'RDB$RELATION_TYPE'
  and coalesce (r.rdb$system_flag, 0) = 0
```

GENERATED ALWAYS AS

Added in: 2.1

Description: Instead of COMPUTED [BY], you may also use the SQL-2003-compliant equivalent GENERATED ALWAYS AS for computed fields.

Syntax:

```
colname [coltype] GENERATED ALWAYS AS (expression)
```

Example:

```
create table Persons (
  id int primary key,
  firstname varchar(24) not null,
  middlename varchar(24),
```

```
  lastname varchar(24) not null,
  fullname varchar(74) generated always as
    (firstname || coalesce(' ' || middlename, '') || ' ' || lastname),
  street varchar(32),
  ...
  ...
)
```

Note: GENERATED ALWAYS AS is not currently supported in index definitions.

CHECK accepts NULL outcome

Changed in: 2.0

Description: If a CHECK constraint resolves to `NULL`, Firebird versions before 2.0 reject the input. Following the SQL standard to the letter, Firebird 2.0 and above let `NULL`s pass and only consider the check failed if the outcome is `false`.

Example:

Checks like these:

```
check (value > 10000)
```

```
check (Town like 'Amst%')
```

```
check (upper(value) in ( 'A', 'B', 'X' ))
```

```
check (Minimum <= Maximum)
```

all *fail* in pre-2.0 Firebird versions if the value to be checked is `NULL`. In 2.0 and above they *succeed*.

> **Warning**
>
> This change may cause existing databases to behave differently when migrated to Firebird 2.0+. Carefully examine your CREATE/ALTER TABLE statements and add "`and XXX is not null`" predicates to your CHECKs if they should continue to reject `NULL` input.

Context variables as column defaults

Changed in: IB

Description: Any context variable that is assignment-compatible to the column datatype can be used as a default. This was already the case in InterBase 6, but the *Language Reference* only mentioned `USER`.

Example:

```
create table MyData (
  id int not null primary key,
  record_created timestamp default current_timestamp,
  ...
)
```

FOREIGN KEY without target column references PK

Changed in: IB

Description: If you create a foreign key without specifying a target column, it will reference the **primary key** of the target table. This was already the case in InterBase 6, but the IB Language Reference wrongly states that in such cases, the engine scans the target table for a column with the same name as the referencing column.

Example:

```
create table eik (
  a int not null primary key,
  b int not null unique
);

create table beuk (
  b int references eik
);

-- beuk.b references eik.a, not eik.b !
```

FOREIGN KEY creation no longer requires exclusive access

Changed in: 2.0

Description: In Firebird 2.0 and above, creating a foreign key constraint no longer requires exclusive access to the database.

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

Available in: DSQL

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.

ALTER TABLE

Available in: DSQL, ESQL

ADD column: Context variables as defaults

Changed in: IB

Description: Any context variable that is assignment-compatible to the new column's datatype can be used as a default. This was already the case in InterBase 6, but the *Language Reference* only mentioned `USER`.

Example:

```
alter table MyData
  add MyDay date default current_date
```

ALTER COLUMN: DROP DEFAULT

Available in: DSQL

Added in: 2.0

Description: Firebird 2 adds the possibility to drop a column-level default. Once the default is dropped, there will either be no default in place or – if the column's type is a DOMAIN with a default – the domain default will resurface.

Syntax:

```
ALTER TABLE tablename ALTER [COLUMN] colname DROP DEFAULT
```

Example:

```
alter table Trees alter Girth drop default
```

An error is raised if you use DROP DEFAULT on a column that doesn't have a default or whose effective default is domain-based.

ALTER COLUMN: SET DEFAULT

Available in: DSQL

Added in: 2.0

Description: Firebird 2 adds the possibility to set/alter defaults on existing columns. If the column already had a default, the new default will replace it. Column-level defaults always override domain-level defaults.

Syntax:

```
ALTER TABLE tablename ALTER [COLUMN] colname SET DEFAULT <default>

<default>  ::=  literal-value | context-variable | NULL
```

Example:

```
alter table Customers alter EnteredBy set default current_user
```

> **Tip**
>
> If you want to switch off a domain-based default on a column, set the column default to `NULL`.

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.

CHECK accepts NULL outcome

Changed in: 2.0

Description: If a CHECK constraint resolves to `NULL`, Firebird versions before 2.0 reject the input. Following the SQL standard to the letter, Firebird 2.0 and above let `NULL`s pass and only consider the check failed if the outcome is `false`. For more information see under *CREATE TABLE*.

FOREIGN KEY without target column references PK

Changed in: IB

Description: If you create a foreign key without specifying a target column, it will reference the **primary key** of the target table. This was already the case in InterBase 6, but the IB Language Reference wrongly states that in such cases, the engine scans the target table for a column with the same name as the referencing column.

Example:

```
create table eik (
  a int not null primary key,
  b int not null unique
);

create table beuk (
  b int
);

alter table beuk
  add constraint fk_beuk
  foreign key (b) references eik;

-- beuk.b now references eik.a, not eik.b !
```

FOREIGN KEY creation no longer requires exclusive access

Changed in: 2.0

Description: In Firebird 2.0 and above, adding a foreign key constraint no longer requires exclusive access to the database.

GENERATED ALWAYS AS

Added in: 2.1

Description: Instead of COMPUTED [BY], you may also use the SQL-2003-compliant equivalent GENERATED ALWAYS AS for computed fields.

Syntax:

```
colname [coltype] GENERATED ALWAYS AS (expression)
```

Example:

```
alter table Friends
  add fullname varchar(74)
  generated always as
    (firstname || coalesce(' ' || middlename, '') || ' ' || lastname)
```

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. For a full discussion, see *CREATE TABLE :: UNIQUE constraints now allow NULLs*.

USING INDEX subclause

Available in: DSQL

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*.

RECREATE TABLE

Available in: DSQL

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*.

TRIGGER

CREATE TRIGGER

Available in: DSQL, ESQL

Description: Creates a trigger, a block of PSQL code that is executed automatically upon certain database events or mutations to a table or view.

Syntax:

```
CREATE TRIGGER name
   {<relation_trigger_legacy>
      | <relation_trigger_sql2003>
      | <database_trigger>        }
   AS
      [<declarations>]
   BEGIN
      [<statements>]
   END

<relation_trigger_legacy>   ::=  FOR {tablename | viewname}
                                   [ACTIVE | INACTIVE]
                                   {BEFORE | AFTER} <mutation_list>
                                   [POSITION number]

<relation_trigger_sql2003>  ::=  [ACTIVE | INACTIVE]
                                   {BEFORE | AFTER} <mutation_list>
                                   [POSITION number]
                                   ON {tablename | viewname}

<database_trigger>          ::=  [ACTIVE | INACTIVE]
                                   ON db_event
                                   [POSITION number]

<mutation_list>             ::=  mutation [OR mutation [OR mutation]]
mutation                    ::=  INSERT | UPDATE | DELETE

db_event                    ::=  CONNECT | DISCONNECT | TRANSACTION START
                                   | TRANSACTION COMMIT | TRANSACTION ROLLBACK

number                      ::=  0..32767 (default is 0)

<declarations>              ::=  See PSQL::DECLARE for the exact syntax
```

- “Legacy” and “sql2003” relation triggers are exactly the same. The only thing that differs is the creation syntax.
- Triggers with lower position numbers fire first. Position numbers need not be unique, but if two or more triggers have the same position, the firing order between them is undefined.
- When defining relation triggers, each mutation type (INSERT, UPDATE or DELETE) may occur at most once in the mutation list.

SQL-2003-compliant syntax for relation triggers

Added in: 2.1

Description: Since Firebird 2.1, an alternative, SQL-2003-compliant syntax can be used for triggers on tables and views. Instead of specifying “FOR *relationname*” before the event type and the optional directives surrounding it, you can now put “ON *relationname*” after it, as shown in the syntax earlier in this chapter.

Example:

```
create trigger biu_books
  active before insert or update position 3
  on books
as
begin
  if (new.id is null)
    then new.id = next value for gen_bookids;
end
```

Database triggers

Added in: 2.1

Description: Since Firebird 2.1, triggers can be defined to fire upon the database events CONNECT, DISCONNECT, TRANSACTION START, TRANSACTION COMMIT and TRANSACTION ROLLBACK. Only the database owner and SYSDBA can create, alter and drop these triggers.

Syntax:

```
CREATE TRIGGER name
   [ACTIVE | INACTIVE]
   ON db_event
   [POSITION number]
   AS
      [<declarations>]
   BEGIN
      [<statements>]
   END

db_event                    ::=  CONNECT | DISCONNECT | TRANSACTION START
                                    | TRANSACTION COMMIT | TRANSACTION ROLLBACK

number                      ::=  0..32767 (default is 0)

<declarations>              ::=  See PSQL::DECLARE for the exact syntax
```

Example:

```
create trigger tr_connect
  on connect
as
begin
  insert into dblog (wie, wanneer, wat)
    values (current_user, current_timestamp, 'verbind');
end
```

Execution of database triggers and handling of exceptions:

- CONNECT and DISCONNECT triggers are executed in a transaction created specifically for this purpose. If all goes well, the transaction is committed. Uncaught exceptions roll back the transaction, and:
  - In the case of a CONNECT trigger, the connection is then broken and the exception returned to the client.
  - With a DISCONNECT trigger, exceptions are not reported and the connection is broken as foreseen.

- TRANSACTION triggers are executed within the transaction whose opening, committing or rolling-back evokes them. The actions taken after an uncaught exception depend on the type:
  - In a START trigger, the exception is reported to the client and the transaction is rolled back.
  - In a COMMIT trigger, the exception is reported, the trigger's actions so far are undone and the commit is canceled.
  - In a ROLLBACK trigger, the exception is not reported and the transaction is rolled back as foreseen.
- It follows from the above that there is no direct way of knowing if a DISCONNECT or TRANSACTION ROLLBACK trigger caused an exception.
- It also follows that you can't connect to a database if a CONNECT trigger causes an exception, and that you can't start a transaction if a TRANSACTION START trigger does so. Both phenomena effectively lock you out of your database while you need to get in there to fix the problem. See the note below for a way around this Catch-22 situation.
- In the case of a two-phase commit, TRANSACTION COMMIT triggers fire in the prepare, not the commit phase.

> **Note**
>
> Some Firebird command-line tools have been supplied with new switches to suppress the automatic firing of database triggers:
>
> ```
> gbak -nodbtriggers
> isql -nodbtriggers
> nbackup -T
> ```
>
> These switches can only be used by the database owner and SYSDBA.

Domains instead of datatypes

Changed in: 2.1

Description: Firebird 2.1 and up allow the use of domains instead of SQL datatypes when declaring local trigger variables. See *PSQL::DECLARE* for the exact syntax and details.

COLLATE in variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow COLLATE clauses in local variable declarations. See *PSQL::DECLARE* for syntax and details.

NOT NULL in variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow NOT NULL constraints in local variable declarations. See *PSQL::DECLARE* for syntax and details.

Multi-action triggers

Added in: 1.5

Description: Relation triggers can 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.

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.

BEGIN ... END blocks may be empty

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
```

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.

ALTER TRIGGER

Available in: DSQL, ESQL

Description: Alters an existing trigger. Relation triggers cannot be changed into database triggers or vice versa. The associated table or view of a relation trigger cannot be changed.

Syntax:

```
ALTER TRIGGER name
   [ACTIVE | INACTIVE]
   [{BEFORE | AFTER} <mutation_list> | ON db_event]
   [POSITION number]
   [AS
       [<declarations>]
    BEGIN
       [<statements>]
    END                 ]
```

- See *CREATE TRIGGER* for the meaning of `<mutation_list>` etc.

Database triggers

Added in: 2.1

Description: The ALTER TRIGGER syntax (see above) has been extended to support database triggers. For a full discussion of this feature, see *CREATE TRIGGER :: Database triggers*.

Domains instead of datatypes

Changed in: 2.1

Description: Firebird 2.1 and up allow the use of domains instead of SQL datatypes when declaring local trigger variables. See *PSQL::DECLARE* for the exact syntax and details.

COLLATE in variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow COLLATE clauses in local variable declarations. See *PSQL::DECLARE* for syntax and details.

NOT NULL in variable declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow NOT NULL constraints in local variable declarations. See *PSQL::DECLARE* for syntax and details.

Multi-action triggers

Added in: 1.5

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

Restriction on altering used triggers

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

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.

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.

CREATE OR ALTER TRIGGER

Available in: DSQL

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.

DROP TRIGGER

Available in: DSQL, ESQL

Restriction on dropping used triggers

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

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 TRIGGER

Available in: DSQL

Added in: 2.0

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

Syntax: Exactly the same as CREATE TRIGGER.

Restriction on recreating used triggers

Changed in: 2.0, 2.0.1

Description: In Firebird 2.0 only, a restriction is in place which prevents anyone from dropping, altering or recreating a trigger or stored procedure if it has been used since the database was opened. This restriction has been removed again in version 2.0.1. Still, performing these operations on a live database is potentially risky and should only be done with the utmost care.

VIEW

CREATE VIEW

Available in: DSQL, ESQL

Per-column aliases supported in view definition

Changed in: 2.1

Description: Firebird 2.1 and up allow the use of column aliases in the SELECT statement. You can alias none, some or all of the columns; each alias used becomes the name of the corresponding view column.

Syntax (partial):

```
CREATE VIEW viewname [<full_column_list>]
   AS
   SELECT <column_def> [, <column_def> ...]
   FROM ...
   [WITH CHECK OPTION]

<full_column_list>  ::=  (colname [, colname ...])

<column_def>        ::=  {source_col | expr} [[AS] colalias]
```

Notes:

- If the full column list is also present, specifying column aliases is futile as they will be overridden by the names in the column list.
- The full column list used to be mandatory for views whose SELECT statement contains expression-based columns or identical column names. Now you can omit the full column list, provided that you alias such columns in the SELECT clause.

Full SELECT syntax supported

Changed in: 2.0

Description: From Firebird 2.0 onward view definitions are considered full-fledged SELECT statements. Consequently, the following elements are (re)allowed in view definitions: FIRST, SKIP, ROWS, ORDER BY, PLAN and UNION.

> **Note**
>
> The use of a UNION within a view is currently only supported if you supply a column list for the view (this list is normally optional):
>
> ```
> create view vplanes (make, model) as
>   select make, model from jets
>     union
>   select make, model from props
>     union
>   select make, model from gliders
> ```
>
> In Firebird 2.5, the column list will become optional also for views with UNIONs.

PLAN subclause disallowed in 1.5, reallowed in 2.0

Changed in: 1.5, 2.0

Description: Firebird versions 1.5.x forbid the use of a PLAN subclause in a view definition. From 2.0 onward a PLAN is allowed again.

Triggers on updatable views block auto-writethrough

Changed in: 2.0

Description: In versions prior to 2.0, Firebird often did not block the automatic writethrough to the underlying table if one or more triggers were defined on a naturally updatable view. This could cause mutations to be performed twice unintentionally, sometimes leading to data corruption and other mishaps. Starting at Firebird 2.0, this misbehaviour has been corrected: now if you define a trigger on a naturally updatable view, no mutations to the view will be automatically passed on to the table; either your trigger takes care of that, or nothing will. This is in accordance with the description in the InterBase 6 *Data Definition Guide* under *Updating views with triggers*.

> **Warning**
>
> Some people have developed code that counts on or takes advantage of the prior behaviour. Such code should be corrected for Firebird 2.0 and higher, or mutations may not reach the table at all.

View with non-participating NOT NULL columns in base table can be made insertable

Changed in: 2.0

Description: Any view whose base table contains one or more non-participating NOT NULL columns is read-only by nature. It can be made updatable by the use of triggers, but even with those, all INSERT attempts into such views used to fail because the NOT NULL constraint on the base table was checked before the view trigger

got a chance to put things right. In Firebird 2.0 and up this is no longer the case: provided the right trigger is in place, such views are now insertable.

Example:

The view below would give validation errors for any insert attempts in Firebird 1.5 and earlier. In Firebird 2.0 and up it is insertable:

```
create table base (x int not null, y int not null);

create view vbase as select x from base;

set term #;
create trigger bi_base for vbase before insert
as
begin
  if (new.x is null) then new.x = 33;
  insert into base values (new.x, 0);
end#
set term ;#
```

Notes:

- Please notice that the problem described above only occurred for NOT NULL columns that were left *outside* the view.
- Oddly enough, the problem would be gone if the base table itself had a trigger converting `NULL` input to something valid. But then there was a risk that the insert would take place twice, due to the auto-writethrough bug that has also been fixed in Firebird 2.

RECREATE VIEW

Available in: DSQL

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

DELETE

Available in: DSQL, ESQL, PSQL

Description: Deletes rows from a database table (or from one or more tables underlying a view), depending on the WHERE and ROWS clauses.

Syntax:

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

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

Restrictions

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

COLLATE subclause for text BLOB columns

Added in: 2.0

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

Example:

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

ORDER BY

Available in: DSQL, PSQL

Added in: 2.0

Description: DELETE now allows an ORDER BY clause. This only makes sense in combination with ROWS, but is also valid without it.

PLAN

Available in: DSQL, PSQL

Added in: 2.0

Description: DELETE now allows a PLAN clause, so users can optimize the operation manually.

Relation alias makes real name unavailable

Changed in: 2.0

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

Examples:

Correct usage:

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

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

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

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

No longer possible:

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

RETURNING

Available in: DSQL, PSQL

Added in: 2.1

Description: A DELETE statement removing *at most one row* may optionally include a RETURNING clause in order to return values from the deleted row. The clause, if present, need not contain all of the relation's columns and may also contain other columns or expressions.

Examples:

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

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

Notes:

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

ROWS

Available in: DSQL, PSQL

Added in: 2.0

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

Syntax:

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

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

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

Points to note:

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

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

Points to note when using two arguments:

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

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

EXECUTE BLOCK

Available in: DSQL

Added in: 2.0

Changed in: 2.1

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

Syntax:

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

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

Examples:

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

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

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

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

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

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

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

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

Notes:

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

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

  In Firebird's isql client you must set the terminator to something other than "`;`" before you type in the EXECUTE BLOCK statement. Otherwise isql, being line-oriented, will try to execute the part you have entered as soon as it encounters the first semicolon.

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

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

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

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

COLLATE in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow COLLATE clauses in declarations of input/output parameters and local variables.

Example:

```
execute block
  (es_1 varchar(20) character set iso8859_1 collate es_es = ?)
returns
  (nl_1 varchar(20) character set iso8859_1 collate du_nl)
as
  declare s_temp varchar(100) character set utf8 collate unicode;
begin
  ...
  ...
end
```

NOT NULL in variable and parameter declarations

Changed in: 2.1

Description: Firebird 2.1 and up allow NOT NULL constraints in declarations of input/output parameters and local variables.

Example:

```
execute block (a int not null = ?, b int not null = ?)
returns (product bigint not null, message varchar(20) not null)
as
  declare useless_dummy timestamp not null;
begin
  product = a*b;
  if (product < 0) then message = 'This is below zero.';
  else if (product > 0) then message = 'This is above zero.';
  else message = 'This must be zero.';
  suspend;
end
```

Domains instead of datatypes

Changed in: 2.1

Description: Firebird 2.1 and up allow the use of domains instead of SQL datatypes when declaring input/output parameters and local variables. With the "TYPE OF" modifier only the domain's type is used, not its NOT NULL setting, CHECK constraint and/or default value.

Example:

```
execute block (a my_domain = ?, b type of my_other_domain = ?)
returns (p my_third_domain)
as
  declare s_temp type of my_third_domain;
begin
  ...
  ...
end
```

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

INSERT

Available in: DSQL, ESQL, PSQL

Description: Adds rows to a database table, or to one or more tables underlying a view. Field values can be given in the VALUES clause, they can be totally absent (in both cases, exactly one row is inserted), or they can come from a SELECT statement (0 to many rows inserted).

Syntax:

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

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

Restrictions

- The TRANSACTION directive is only available in ESQL.
- The RETURNING clause is not available in ESQL.
- The "INTO *<variables>*" subclause is only available in PSQL.
- When returning values into the context variable NEW, this name must not be preceded by a colon (":").
- Since v. 2.0, no column may appear more than once in the insert list.

INSERT ... DEFAULT VALUES

Available in: DSQL, PSQL

Added in: 2.1

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

Example:

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

RETURNING clause

Available in: DSQL, PSQL

Added in: 2.0

Changed in: 2.1

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

Examples:

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

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

Notes:

- RETURNING is only supported for VALUES inserts and – since version 2.1 – singleton SELECT inserts.
- In DSQL, a statement with a RETURNING clause *always* returns exactly one row. If no record was actually inserted, the fields in this row are all `NULL`. This behaviour may change in a later version of Firebird. In PSQL, if no row was inserted, nothing is returned, and the receiving variables keep their existing values.

UNION allowed in feeding SELECT

Changed in: 2.0

Description: A SELECT query used in an INSERT statement may now be a UNION.

Example:

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

MERGE

Available in: DSQL, PSQL

Added in: 2.1

Description: Merges data into a table or view. The source may a table, view or derived table (i.e. a parenthesized SELECT statement or CTE). Each source record will be used to update one or more target records, insert a new record in the target table, or neither. The action taken depends on the provided condition and the WHEN clause(s). The condition will typically contain a comparison of fields in the source and target relations.

Syntax:

```
MERGE INTO {tablename | viewname} [[AS] alias]
   USING {tablename | viewname | (select_stmt)} [[AS] alias]
   ON condition
   WHEN MATCHED THEN UPDATE SET colname = value [, colname = value ...]
   WHEN NOT MATCHED THEN INSERT [(<columns>)] VALUES (<values>)

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

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

Examples:

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

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

> **Note**
>
> WHEN NOT MATCHED should be interpreted from the point of view of the *source* (the relation in the USING clause). That is: if a source record doesn't have a match in the target table, the INSERT clause is executed. Conversely, records in the target table without a matching source record don't trigger any action.

Warning

If the WHEN MATCHED clause is present and multiple source records match the same record in the target table, the UPDATE clause is executed for all the matching source records, each update overwriting the previous one. This is non-standard behaviour: SQL-2003 specifies that in such a case an exception must be raised.

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 based upon a non-aggregate column that appears in the GROUP BY list (by name or position).

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

COLLATE subclause for text BLOB columns

Added in: 2.0

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

Example:

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

Common Table Expressions ("WITH ... AS ... SELECT")

Available in: DSQL, PSQL

Added in: 2.1

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

Syntax:

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

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

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

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

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

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

Example:

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

Notes:

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

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

Recursive CTEs

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

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

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

The execution pattern of a recursive CTE is as follows:

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

Example with a recursive CTE:

```
with recursive
  dept_year_budget as (
    select fiscal_year,
           dept_no,
           sum(projected_budget) as budget
    from proj_dept_budget
    group by fiscal_year, dept_no
  ),
  dept_tree as (
    select dept_no,
           head_dept,
           department,
           cast('' as varchar(255)) as indent
    from department
    where head_dept is null
    union all
    select d.dept_no,
           d.head_dept,
           d.department,
           h.indent || '  '
    from department d
         join dept_tree h on d.head_dept = h.dept_no
  )
```

```
select d.dept_no,
       d.indent || d.department as department,
       dyb_2008.budget as budget_08,
       dyb_2009.budget as budget_09
from dept_tree d
     left join dept_year_budget dyb_2008
       on d.dept_no = dyb_2008.dept_no
       and dyb_2008.fiscal_year = 2008
     left join dept_year_budget dyb_2009
       on d.dept_no = dyb_2009.dept_no
       and dyb_2009.fiscal_year = 2009
```

Notes on recursive CTEs:

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

Derived tables ("SELECT FROM SELECT")

Added in: 2.0

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

Syntax:

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

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

Examples:

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

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

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

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

Notes:

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

FIRST and SKIP

Available in: DSQL, PSQL

Added in: 1.0

Changed in: 1.5

Better alternative: ROWS

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

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

Syntax:

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

Two Gotchas with FIRST in subselects

- This:

  ```
  delete from MyTable where ID in (select first 10 ID from 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! Or better: use the ROWS syntax, available since Firebird 2.0.

- Queries like:

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

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

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

GROUP BY

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

Syntax:

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

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

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

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

Grouping by alias, position and expressions

Changed in: 1.0, 1.5, 2.0

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

Examples:

These three queries all achieve the same result:

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

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

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

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

HAVING: Stricter rules

Changed in: 1.5

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

JOIN

Ambiguous field names rejected

Changed in: 1.0

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

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

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

CROSS JOIN

Added in: 2.0

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

Syntax:

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

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

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

Example:

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

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

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

Named colums JOIN

Added in: 2.1

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

Syntax:

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

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

Example:

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

The equivalent in traditional syntax:

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

Notes:

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

Natural JOIN

Added in: 2.1

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

Syntax:

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

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

Example:

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

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

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

Notes:

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

ORDER BY

Syntax:

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

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

Order by colum alias

Added in: 2.0

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

Example:

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

Ordering by column position causes * expansion

Changed in: 2.0

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

Examples:

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

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

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

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

Ordering by expressions

Added in: 1.5

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

Example:

```
select x, y, note from Pairs
order by x+y desc
```

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

Notes:

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

- Only non-negative whole number *literals* are interpreted as column positions. A whole number resulting from an expression evaluation or parameter substitution is seen as an integer invariable and will lead to a dummy sort, since its value is the same for each row.

NULLS placement

Changed in: 1.5, 2.0

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

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

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

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

Table 6.1. NULLs placement in ordered columns

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

Notes

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

Examples:

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

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

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

Stricter ordering rules with aggregate statements

Changed in: 1.5

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

PLAN

Available in: DSQL, ESQL, PSQL

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

Syntax:

```
PLAN <plan_expr>

<plan_expr>   ::=  [JOIN | [SORT] [MERGE]] (<plan_item> [, <plan_item> ...])

<plan_item>   ::=  <basic_item> | <plan_expr>

<basic_item>  ::=  {table | alias}
                    {NATURAL
                     | INDEX (<indexlist>))
                     | ORDER index [INDEX (<indexlist>)]}

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

Handling of user PLANs improved

Changed in: 2.0

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

- Plan fragments are propagated to nested levels of joins, enabling manual optimization of complex outer joins.
- User-supplied plans will be checked for correctness in outer joins.
- Short-circuit optimization for user-supplied plans has been added.
- A user-specified access path can be supplied for any SELECT-based statement or clause.

ORDER with INDEX

Changed in: 2.0

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

Example:

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

PLAN must include all tables

Changed in: 2.0

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

Relation alias makes real name unavailable

Changed in: 2.0

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

Examples:

Correct usage:

```
select pears from Fruit
```

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

```
select pears from Fruit F
```

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

No longer possible:

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

ROWS

Available in: DSQL, PSQL

Added in: 2.0

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

Syntax:

With a single SELECT:

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

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

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

With a UNION:

```
SELECT [FIRST p] [SKIP q] <columns> FROM ...
   [WHERE ...]
   [ORDER BY ...]

UNION [ALL | DISTINCT]

SELECT [FIRST r] [SKIP s] <columns> FROM ...
   [WHERE ...]
   [ORDER BY ...]

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

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

Points to note:

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

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

Points to note when using two arguments:

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

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

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

When used with a UNION, the ROWS subclause applies to the UNION as a whole and must be placed after the last SELECT. If you want to limit the output of one or more individual SELECTs within the UNION, you have two options: either use FIRST/SKIP on those SELECT statements, or convert them to derived tables with ROWS clauses.

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

UNION

Available in: DSQL, ESQL, PSQL

UNIONs in subqueries

Changed in: 2.0

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

Example:

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

UNION DISTINCT

Added in: 2.0

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

Syntax:

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

Example:

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

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

WITH LOCK

Available in: DSQL, PSQL

Added in: 1.5

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

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

This is for experts only!

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

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

Syntax:

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

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

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

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

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

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

UPDATE

Available in: DSQL, ESQL, PSQL

Description: Changes values in a table (or in one or more tables underlying a view). The columns affected are specified in the SET clause; the rows affected may be limited by the WHERE and ROWS clauses.

Syntax:

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

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

Restrictions

- The TRANSACTION directive is only available in ESQL.
- In a pure DSQL session, WHERE CURRENT OF isn't of much use, since there exists no DSQL statement to create a cursor.
- The PLAN, ORDER BY and ROWS clauses are not available in ESQL.
- Since v. 2.0, no column may be SET more than once in the same UPDATE statement.
- The RETURNING clause is not available in ESQL.
- The "INTO *<variables>*" subclause is only available in PSQL.
- When returning values into the context variable NEW, this name must not be preceded by a colon (":").

COLLATE subclause for text BLOB columns

Added in: 2.0

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

Example:

```
update MyTable
  set NameBlobSp = 'Juan'
  where NameBlobBr collate pt_br = 'João'
```

ORDER BY

Available in: DSQL, PSQL

Added in: 2.0

Description: UPDATE now allows an ORDER BY clause. This only makes sense in combination with ROWS, but is also valid without it.

PLAN

Available in: DSQL, PSQL

Added in: 2.0

Description: UPDATE now allows a PLAN clause, so users can optimize the operation manually.

Relation alias makes real name unavailable

Changed in: 2.0

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

Examples:

Correct usage:

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

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

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

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

No longer possible:

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

RETURNING

Available in: DSQL, PSQL

Added in: 2.1

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

Example:

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

Notes:

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

ROWS

Available in: DSQL, PSQL

Added in: 2.0

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

Syntax:

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

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

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

Points to note:

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

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

Points to note when using two arguments:

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

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

UPDATE OR INSERT

Available in: DSQL, PSQL

Added in: 2.1

Description: UPDATE OR INSERT checks if any existing records already contain the new values supplied for the MATCHING columns. If so, those records are updated. If not, a new record is inserted. In the absence of a MATCHING clause, matching is done against the primary key. If a RETURNING clause is present and more than one matching record is found, an error is raised.

Syntax:

```
UPDATE OR INSERT INTO
   {tablename | viewname} [(<columns>)]
   VALUES (<values>)
   [MATCHING (<columns>)]
   [RETURNING <values> [INTO <variables>]]

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

Restrictions

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

Example:

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

Notes:

- Matches are determined with IS NOT DISTINCT, not with the "=" operator. This means that one `NULL` matches another.
- The optional RETURNING clause:
  - ...may contain any or all columns of the target table, regardless if they were mentioned earlier in the statement, but also other expressions.
  - ...may contain OLD and NEW qualifiers for field names; by default, the new field value is returned.
  - ...returns field values as they are after the BEFORE triggers have run, but before any AFTER triggers.

Chapter 7

Transaction control statements

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

Available in: DSQL, ESQL

Syntax:

```
ROLLBACK [WORK]
   [TRANSACTION tr_name]
   [RETAIN [SNAPSHOT] | TO [SAVEPOINT] sp_name | RELEASE]
```

- The TRANSACTION clause is only available in ESQL.
- The RELEASE clause is only available in ESQL, and is discouraged.
- RETAIN and TO are only available in DSQL.

ROLLBACK RETAIN

Available in: DSQL

Added in: 2.0

Description: Undoes all the database changes carried out in the transaction without closing it. User variables set with RDB$SET_CONTEXT() remain unchanged.

Syntax:

```
ROLLBACK [WORK] RETAIN [SNAPSHOT]
```

> **Note**
>
> The functionality provided by ROLLBACK RETAIN has been present since InterBase 6, but the only way to access it was through the API call `isc_rollback_retaining()`.

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 database mutations performed within the transaction since the savepoint was created are undone. User variables set with RDB$SET_CONTEXT() remain unchanged.
- 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 database mutations performed within the transaction since the savepoint was created are undone. User variables set with RDB$SET_CONTEXT() remain unchanged.
- 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 specify the NO AUTO UNDO option in your SET TRANSACTION statement, or – if you use the API – set the TPB flag `isc_tpb_no_auto_undo`. Both prevent the creation of the transaction-level savepoint.

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 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.

SET TRANSACTION

Available in: DSQL, ESQL

Changed in: 2.0

Description: Starts and optionally configures a transaction.

Syntax:

```
SET TRANSACTION
   [NAME hostvar]
   [READ WRITE | READ ONLY]
   [ [ISOLATION LEVEL] { SNAPSHOT [TABLE STABILITY]
                       | READ COMMITTED [[NO] RECORD_VERSION] } ]
   [WAIT | NO WAIT]
   [LOCK TIMEOUT seconds]
   [NO AUTO UNDO]
   [IGNORE LIMBO]
   [RESERVING <tables> | USING <dbhandles>]

<tables>      ::=  <table_spec> [, <table_spec> ...]

<table_spec>  ::=  tablename [, tablename ...]
                       [FOR [SHARED | PROTECTED] {READ | WRITE}]

<dbhandles>   ::=  dbhandle [, dbhandle ...]
```

- The NAME option is only available in ESQL. It must be followed by a previously declared and initialized host-language variable. Without NAME, SET TRANSACTION applies to the default transaction.

- The USING option is also ESQL-only. It limits the databases that the transaction can access to the ones mentioned here.
- IGNORE LIMBO and LOCK TIMEOUT are not supported in ESQL.
- LOCK TIMEOUT and NO WAIT are mutually exclusive.
- Default option settings are: READ WRITE + WAIT + SNAPSHOT.

IGNORE LIMBO

Available in: DSQL

Added in: 2.0

Description: With this option, records created by limbo transactions are ignored. Transactions are in limbo if the second stage of a two-phase commit fails.

> **Note**
>
> IGNORE LIMBO surfaces the `isc_tpb_ignore_limbo` TPB parameter, available in the API since InterBase times and mainly used by gfix.

LOCK TIMEOUT

Available in: DSQL

Added in: 2.0

Description: This option is only available for WAIT transactions. It takes a non-negative integer as argument, prescribing the maximum number of seconds that the transaction should wait when a lock conflict occurs. If the the waiting time has passed and the lock has still not been released, an error is generated.

> **Note**
>
> This is a brand new feature in Firebird 2. Its API equivalent is the new `isc_tpb_lock_timeout` TPB parameter.

NO AUTO UNDO

Available in: DSQL, ESQL

Added in: 2.0

Description: With NO AUTO UNDO, the transaction refrains from keeping the log that is normally used to undo changes in the event of a rollback. Should the transaction be rolled back after all, other transactions will pick up the garbage (eventually). This option can be useful for massive insertions that don't need to be rolled back. For transactions that don't perform any mutations, NO AUTO UNDO makes no difference at all.

Note

NO AUTO UNDO is the SQL equivalent of the `isc_tpb_no_auto_undo` TPB parameter, available in the API since InterBase times.

Chapter 8

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

Better alternative: LEAVE

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, use of the SQL-99 compliant alternative LEAVE is preferred.

CLOSE cursor

Available in: PSQL

Added in: 2.0

Description: Closes an open cursor. Any cursors still open when the trigger, stored procedure or EXECUTE BLOCK statement they belong to is exited, will be closed automatically.

Syntax:

```
CLOSE cursorname;
```

Example: See DECLARE ... CURSOR.

DECLARE

Available in: PSQL

Description: Declares a PSQL local variable.

Syntax:

```
DECLARE [VARIABLE] varname <var_spec>;

<var_spec>  ::=  <type> [NOT NULL] [<coll>] [<default>]
                    | CURSOR FOR (select-statement)
<type>      ::=  sql_datatype | [TYPE OF] domain
<coll>      ::=  COLLATE collation
<default>   ::=  {= | DEFAULT} value
```

- If *sql_datatype* is a text type, it may include a character set.
- Obviously, a COLLATE clause is only allowed with text types.

DECLARE ... CURSOR

Added in: 2.0

Description: Declares a named cursor and binds it to its own SELECT statement. The cursor can later be opened, used to walk the result set, and closed again. Positioned updates and deletes (using WHERE CURRENT OF) are also supported. PSQL cursors are available in triggers, stored procedures and EXECUTE BLOCK statements.

Example:

```
execute block
returns (relation char(31), sysflag int)
as
declare cur cursor for
  (select rdb$relation_name, rdb$system_flag from rdb$relations);
begin
  open cur;
  while (1=1) do
  begin
    fetch cur into relation, sysflag;
    if (row_count = 0) then leave;
    suspend;
  end
  close cur;
end
```

Notes:

- A "FOR UPDATE" clause is allowed in the SELECT statement, but not required for a positioned update or delete to succeed.
- Make sure that declared cursor names do not clash with any names defined later on in AS CURSOR clauses.
- If you need a cursor to loop through an output set, it is almost always easier – and less error-prone – to use a FOR SELECT statement with an AS CURSOR clause. Declared cursors must be explicitly opened, fetched from, and closed. Furthermore, you need to check `row_count` after every fetch and break out of the loop if it is zero. AS CURSOR takes care of all of that automagically. However, declared cursors give you more control over the sequence of events, and allow you to operate several cursors in parallel.
- The SELECT statement may contain named SQL parameters, like in "`select name || :sfx from names where number = :num`". Each parameter must be a PSQL variable that has been declared previously (this includes any in/out params of the PSQL module). When the cursor is opened, the parameter is assigned the current value of the variable.
- Caution! If the value of a PSQL variable that is used in the SELECT statement changes during execution of the loop, the statement *may* (but will not always) be re-evaluated for the remaining rows. In general, this situation should be avoided. If you really need this behaviour, test your code thoroughly and make sure you know how variable changes affect the outcome. Also be advised that the behaviour may depend on the query

plan, in particular the use of indices. As it is currently not strictly defined, it may change in some future version of Firebird.

See also: OPEN cursor, FETCH cursor, CLOSE cursor

DECLARE [VARIABLE] with initialization

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.

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
```

DECLARE with DOMAIN instead of datatype

Added in: 2.1

Description: In Firebird 2.1 and above, PSQL local variables and input/output parameters can be declared with a domain instead of a datatype. The TYPE OF modifier allows using only the domain's datatype and not its NOT NULL setting, CHECK constraint and/or default value.

Example:

```
create procedure MyProc (a int, f ternbool)
  returns (b int, x type of bigfloat)
as
  declare p int;
  declare q int = 8;
  declare y stocknum default -1;
begin
  <very intelligent code here>
end
```

(This example presupposes that TERNBOOL, BIGFLOAT and STOCKNUM are domains already defined in the database.)

Warning

If you change a domain's definition, existing PSQL code using that domain may become invalid. If this happens, the system table field RDB$VALID_BLR will be set to 0 for any procedure or trigger whose code is no longer valid. If you have changed a domain, the following query will find the code modules that depend on it and report the state of RDB$VALID_BLR:

```
select * from (
  select 'Procedure', rdb$procedure_name, rdb$valid_blr from rdb$procedures
    union
  select 'Trigger', rdb$trigger_name, rdb$valid_blr from rdb$triggers
) (type, name, valid)
where exists
  (select * from rdb$dependencies
   where rdb$dependent_name = name and rdb$depended_on_name = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name. Use all-caps if the domain
   was created case-insensitively. Otherwise, use the exact capitalisation. */
```

Unfortunately, not all PSQL invalidations will be reflected in the RDB$VALID_BLR field. It is therefore advisable to look at all the procedures and triggers reported by the above query, even those having a 1 in the "VALID" column.

Please notice that for PSQL modules inherited from earlier Firebird versions (including a number of system triggers, even if the database was created under Firebird 2.1 or higher), RDB$VALID_BLR is NULL. This does *not* indicate that their BLR is invalid.

The isql commands SHOW PROCEDURES and SHOW TRIGGERS flag modules whose RDB$VALID_BLR field is zero with an asterisk. SHOW PROCEDURE *PROCNAME* and SHOW TRIGGER *TRIGNAME*, which display individual PSQL modules, do not signal invalid BLR.

COLLATE in variable declaration

Added in: 2.1

Description: In Firebird 2.1 and above, a COLLATE clause is allowed in the declaration of text-type PSQL local variables and input/output parameters.

Example:

```
create procedure GimmeText
  returns (txt char(32) character set utf8 collate unicode)
as
  declare simounao mytextdomain collate pt_br default 'não';
begin
  <stunningly intelligent code here>
end
```

NOT NULL in variable declaration

Added in: 2.1

Description: In Firebird 2.1 and above, a NOT NULL constraint is allowed in the declaration of PSQL local variables and input/output parameters.

Example:

```
create procedure Compute(a int not null, b int not null)
  returns (outcome bigint not null)
as
  declare temp bigint not null;
begin
  <rather disappointing 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;
```

> **Note**
>
> Starting at version 2.0, the maximum message length is 1021 instead of 78 characters.

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 DSQL 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. The submitted DSQL statement is always executed with the privileges of the current **user**. Privileges granted to the trigger or SP that contains the EXECUTE STATEMENT statement are not in effect while the DSQL statement runs.

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.

FETCH cursor

Available in: PSQL

Added in: 2.0

Description: Fetches the next data row from a cursor's result set and stores the column values in PSQL variables.

Syntax:

```
FETCH cursorname INTO [:]varname [, [:]varname ...];
```

Notes:

- The ROW_COUNT context variable will be 1 if the fetch returned a data row and 0 if the end of the set has been reached.
- You can do a positioned UPDATE or DELETE on the fetched row with the WHERE CURRENT OF clause.

Example: See DECLARE ... CURSOR.

FOR EXECUTE STATEMENT ... DO

Available in: PSQL

Added in: 1.5

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

FOR SELECT ... INTO ... DO

Available in: PSQL

Description: Executes a SELECT statement and retrieves the result set. In each iteration of the loop, the field values of the current row are copied into local variables. Adding an AS CURSOR clause enables positioned deletes and updates. FOR SELECT statements may be nested.

Syntax:

```
FOR <select-stmt>
   INTO <var> [, <var> ...]
   [AS CURSOR name]
DO
   <psql-stmt>

<select-stmt>  ::=  A valid SELECT statement.
<var>          ::=  A PSQL variable name, optionally preceded by ":"
<psql-stmt>    ::=  A single statement or a block of PSQL code.
```

- The SELECT statement may contain named SQL parameters, like in "`select name || :sfx from names where number = :num`". Each parameter must be a PSQL variable that has been declared previously (this includes any in/out params of the PSQL module).
- Caution! If the value of a PSQL variable that is used in the SELECT statement changes during execution of the loop, the statement *may* (but will not always) be re-evaluated for the remaining rows. In general, this situation should be avoided. If you really need this behaviour, test your code

thoroughly and make sure you know how variable changes affect the outcome. Also be advised that the behaviour may depend on the query plan, in particular the use of indices. And as it is currently not strictly defined, it may also change in some future version of Firebird.

Examples:

```
create procedure shownums
  returns (aa int, bb int, sm int, df int)
as
begin
  for select distinct a, b from numbers order by a, b
    into :aa, :bb
  do
  begin
    sm = aa + bb;
    df = aa - bb;
    suspend;
  end
end
```

```
create procedure relfields
  returns (relation char(32), pos int, field char(32))
as
begin
  for select rdb$relation_name from rdb$relations
      into :relation
  do
  begin
    for select rdb$field_position + 1, rdb$field_name
        from rdb$relation_fields
        where rdb$relation_name = :relation
        order by rdb$field_position
        into :pos, :field
    do
    begin
      if (pos = 2) then relation = '  "';   -- for nicer output
      suspend;
    end
  end
end
```

AS CURSOR clause

Available in: PSQL

Added in: IB

Description: The optional AS CURSOR clause creates a named cursor that can be referenced (after WHERE CURRENT OF) within the FOR SELECT loop in order to update or delete the current row. This feature was already added in InterBase, but not mentioned in the *Language Reference*.

Example:

```
create procedure deltown (towntodelete varchar(24))
  returns (town varchar(24), pop int)
```

```
as
begin
  for select town, pop from towns into :town, :pop as cursor tcur do
  begin
    if (town = towntodelete)
      then delete from towns where current of tcur;
      else suspend;
  end
end
```

Notes:

- A "FOR UPDATE" clause is allowed in the SELECT statement., but not required for a positioned update or delete to succeed.
- Make sure that cursor names defined here do not clash with any names created earlier on in DECLARE CURSOR statements.
- AS CURSOR is not supported in FOR EXECUTE STATEMENT loops, even if the statement to execute is a suitable SELECT query.

LEAVE

Available in: PSQL

Added in: 1.5

Changed in: 2.0

Description: LEAVE immediately terminates the innermost WHILE or FOR loop. With the optional *label* argument introduced in Firebird 2.0, LEAVE can break out of surrounding loops as well. Execution continues with the first statement after the outermost terminated loop.

Syntax:

```
[label:]
{FOR | WHILE} ... DO
   ...
   (possibly nested loops, with or without labels)
   ...
   LEAVE [label];
```

Example:

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

```
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;
```

The next example uses labels. "`Leave LoopA`" terminates the outer loop, "`leave LoopB`" the inner loop. Notice that a plain "`leave`" would also suffice to terminate the inner loop.

```
stmt1 = 'select Name from Farms';
LoopA:
for execute statement :stmt1 into :farm do
begin
  stmt2 = 'select Name from Animals where Farm = ''';
  LoopB:
  for execute statement :stmt2 || :farm || '''' into :animal do
  begin
    if  (animal = 'Fluffy') then leave LoopB;
    else if (animal = farm) then leave LoopA;
    else suspend;
  end
end
```

OPEN cursor

Available in: PSQL

Added in: 2.0

Description: Opens a previously declared cursor, executing its SELECT statement and enabling it to fetch records from the result set.

Syntax:

```
OPEN cursorname;
```

Example: See DECLARE ... CURSOR.

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.

UDFs callable as void functions

Changed in: 2.0

Description: In Firebird 2.0 and above, PSQL code may call UDFs without assigning the result value, i.e. like a Pascal procedure or C void function. In most cases this is senseless, because the main purpose of almost every UDF is to produce the result value. Some functions however perform a specific task, and if you're not interested in the result value you can now spare yourself the trouble of assigning it to a dummy variable.

> **Note**
>
> `RDB$GET_CONTEXT` and `RDB$SET_CONTEXT`, though classified in this guide under internal functions, are actually a kind of auto-declared UDFs. You may therefore call them without catching the result. Of course this only makes sense for `RDB$SET_CONTEXT`.

WHERE CURRENT OF valid again for view cursors

Changed in: 2.0, 2.1

Description: Because of possible reliability issues, Firebird 2.0 disallowed WHERE CURRENT OF for view cursors. In Firebird 2.1, with its improved view validation logic, this restriction has been lifted.

Chapter 9

Context variables

CURRENT_CONNECTION

Available in: DSQL, PSQL

Added in: 1.5

Changed in: 2.1

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

Type: INTEGER

Examples:

```
select current_connection from rdb$database
```

```
execute procedure P_Login(current_connection)
```

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

CURRENT_ROLE

Available in: DSQL, PSQL

Added in: 1.0

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

Type: VARCHAR(31)

Example:

```
if (current_role <> 'MANAGER')
  then exception only_managers_may_delete;
```

```
else
  delete from Customers where custno = :custno;
```

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

CURRENT_TIME

Available in: DSQL, PSQL, ESQL

Changed in: 2.0

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

Type: TIME

Syntax:

```
CURRENT_TIME [(precision)]

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

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

Examples:

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

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

Notes:

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

CURRENT_TIMESTAMP

Available in: DSQL, PSQL, ESQL

Changed in: 2.0

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

Type: TIMESTAMP

Syntax:

```
CURRENT_TIMESTAMP [(precision)]

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

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

Examples:

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

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

Notes:

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

CURRENT_TRANSACTION

Available in: DSQL, PSQL

Added in: 1.5

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

Type: INTEGER

Examples:

```
select current_transaction from rdb$database
```

```
New.Txn_ID = current_transaction;
```

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

CURRENT_USER

Available in: DSQL, PSQL

Added in: 1.0

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

Type: VARCHAR(31)

Example:

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

DELETING

Available in: PSQL

Added in: 1.5

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

Type: boolean

Example:

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

GDSCODE

Available in: PSQL

Added in: 1.5

Changed in: 2.0

Description: In a WHEN GDSCODE handling block, the `GDSCODE` context variable contains a numerical representation of the current Firebird error code. Starting with Firebird 2.0, the same is true in a WHEN ANY block if

its execution was triggered by a Firebird error; otherwise it contains 0. `GDSCODE` is also 0 in WHEN SQLCODE and WHEN EXCEPTION 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
```

NEW

Available in: PSQL, triggers only

Changed in: 1.5, 2.0

Description: `NEW` contains the new version of a database record that has just been inserted or updated. Starting with Firebird 2.0 it is read-only in AFTER triggers.

Type: Data row

> **Note**
>
> In multi-action triggers – introduced in Firebird 1.5 – `NEW` is always available. But if the trigger is fired by a DELETE, there will be no new version of the record. In that situation, reading from `NEW` will always return `NULL`; writing to it will cause a runtime exception.

'NOW'

Available in: DSQL, PSQL, ESQL

Changed in: 2.0

Description: `'NOW'` is not a variable but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the current date and/or time. The fractional part of the time used to be always "`.0000`", giving an effective seconds precision. Since Firebird 2.0 the precision is 3 decimals, i.e. milliseconds. `'NOW'` is case-insensitive, and the engine ignores leading or trailing spaces when casting.

Type: CHAR(3)

Examples:

```
select 'Now' from rdb$database
-- returns 'Now'
```

```
select cast('Now' as date) from rdb$database
-- returns e.g. 2008-08-13
```

```
select cast('now' as time) from rdb$database
-- returns e.g. 14:20:19.6170
```

```
select cast('NOW' as timestamp) from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170
```

Shorthand syntax for the last three statements:

```
select date 'Now' from rdb$database
select time 'now' from rdb$database
select timestamp 'NOW' from rdb$database
```

Notes:

- `'NOW'` always returns the actual date/time, even in PSQL modules, where `CURRENT_DATE`, `CURRENT_TIME` and `CURRENT_TIMESTAMP` return the same value throughout the duration of the outermost routine. This makes `'NOW'` useful for measuring time intervals in triggers, procedures and executable blocks.
- Except in the situation mentioned above, reading `CURRENT_DATE`, `CURRENT_TIME` and `CURRENT_TIMESTAMP` is generally preferable to casting `'NOW'`. Be aware though that `CURRENT_TIME` defaults to seconds precision; to get milliseconds precision, use `CURRENT_TIME(3)`.

OLD

Available in: PSQL, triggers only

Changed in: 1.5, 2.0

Description: `OLD` contains the existing version of a database record just before a deletion or update. Starting with Firebird 2.0 it is read-only.

Type: Data row

> **Note**
>
> In multi-action triggers – introduced in Firebird 1.5 – `OLD` is always available. But if the trigger is fired by an INSERT, there is obviously no pre-existing version of the record. In that situation, reading from `OLD` will always return `NULL`; writing to it will cause a runtime exception.

ROW_COUNT

Available in: PSQL

Added in: 1.5

Changed in: 2.0

Description: The `ROW_COUNT` context variable contains the number of rows affected by the most recent DML statement (INSERT, UPDATE, DELETE, SELECT or FETCH) in the current trigger, stored procedure or executable block.

Type: INTEGER

Example:

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

Behaviour with SELECT and FETCH:

- After a singleton SELECT, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise.
- In a FOR SELECT loop, `ROW_COUNT` is incremented with every iteration (starting at 0 before the first).
- After a FETCH from a cursor, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise. Fetching more records from the same cursor does *not* increment `ROW_COUNT` beyond 1.
- In Firebird 1.5.*x*, `ROW_COUNT` is 0 after any type of SELECT statement.

> **Note**
>
> `ROW_COUNT` cannot be used to determine the number of rows affected by an EXECUTE STATEMENT or EXECUTE PROCEDURE command.

SQLCODE

Available in: PSQL

Added in: 1.5

Description: In a WHEN SQLCODE handling block, the `SQLCODE` context variable contains the current SQL error code. The same is true in a WHEN ANY block if its execution was triggered by an SQL error; 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 10

Operators and predicates

NULL literals allowed as operands

Changed in: 2.0

Description: Before Firebird 2.0, most operators and predicates did not allow NULL literals as operands. Tests or operations like "A <> NULL", "B + NULL" or "NULL < ANY(...)" would be rejected by the parser. Now they are allowed almost everywhere, but please be aware of the following:

> *The vast majority of these newly allowed expressions return NULL regardless of the state or value of the other operand, and are therefore worthless for any practicle purpose whatsoever.*

In particular, don't try to determine (non-)nullness of a field or variable by testing with "= NULL" or "<> NULL". Always use "IS [NOT] NULL".

Predicates: The IN, ANY/SOME and ALL predicates now also allow NULL literals where they were previously taboo. Here too, there is no practical benefit to enjoy, but the situation is a little more complicated in that predicates with NULLs do not always return a NULL result. For details, see the *Firebird Null Guide*, section *Predicates*.

|| (string concatenator)

Available in: DSQL, ESQL, PSQL

Text BLOB concatenation

Changed in: 2.1

Description: Since Firebird 2.1 the concatenation operator supports BLOBs of any length and any character set. If a mixture of BLOBs and non-BLOBs is involved, the result is a BLOB. If both text and binary BLOBs are involved, the result is a binary BLOB.

Result type VARCHAR or BLOB

Changed in: 2.0, 2.1

Description: Before Firebird 2.0, the result type of string concatenations used to be CHAR(*n*). In Firebird 2.0 this was changed to VARCHAR(*n*). As a result, the maximum length of a concatenation outcome became 32765

instead of 32767. In Firebird 2.1 and up, if at least one of the operands is a BLOB, the result is also a BLOB and the maximum doesn't apply. For non-BLOB concatenations the result is still VARCHAR(*n*) with a maximum of 32765 bytes.

Overflow checking

Changed in: 1.0, 2.0

Description: In Firebird versions 1.x, an error would be raised if the sum of the *declared* string lengths in a concatenation exceeded 65535 bytes, even if the *actual* result lay within the maximum string length of 32767 bytes. In Firebird 2.0 and up, the declared string lengths will never cause an error. Only if the actual outcome exceeds 32765 bytes (the new limit for concatenation results) will an error be raised.

ALL

Available in: DSQL, ESQL, PSQL

NULL literals allowed

Changed in: 2.0

Description: The ALL predicate now allows a `NULL` as the test value. Notice that this brings no practical benefits. In particular, a `NULL` test value will not be considered equal to `NULL`s in the subquery result set. Even if the entire set is filled with `NULL`s and the operator chosen is "=", the predicate will not return `true`, but `NULL`.

UNION as subselect

Changed in: 2.0

Description: The subselect in an ALL predicate may now also be a UNION.

ANY / SOME

Available in: DSQL, ESQL, PSQL

NULL literals allowed

Changed in: 2.0

Description: The ANY (or SOME) predicate now allows a `NULL` as the test value. Notice that this brings no practical benefits. In particular, a `NULL` test value will not be considered equal to a `NULL` in the subquery result set.

UNION as subselect

Changed in: 2.0

Description: The subselect in an ANY (or SOME) predicate may now also be a UNION.

IN

Available in: DSQL, ESQL, PSQL

NULL literals allowed

Changed in: 2.0

Description: The IN predicate now allows `NULL` literals, both as the test value and in the list. Notice that this brings no practical benefits. In particular, "`NULL` IN (..., `NULL`, ..., ...)" will not return `true` and "`NULL` NOT IN (..., `NULL`, ..., ...)" will not return `false`.

UNION as subselect

Changed in: 2.0

Description: A subselect in an IN predicate may now also be a UNION.

IS [NOT] DISTINCT FROM

Available in: DSQL, PSQL

Added in: 2.0

Description: Two operands are considered DISTINCT if they have a different value or if one of them is `NULL` and the other isn't. They are NOT DISTINCT if they have the same value or if both of them are `NULL`.

Result type: Boolean

Syntax:

```
op1 IS [NOT] DISTINCT FROM op2
```

Examples:

```
select id, name, teacher from courses
  where start_day is not distinct from end_day
```

```
if (New.Job is distinct from Old.Job)
  then post_event 'job_changed';
```

IS [NOT] DISTINCT FROM always returns `true` or `false`, never `NULL` (unknown). The "=" and "<>" operators, by contrast, return `NULL` if one or both operands are `NULL`. See also the table below.

Table 10.1. Comparison of [NOT] DISTINCT to "=" and "<>"

Operand characteristics	Results with the different operators			
	=	NOT DISTINCT	<>	DISTINCT
Same value	`true`	`true`	`false`	`false`
Different values	`false`	`false`	`true`	`true`
Both `NULL`	`NULL`	`true`	`NULL`	`false`
One `NULL`	`NULL`	`false`	`NULL`	`true`

NEXT VALUE FOR

Available in: DSQL, PSQL

Added in: 2.0

Description: Returns the next value in a sequence. SEQUENCE is the SQL-compliant term for what InterBase and Firebird have always called a generator. NEXT VALUE FOR is fully equivalent to GEN_ID(..., 1) and is the recommended syntax from Firebird 2.0 onward.

Syntax:

```
NEXT VALUE FOR sequence-name
```

Example:

```
new.cust_id = next value for custseq;
```

NEXT VALUE FOR doesn't support increment values other than 1. If you absolutely need other step values, use the legacy GEN_ID function.

See also: CREATE SEQUENCE, GEN_ID()

SOME

See ANY

Chapter 11

Aggregate functions

Aggregate functions operate on groups of records, rather than on individual records or variables. They are often used in combination with a GROUP BY clause.

LIST()

Available in: DSQL, PSQL

Added in: 2.1

Description: LIST returns a string consisting of the non-`NULL` argument values in the group, separated either by a comma or by a user-supplied delimiter. If there are no non-`NULL` values (this includes the case where the group is empty), `NULL` is returned.

Result type: BLOB

Syntax:

```
LIST ([ALL | DISTINCT] expression [, separator])
```

- ALL (the default) results in all non-`NULL` values to be listed. With DISTINCT, duplicates are removed, except if *`expression`* is a BLOB.
- The optional *`separator`* argument may be a string literal, a parameter or a variable in versions up to 2.1.3. Starting at 2.1.4 it may be any string expression. This makes it possible to specify e.g. *`ascii_char(13)`* as a separator.
- The *`expression`* and *`separator`* arguments support BLOBs of any size and character set.
- Date/time and numerical arguments are implicitly converted to strings before concatenation.
- The result is a text BLOB, except when *`expression`* is a BLOB of another subtype.
- The ordering of the list values is undefined.

Bug

In versions 2.1–2.1.3, the last part of the result is sometimes truncated. With a single-row set, this happens when the length gets somewhere above 4000. As the number of rows grows, the threshold climbs rapidly, so in practice this bug might not raise its head very often. It is fixed in 2.1.4.

MAX()

Available in: DSQL, ESQL, PSQL

Added in: IB

Changed in: 2.1

Description: MAX returns the maximum argument value in the group. If the argument is a string, this is the value that comes last when the active collation is applied.

Result type: Varies

Syntax:

```
MAX (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.
- Since Firebird 2.1, this function fully supports text BLOBs of any size and character set.

MIN()

Available in: DSQL, ESQL, PSQL

Added in: IB

Changed in: 2.1

Description: MIN returns the minimum argument value in the group. If the argument is a string, this is the value that comes first when the active collation is applied.

Result type: Varies

Syntax:

```
MIN (expression)
```

- If the group is empty or contains only NULLs, the result is NULL.
- Since Firebird 2.1, this function fully supports text BLOBs of any size and character set.

Chapter 12

Internal functions

ABS()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the absolute value of the argument.

Result type: Numerical

Syntax:

```
ABS (number)
```

Important

If the external function ABS is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ACOS()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the arc cosine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
ACOS (number)
```

- The result is an angle in the range [0, #].
- If the argument is outside the range [-1, 1], NaN is returned.

Important

If the external function `ACOS` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ASCII_CHAR()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the ASCII character corresponding to the number passed in the argument.

Result type: [VAR]CHAR(1) CHARACTER SET NONE

Syntax:

```
ASCII_CHAR (<code>)

<code>  ::=  an integer in the range [0..255]
```

Important

- If the external function `ASCII_CHAR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).
- If you are used to the behaviour of the `ASCII_CHAR` UDF, which returns an empty string if the argument is 0, please notice that the internal function correctly returns a character with ASCII code 0 here.

ASCII_VAL()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the ASCII code of the character passed in.

Result type: SMALLINT

Syntax:

```
ASCII_VAL (ch)

ch  ::=  a [VAR]CHAR or text BLOB of max. 32767 bytes
```

- If the argument is a string with more than one character, the ASCII code of the first character is returned.
- If the argument is an empty string, 0 is returned.

- If the argument is `NULL`, `NULL` is returned.
- If the first character of the argument string is multi-byte, an error is raised. (A bug in Firebird 2.1–2.1.3 causes an error to be raised if *any* character in the string is multi-byte. This is fixed in 2.1.4.)

> **Important**
>
> If the external function `ASCII_VAL` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ASIN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the arc sine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
ASIN (number)
```

- The result is an angle in the range [-#/2, #/2].
- If the argument is outside the range [-1, 1], NaN is returned.

> **Important**
>
> If the external function `ASIN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ATAN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the arc tangent of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
ATAN (number)
```

- The result is an angle in the range <-#/2, #/2>.

Important

If the external function `ATAN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

ATAN2()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the angle whose sine-to-cosine *ratio* is given by the two arguments, and whose sine and cosine *signs* correspond to the signs of the arguments. This allows results across the entire circle, including the angles -#/2 and #/2.

Result type: DOUBLE PRECISION

Syntax:

```
ATAN2 (y, x)
```

- The result is an angle in the range [-#, #].
- If *x* is negative, the result is # if *y* is 0, and -# if *y* is -0.
- If both *y* and *x* are 0, the result is meaningless.

Important

If the external function `ATAN2` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Notes:

- A fully equivalent description of this function is the following: ATAN2(*y*, *x*) is the angle between the positive X-axis and the line from the origin to the point (*x*, *y*). This also makes it obvious that ATAN2(0, 0) is undefined.
- If *x* is greater than 0, ATAN2(*y*, *x*) is the same as ATAN(*y*/*x*).
- If both sine and cosine of the angle are already known, ATAN2(*sin*, *cos*) gives the angle.

BIN_AND()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the result of the bitwise AND operation on the argument(s).

Result type: INTEGER or BIGINT

Syntax:

```
BIN_AND (number [, number ...])
```

> **Important**
>
> If the external function `BIN_AND` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIN_OR()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the result of the bitwise OR operation on the argument(s).

Result type: INTEGER or BIGINT

Syntax:

```
BIN_OR (number [, number ...])
```

> **Important**
>
> If the external function `BIN_OR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIN_SHL()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the first argument bitwise left-shifted by the second argument, i.e. *a* << *b* or *a*·2^*b*.

Result type: BIGINT

Syntax:

```
BIN_SHL (number, shift)
```

BIN_SHR()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the first argument bitwise right-shifted by the second argument, i.e. *a* >> *b* or *a*/2^*b*.

Result type: BIGINT

Syntax:

```
BIN_SHR (number, shift)
```

- The operation performed is an arithmetic right shift (SAR), meaning that the sign of the first operand is always preserved.

BIN_XOR()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the result of the bitwise XOR operation on the argument(s).

Result type: INTEGER or BIGINT

Syntax:

```
BIN_XOR (number [, number ...])
```

> **Important**
>
> If the external function `BIN_XOR` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

BIT_LENGTH()

Available in: DSQL, PSQL

Added in: 2.0

Changed in: 2.1

Description: Gives the length in bits of the input string. For multi-byte character sets, this may be less than the number of characters times 8 times the “formal” number of bytes per character as found in RDB $CHARACTER_SETS.

> **Note**
>
> With arguments of type CHAR, this function takes the entire formal string length (e.g. the declared length of a field or variable) into account. If you want to obtain the “logical” bit length, not counting the trailing spaces, right-TRIM the argument before passing it to BIT_LENGTH.

Result type: INTEGER

Syntax:

```
BIT_LENGTH (str)
```

BLOB support: Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples:

```
select bit_length('Hello!') from rdb$database
  -- returns 48
```

```
select bit_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 64: ü and ß take up one byte each in ISO8859_1
```

```
select bit_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
  -- returns 80: ü and ß take up two bytes each in UTF8
```

```
select bit_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
  -- returns 208: all 24 CHAR positions count, and two of them are 16-bit
```

See also: OCTET_LENGTH(), CHARACTER_LENGTH

CAST()

Available in: DSQL, ESQL, PSQL

Added in: IB

Changed in: 2.0, 2.1

Description: CAST converts an expression to the desired datatype or domain. If the conversion is not possible, an error is raised.

Result type: User-chosen.

Syntax:

```
CAST (expression AS {datatype | [TYPE OF] domain})
```

Shorthand syntax:

Alternative syntax, supported only when casting a string literal to a DATE, TIME or TIMESTAMP:

```
datatype 'date/timestring'
```

This syntax was already available in InterBase, but was never properly documented.

Examples:

A full-syntax cast:

```
select cast ('12' || '-June-' || '1959' as date) from rdb$database
```

A shorthand string-to-date cast:

```
update People set AgeCat = 'Old'
  where BirthDate < date '1-Jan-1943'
```

Notice that you can drop even the shorthand cast from the example above, as the engine will understand from the context (comparison to a DATE field) how to interpret the string:

```
update People set AgeCat = 'Old'
  where BirthDate < '1-Jan-1943'
```

But this is not always possible. The cast below cannot be dropped, otherwise the engine would find itself with an integer to be subtracted from a string:

```
select date 'today' - 7 from rdb$database
```

The following table shows the type conversions possible with CAST.

Table 12.1. Possible CASTs

From	To
Numeric types	Numeric types [VAR]CHAR BLOB
[VAR]CHAR BLOB	[VAR]CHAR BLOB Numeric types DATE TIME TIMESTAMP
DATE TIME	[VAR]CHAR BLOB TIMESTAMP
TIMESTAMP	[VAR]CHAR BLOB DATE TIME

Keep in mind that sometimes information is lost, for instance when you cast a TIMESTAMP to a DATE. Also, the fact that types are CAST-compatible is in itself no guarantee that a conversion will succeed. "CAST(123456789 as SMALLINT)" will definitely result in an error, as will "CAST('Judgement Day' as DATE)".

Casting input fields: Since Firebird 2.0, you can cast statement parameters to a datatype:

```
cast (? as integer)
```

This gives you control over the type of input field set up by the engine. Please notice that with statement parameters, you always need a full-syntax cast – shorthand casts are not supported.

Casting to a domain or its type: Firebird 2.1 and above support casting to a domain or its base type. When casting to a domain, any constraints (NOT NULL and/or CHECK) declared for the domain must be satisfied or the cast will fail. Please be aware that a CHECK passes if it evaluates to `TRUE` *or* `NULL`! So, given the following statements:

```
create domain quint as int check (value >= 5000)
select cast (2000 as quint) from rdb$database     -- (1)
select cast (8000 as quint) from rdb$database     -- (2)
select cast (null as quint) from rdb$database     -- (3)
```

only cast number (1) will result in an error.

When the TYPE OF modifier is used, the expression is cast to the base type of the domain, ignoring any constraints. With domain quint defined as above, the following two casts are equivalent and will both succeed:

```
select cast (2000 as type of quint) from rdb$database
select cast (2000 as int) from rdb$database
```

If TYPE OF is used with a (VAR)CHAR type, its character set and collation are retained:

```
create domain iso20 varchar(20) character set iso8859_1;
create domain dunl20 varchar(20) character set iso8859_1 collate du_nl;
create table zinnen (zin varchar(20));
commit;
insert into zinnen values ('Deze');
insert into zinnen values ('Die');
insert into zinnen values ('die');
insert into zinnen values ('deze');

select cast(zin as type of iso20) from zinnen order by 1;
  -- returns Deze -> Die -> deze -> die

select cast(zin as type of dunl20) from zinnen order by 1;
  -- returns deze -> Deze -> die -> Die
```

Casting BLOBs: Successful casting to and from BLOBs is possible since Firebird 2.1.

CEIL(), CEILING()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the smallest whole number greater than or equal to the argument.

Result type: BIGINT or DOUBLE PRECISION

Syntax:

```
CEIL[ING] (number)
```

> **Important**
>
> If the external function `CEILING` is declared in your database, it will override the internal function CEILING (but not CEIL). To make the internal function available, DROP or ALTER the external function (UDF).

See also: FLOOR()

CHAR_LENGTH(), CHARACTER_LENGTH()

Available in: DSQL, PSQL

Added in: 2.0

Changed in: 2.1

Description: Gives the length in characters of the input string.

> **Note**
>
> With arguments of type CHAR, this function returns the formal string length (i.e. the declared length of a field or variable). If you want to obtain the "logical" length, not counting the trailing spaces, right-TRIM the argument before passing it to CHAR[ACTER]_LENGTH.

Result type: INTEGER

Syntax:

```
CHAR_LENGTH (str)
CHARACTER_LENGTH (str)
```

BLOB support: Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples:

```
select char_length('Hello!') from rdb$database
  -- returns 6
```

```
select char_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 8
```

```
select char_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
  -- returns 8; the fact that ü and ß take up two bytes each is irrelevant
```

```
select char_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
  -- returns 24: all 24 CHAR positions count
```

See also: BIT_LENGTH(), OCTET_LENGTH

COALESCE()

Available in: DSQL, PSQL

Added in: 1.5

Description: The COALESCE function takes two or more arguments and returns the value of the first non-`NULL` argument. If all the arguments evaluate to `NULL`, the result is `NULL`.

Result type: Depends on input.

Syntax:

```
COALESCE (<exp1>, <exp2> [, <expN> ... ])
```

Example:

```
select
  coalesce (Nickname, FirstName, 'Mr./Mrs.') || ' ' || LastName
  as FullName
from Persons
```

This example picks the Nickname from the Persons table. If it happens to be `NULL`, it goes on to FirstName. If that too is `NULL`, “Mr./Mrs.” is used. Finally, it adds the family name. All in all, it tries to use the available data to compose a full name that is as informal as possible. Notice that this scheme only works if absent nicknames and first names are really `NULL`: if one of them is an empty string instead, COALESCE will happily return that to the caller.

> **Note**
>
> In Firebird 1.0.x, where COALESCE is not available, you can accomplish the same with the `*nvl` external functions.

COS()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns an angle's cosine. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
COS (angle)
```

- Any non-`NULL` result is – obviously – in the range [-1, 1].

> **Important**
>
> If the external function `COS` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

COSH()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the hyperbolic cosine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
COSH (number)
```

- Any non-`NULL` result is in the range [1, INF].

> **Important**
>
> If the external function `COSH` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

COT()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns an angle's cotangent. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
COT (angle)
```

> **Important**
>
> If the external function `COT` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

DATEADD()

Available in: DSQL, PSQL

Added in: 2.1

Description: Adds the specified number of years, months, days, hours, minutes, seconds or milliseconds to a date/time value.

Result type: DATE, TIME or TIMESTAMP

Syntax:

```
DATEADD (<args>)

<args>      ::=  <amount> <unit> TO <datetime>
                 | <unit>, <amount>, <datetime>

<amount>    ::=  an integer expression (negative to subtract)
<unit>      ::=  YEAR | MONTH | DAY
                    | HOUR | MINUTE | SECOND | MILLISECOND
<datetime>  ::=  a DATE, TIME or TIMESTAMP expression
```

- The result type is determined by the third argument.
- With DATE arguments, only YEAR, MONTH and DAY can be used.
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Examples:

```
dateadd (28 day to current_date)
dateadd (-6 hour to current_time)
dateadd (month, 9, DateOfConception)
dateadd (minute, 90, time 'now')
dateadd (? year to date '11-Sep-1973')
```

DATEDIFF()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the number of years, months, days, hours, minutes, seconds or milliseconds elapsed between two date/time values.

Result type: BIGINT

Syntax:

```
DATEDIFF (<args>)
```

```
<args>     ::=  <unit> FROM <moment1> TO <moment2>
                | <unit>, <moment1>, <moment2>

<unit>     ::=  YEAR | MONTH | DAY
                  | HOUR | MINUTE | SECOND | MILLISECOND
<momentN>  ::=  a DATE, TIME or TIMESTAMP expression
```

- DATE and TIMESTAMP arguments can be combined. No other mixes are allowed.
- With DATE arguments, only YEAR, MONTH and DAY can be used.
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Computation:

- DATEDIFF doesn't look at any smaller units than the one specified in the first argument. As a result,
  - "`datediff (year, date '1-Jan-2009', date '31-Dec-2009')`" returns 0, but
  - "`datediff (year, date '31-Dec-2009', date '1-Jan-2010')`" returns 1
- It does, however, look at all the *bigger* units. So:
  - "`datediff (day, date '26-Jun-1908', date '11-Sep-1973')`" returns 23818
- A negative result value indicates that *`moment2`* lies before *`moment1`*.

Examples:

```
datediff (hour from current_timestamp to timestamp '12-Jun-2059 06:00')
datediff (minute from time '0:00' to current_time)
datediff (month, current_date, date '1-1-1900')
datediff (day from current_date to cast(? as date))
```

DECODE()

Available in: DSQL, PSQL

Added in: 2.1

Description: DECODE is a shortcut for the so-called "simple CASE" construct, in which a given expression is compared to a number of other expressions until a match is found. The result is determined by the value listed after the matching expression. If no match is found, the default result is returned, if present. Otherwise, `NULL` is returned.

Result type: Varies

Syntax:

```
DECODE ( <test-expr>,
         <expr>, result
         [, <expr>, result ...]
         [, defaultresult] )
```

The equivalent CASE construct:

```
CASE <test-expr>
   WHEN <expr> THEN result
   [WHEN <expr> THEN result ...]
   [ELSE defaultresult]
END
```

> **Caution**
>
> Matching is done with the "=" operator, so if <test-expr> is NULL, it won't match any of the <expr>s, not even those that are NULL.

Example:

```
select name,
       age,
       decode( upper(sex),
               'M', 'Male',
               'F', 'Female',
               'Unknown' ),
       religion
from people
```

See also: CASE, Simple CASE

EXP()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the natural exponential, e^{number}

Result type: DOUBLE PRECISION

Syntax:

```
EXP (number)
```

See also: LN()

EXTRACT()

Available in: DSQL, ESQL, PSQL

Added in: IB 6

Changed in: 2.1

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

Result type: SMALLINT or NUMERIC

Syntax:

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

<part>      ::=  YEAR | MONTH | WEEK
                   | DAY | WEEKDAY | YEARDAY
                   | HOUR | MINUTE | SECOND | MILLISECOND
<datetime>  ::=  a DATE, TIME or TIMESTAMP expression
```

The returned datatypes and possible ranges are shown in the table below. If you try to extract a part that isn't present in the date/time argument (e.g. SECOND from a DATE or YEAR from a TIME), an error occurs.

Table 12.2. Types and ranges of EXTRACT results

Part	Type	Range	Comment
YEAR	SMALLINT	1–9999	
MONTH	SMALLINT	1–12	
WEEK	SMALLINT	1–53	
DAY	SMALLINT	1–31	
WEEKDAY	SMALLINT	0–6	0 = Sunday
YEARDAY	SMALLINT	0–365	0 = January 1
HOUR	SMALLINT	0–23	
MINUTE	SMALLINT	0–59	
SECOND	NUMERIC(9,4)	0.0000–59.9999	includes millisecond as fraction
MILLISECOND	NUMERIC(9,1)	0.0000–999.9	broken in 2.1, 2.1.1

MILLISECOND

Added in: 2.1 (with bug)

Fixed in: 2.1.2

Description: Firebird 2.1 and up support extraction of the millisecond from a TIME or TIMESTAMP. The datatype returned is NUMERIC(9,1).

Bug alert

MILLISECOND extraction is broken in Firebird 2.1 and 2.1.1. In those versions, the number returned is an INTEGER including SECOND*1000, so if the time is e.g. 20:48:17.637, the MILLISECOND value is 17637 while it should be 637. This bug has been fixed in version 2.1.2.

Note

If you extract the millisecond from CURRENT_TIME, be aware that this variable defaults to seconds precision, so the result will always be 0. Extract from CURRENT_TIME(3) or CURRENT_TIMESTAMP to get milliseconds precision.

WEEK

Added in: 2.1

Description: Firebird 2.1 and up support extraction of the ISO-8601 week number from a DATE or TIMESTAMP. ISO-8601 weeks start on a Monday and always have the full seven days. Week 1 is the first week that has a majority (at least 4) of its days in the new year. The first 1–3 days of the year may belong to the last week (52 or 53) of the previous year. Likewise, a year's final 1–3 days may belong to week 1 of the following year.

Caution

Be careful when combining WEEK and YEAR results. For instance, 30 December 2008 lies in week 1 of 2009, so "`extract (week from date '30 Dec 2008')`" returns 1. However, extracting YEAR always gives the calendar year, which is 2008. In this case, WEEK and YEAR are at odds with each other. The same happens when the first days of January belong to the last week of the previous year.

Please also notice that WEEKDAY is *not* ISO-8601 compliant: it returns 0 for Sunday, whereas ISO-8601 specifies 7.

FLOOR()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the largest whole number smaller than or equal to the argument.

Result type: BIGINT or DOUBLE PRECISION

Syntax:

```
FLOOR (number)
```

Important

If the external function FLOOR is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also: CEIL() / CEILING()

GEN_ID()

Available in: DSQL, ESQL, PSQL

Added in: IB

Description: Increments a generator or sequence and returns its new value. From Firebird 2.0 onward, the SQL-compliant NEXT VALUE FOR syntax is preferred, except when an increment other than 1 is needed.

Result type: BIGINT

Syntax:

```
GEN_ID (generator-name, <step>)

<step>  ::=  An integer expression.
```

Example:

```
new.rec_id = gen_id(gen_recnum, 1);
```

> **Warning**
>
> Unless you know very well what you are doing, using GEN_ID() with step values lower than 1 may compromise your data's integrity.

See also: NEXT VALUE FOR, CREATE GENERATOR

GEN_UUID()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns a universally unique ID as a 16-byte character string.

Result type: CHAR(16) CHARACTER SET OCTETS

Syntax:

```
GEN_UUID ()
```

HASH()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns a hash value for the input string. This function fully supports text BLOBs of any length and character set.

Result type: BIGINT

Syntax:

```
HASH (string)
```

IIF()

Available in: DSQL, PSQL

Added in: 2.0

Description: IIF takes three arguments. If the first evaluates to `true`, the second argument is returned; otherwise the third is returned.

Result type: Depends on input.

Syntax:

```
IIF (<condition>, ResultT, ResultF)

<condition>  ::=  A boolean expression.
```

Example:

```
select iif( sex = 'M', 'Sir', 'Madam' ) from Customers
```

IIF(*Cond*, *Result1*, *Result2*) is a shortcut for "CASE WHEN *Cond* THEN *Result1* ELSE *Result2* END". You can also compare IIF to the ternary "`? :`" operator in C-like languages.

LEFT()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the leftmost part of the argument string. The number of characters is given in the second argument.

Result type: VARCHAR or BLOB

Syntax:

```
LEFT (string, length)
```

- This function fully supports text BLOBs of any length, including those with a multi-byte character set.
- If *string* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the length of the input string.
- If the *length* argument exceeds the string length, the input string is returned unchanged.
- If the *length* argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

See also: RIGHT()

LN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the natural logarithm of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
LN (number)
```

- An error is raised if the argument is negative or 0.

> **Important**
>
> If the external function LN is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also: EXP()

LOG()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the *x*-based logarithm of *y*.

Result type: DOUBLE PRECISION

Syntax:

```
LOG (x, y)
```

- If *x* is negative or *y* is negative, the result is always `NaN`.
- If *x* is positive and *y* is 0, +/-`INF` is returned, depending on *x*.
- **Bug:** If *x* = 1 and *y* >= 0 (but not 1), +/-`INF` is returned.
- **Bug:** If *x* = 0 and *y* > 0, the result is 0.

> **Important**
>
> If the external function `LOG` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

LOG10()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the 10-based logarithm of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
LOG10 (number)
```

- If the argument is 0, `-INF` is returned. If the argument is negative, `NaN` is returned.

> **Important**
>
> If the external function `LOG10` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

LOWER()

Available in: DSQL, ESQL, PSQL

Added in: 2.0

Changed in: 2.1

Description: Returns the lower-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are lowercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

Result type: (VAR)CHAR or BLOB

Syntax:

```
LOWER (str)
```

> **Important**
>
> If the external function `LOWER` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Example:

```
select Sheriff from Towns
  where lower(Name) = 'cooper''s valley'
```

See also: UPPER

LPAD()

Available in: DSQL, PSQL

Added in: 2.1

Description: Left-pads a string with spaces or with a user-supplied string until a given length is reached.

Result type: VARCHAR(32765) or BLOB

Syntax:

```
LPAD (str, endlen [, padstr])
```

- This function fully supports text BLOBs of any length and character set.
- If *str* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(32765).
- If *padstr* is given and equals `''` (empty string), no padding takes place.
- If *endlen* is less than the current string length, the string is truncated to *endlen*, even if *padstr* is the empty string.

> **Important**
>
> If the external function `LPAD` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

> **Tip**
>
> With (VAR)CHARs, it is generally wise to CAST the result to a smaller size. The default result length of 32765 may, in combination with other output columns, lead to a “block size exceeds implementation restriction” error.

Examples:

```
lpad ('Hello', 12)                -- returns '       Hello'
lpad ('Hello', 12, '-')           -- returns '-------Hello'
lpad ('Hello', 12, '')            -- returns 'Hello'
lpad ('Hello', 12, 'abc')         -- returns 'abcabcaHello'
lpad ('Hello', 12, 'abcdefghij')  -- returns 'abcdefgHello'
lpad ('Hello', 2)                 -- returns 'He'
lpad ('Hello', 2, '-')            -- returns 'He'
lpad ('Hello', 2, '')             -- returns 'He'
```

> **Warning**
>
> When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

See also: RPAD()

MAXVALUE()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the maximum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

Result type: Varies

Syntax:

```
MAXVALUE (expr [, expr ...])
```

- If one or more expressions resolve to `NULL`, MAXVALUE returns `NULL`. This behaviour differs from the aggregate function MAX.

See also: MINVALUE()

MINVALUE()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the minimum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

Result type: Varies

Syntax:

```
MINVALUE (expr [, expr ...])
```

- If one or more expressions resolve to `NULL`, MINVALUE returns `NULL`. This behaviour differs from the aggregate function MIN.

See also: MAXVALUE()

MOD()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the remainder of an integer division.

Result type: INTEGER or BIGINT

Syntax:

```
MOD (a, b)
```

- Non-integer arguments are rounded before the division takes place. So, “7.5 mod 2.5” gives 2 (8 mod 3), not 0.

> **Important**
>
> If the external function `MOD` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

NULLIF()

Available in: DSQL, PSQL

Added in: 1.5

Description: NULLIF returns the value of the first argument, unless it is equal to the second. In that case, `NULL` is returned.

Result type: Depends on input.

Syntax:

```
NULLIF (<exp1>, <exp2>)
```

Example:

```
select avg( nullif(Weight, -1) ) from FatPeople
```

This will return the average weight of the persons listed in FatPeople, excluding those having a weight of -1, since AVG skips NULL data. Presumably, -1 indicates "weight unknown" in this table. A plain AVG(Weight) would include the -1 weights, thus skewing the result.

> **Note**
>
> In Firebird 1.0.x, where NULLIF is not available, you can accomplish the same with the `*nullif` external functions.

OCTET_LENGTH()

Available in: DSQL, PSQL

Added in: 2.0

Changed in: 2.1

Description: Gives the length in bytes (octets) of the input string. For multi-byte character sets, this may be less than the number of characters times the "formal" number of bytes per character as found in RDB $CHARACTER_SETS.

> **Note**
>
> With arguments of type CHAR, this function takes the entire formal string length (e.g. the declared length of a field or variable) into account. If you want to obtain the "logical" byte length, not counting the trailing spaces, right-TRIM the argument before passing it to OCTET_LENGTH.

Result type: INTEGER

Syntax:

```
OCTET_LENGTH (str)
```

BLOB support: Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

Examples:

```
select octet_length('Hello!') from rdb$database
  -- returns 6

select octet_length(_iso8859_1 'Grüß di!') from rdb$database
  -- returns 8: ü and ß take up one byte each in ISO8859_1

select octet_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
  -- returns 10: ü and ß take up two bytes each in UTF8

select octet_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
  -- returns 26: all 24 CHAR positions count, and two of them are 2-byte
```

See also: BIT_LENGTH(), CHARACTER_LENGTH

OVERLAY()

Available in: DSQL, PSQL

Added in: 2.1

Description: Replaces part of a string with another string. By default, the number of characters removed from the host string equals the length of the replacement string. With the optional fourth argument, the user can specify a different number of characters to be removed.

Result type: VARCHAR or BLOB

Syntax:

```
OVERLAY (string PLACING replacement FROM pos [FOR length])
```

- This function supports BLOBs of any length. Due to a bug, BLOBs containing multi-byte characters – and sometimes even single-byte non-ASCII characters – will cause a "Cannot transliterate character between character sets" error. This bug does not occur in Firebird 2.5.
- If *string* or *replacement* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the sum of the lengths of *string* and *replacement*.
- As usual in SQL string functions, *pos* is 1-based.
- If *pos* is beyond the end of *string*, *replacement* is placed directly after *string*.
- If the number of characters from *pos* to the end of *string* is smaller than the length of *replacement* (or than the *length* argument, if present), *string* is truncated at *pos* and *replacement* placed after it.
- The effect of a "FOR 0" clause is that *replacement* is simply inserted into *string*.
- If any argument is NULL, the result is NULL.
- If *pos* or *length* is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

Examples:

```
overlay ('Goodbye' placing 'Hello' from 2)    -- returns 'GHelloe'
overlay ('Goodbye' placing 'Hello' from 5)    -- returns 'GoodHello'
overlay ('Goodbye' placing 'Hello' from 8)    -- returns 'GoodbyeHello'
overlay ('Goodbye' placing 'Hello' from 20)   -- returns 'GoodbyeHello'

overlay ('Goodbye' placing 'Hello' from 2 for 0)   -- r. 'GHellooodbye'
overlay ('Goodbye' placing 'Hello' from 2 for 3)   -- r. 'GHellobye'
overlay ('Goodbye' placing 'Hello' from 2 for 6)   -- r. 'GHello'
overlay ('Goodbye' placing 'Hello' from 2 for 9)   -- r. 'GHello'

overlay ('Goodbye' placing '' from 4)         -- returns 'Goodbye'
```

```
overlay ('Goodbye' placing '' from 4 for 3)   -- returns 'Gooe'
overlay ('Goodbye' placing '' from 4 for 20)  -- returns 'Goo'

overlay ('' placing 'Hello' from 4)           -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 0)     -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 20)    -- returns 'Hello'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

See also: REPLACE()

PI()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns an approximation of the value of #.

Result type: DOUBLE PRECISION

Syntax:

```
PI ()
```

Important

If the external function `PI` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

POSITION()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the (1-based) position of the first occurrence of a substring in a host string. With the optional third argument, the search starts at a given offset, disregarding any matches that may occur earlier in the string. If no match is found, the result is 0.

Result type: INTEGER

Syntax:

```
POSITION (<args>)
```

```
<args>  ::=  substr IN string
             | substr, string [, startpos]
```

- The optional third argument is only supported in the second syntax (comma syntax).
- The empty string is considered a substring of every string. Therefore, if *substr* is '' (empty string) and *string* is not NULL, the result is:
  - 1 if *startpos* is not given;
  - *startpos* if *startpos* lies within *string*;
  - 0 if *startpos* lies beyond the end of *string*.

  Notice: A bug in Firebird 2.1–2.1.3 causes POSITION to *always* return 1 if *substr* is the empty string. This is fixed in 2.1.4.
- This function fully supports text BLOBs of any size and character set.

Examples:

```
position ('be' in 'To be or not to be')      -- returns 4
position ('be', 'To be or not to be')        -- returns 4
position ('be', 'To be or not to be', 4)     -- returns 4
position ('be', 'To be or not to be', 8)     -- returns 17
position ('be', 'To be or not to be', 18)    -- returns 0
position ('be' in 'Alas, poor Yorick!')      -- returns 0
```

> **Warning**
>
> When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

POWER()

Available in: DSQL, PSQL

Added in: 2.1

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

Result type: DOUBLE PRECISION

Syntax:

```
POWER (x, y)
```

- If *x* negative, an error is raised.

> **Important**
>
> If the external function POWER is declared in your database as power instead of the default dPower, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

RAND()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns a random number between 0 and 1.

Result type: DOUBLE PRECISION

Syntax:

```
RAND ()
```

> **Important**
>
> If the external function `RAND` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

RDB$GET_CONTEXT()

> **Note**
>
> RDB$GET_CONTEXT and its counterpart RDB$SET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present – the user doesn't have to do anything to make them available.

Available in: DSQL, ESQL, PSQL

Added in: 2.0

Description: Retrieves the value of a context variable from one of the namespaces SYSTEM, USER_SESSION and USER_TRANSACTION.

Result type: VARCHAR(255)

Syntax:

```
RDB$GET_CONTEXT ('<namespace>', '<varname>')

<namespace>  ::=  SYSTEM | USER_SESSION | USER_TRANSACTION
<varname>    ::=  A case-sensitive string of max. 80 characters
```

The namespaces:
The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The SYSTEM namespace is read-only. It contains a number of predefined variables, shown in the table below.

Table 12.3. Context variables in the SYSTEM namespace

DB_NAME	Either the full path to the database or – if connecting via the path is disallowed – its alias.
NETWORK_PROTOCOL	The protocol used for the connection: 'TCPv4', 'WNET', 'XNET' or NULL.
CLIENT_ADDRESS	For TCPv4, this is the IP address. For XNET, the local process ID. For all other protocols this variable is NULL.
CURRENT_USER	Same as global CURRENT_USER variable.
CURRENT_ROLE	Same as global CURRENT_ROLE variable.
SESSION_ID	Same as global CURRENT_CONNECTION variable.
TRANSACTION_ID	Same as global CURRENT_TRANSACTION variable.
ISOLATION_LEVEL	The isolation level of the current transaction: 'READ COMMITTED', 'SNAPSHOT' or 'CONSISTENCY'.

Return values and error behaviour: If the polled variable exists in the given namespace, its value will be returned as a string of max. 255 characters. If the namespace doesn't exist or if you try to access a non-existing variable in the SYSTEM namespace, an error is raised. If you poll a non-existing variable in one of the other namespaces, NULL is returned. Both namespace and variable names must be given as single-quoted, case-sensitive, non-NULL strings.

Examples:

```
select rdb$get_context('SYSTEM', 'DB_NAME') from rdb$database
```

```
New.UserAddr = rdb$get_context('SYSTEM', 'CLIENT_ADDRESS');
```

```
insert into MyTable (TestField)
  values (rdb$get_context('USER_SESSION', 'MyVar'))
```

See also: RDB$SET_CONTEXT()

RDB$SET_CONTEXT()

> **Note**
>
> RDB$SET_CONTEXT and its counterpart RDB$GET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present – the user doesn't have to do anything to make them available.

Available in: DSQL, ESQL, PSQL

Added in: 2.0

Description: Creates, sets or unsets a variable in one of the user-writable namespaces USER_SESSION and USER_TRANSACTION.

Result type: INTEGER

Syntax:

```
RDB$SET_CONTEXT ('<namespace>', '<varname>', <value> | NULL)

<namespace>  ::=  USER_SESSION | USER_TRANSACTION
<varname>    ::=  A case-sensitive string of max. 80 characters
<value>      ::=  A value of any type, as long as it's castable
                   to a VARCHAR(255)
```

The namespaces:
The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The USER_SESSION context is bound to the current connection. Variables in USER_TRANSACTION only exist in the transaction in which they have been set. When the transaction ends, the context and all the variables defined in it are destroyed.

Return values and error behaviour:
The function returns 1 if the variable already existed before the call and 0 if it didn't. To remove a variable from a context, set it to `NULL`. If the given namespace doesn't exist, an error is raised. Both namespace and variable names must be entered as single-quoted, case-sensitive, non-`NULL` strings.

Examples:

```
select rdb$set_context('USER_SESSION', 'MyVar', 493) from rdb$database
```

```
rdb$set_context('USER_SESSION', 'RecordsFound', RecCounter);
```

```
select rdb$set_context('USER_TRANSACTION', 'Savepoints', 'Yes')
  from rdb$database
```

Notes:

- The maximum number of variables in any single context is 1000.
- All USER_TRANSACTION variables will survive a ROLLBACK RETAIN or ROLLBACK TO SAVEPOINT unaltered, no matter at which point during the transaction they were set.
- Due to its UDF-like nature, RDB$SET_CONTEXT can – in PSQL only – be called like a void function, without assigning the result, as in the second example above. Regular internal functions don't allow this type of use.

See also: RDB$GET_CONTEXT()

REPLACE()

Available in: DSQL, PSQL

Added in: 2.1

Description: Replaces all occurrences of a substring in a string.

Result type: VARCHAR or BLOB

Syntax:

```
REPLACE (str, find, repl)
```

- This function fully supports text BLOBs of any length and character set.
- If any argument is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* calculated from the lengths of *str*, *find* and *repl* in such a way that even the maximum possible number of replacements won't overflow the field.
- If *find* is the empty string, *str* is returned unchanged.
- If *repl* is the empty string, all occurrences of *find* are deleted from *str*.
- If any argument is NULL, the result is always NULL, even if nothing would have been replaced.

Examples:

```
replace ('Billy Wilder',  'il', 'oog')     -- returns 'Boogly Woogder'
replace ('Billy Wilder',  'il',    '')     -- returns 'Bly Wder'
replace ('Billy Wilder',  null, 'oog')     -- returns NULL
replace ('Billy Wilder',  'il',  null)     -- returns NULL
replace ('Billy Wilder', 'xyz',  null)     -- returns NULL (!)
replace ('Billy Wilder', 'xyz', 'abc')     -- returns 'Billy Wilder'
replace ('Billy Wilder',    '', 'abc')     -- returns 'Billy Wilder'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

See also: OVERLAY()

REVERSE()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns a string backwards.

Result type: VARCHAR

Syntax:

```
REVERSE (str)
```

Examples:

```
reverse ('spoonful')                 -- returns 'lufnoops'
```

```
reverse ('Was it a cat I saw?')      -- returns '?was I tac a ti saW'
```

Tip

This function comes in very handy if you want to group, search or order on string endings, e.g. when dealing with domain names or email addresses:

```
create index ix_people_email on people
  computed by (reverse(email));

select * from people
  where reverse(email) starting with reverse('.br');
```

RIGHT()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the rightmost part of the argument string. The number of characters is given in the second argument.

Result type: VARCHAR or BLOB

Syntax:

```
RIGHT (string, length)
```

- This function supports text BLOBs of any length, but has a bug in versions 2.1–2.1.3 that makes it fail with text BLOBs larger than 1024 bytes that have a multi-byte character set. This has been fixed in version 2.1.4.
- If *string* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(*n*) with *n* the length of the input string.
- If the *length* argument exceeds the string length, the input string is returned unchanged.
- If the *length* argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

Warning

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

Important

If the external function `RIGHT` is declared in your database as `right` instead of the default `sright`, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

See also: LEFT()

ROUND()

Available in: DSQL, PSQL

Added in: 2.1

Description: Rounds a number to the nearest integer. If the fractional part is exactly `0.5`, rounding is upward for positive numbers and downward for negative numbers. With the optional *`scale`* argument, the number can be rounded to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

Result type: INTEGER, (scaled) BIGINT or DOUBLE

Syntax:

```
ROUND (<number> [, <scale>])

<number>  ::=  a numerical expression
<scale>   ::=  an integer specifying the number of decimal places
               toward which should be rounded, e.g.:
                  2 for rounding to the nearest multiple of 0.01
                  1 for rounding to the nearest multiple of 0.1
                  0 for rounding to the nearest whole number
                 -1 for rounding to the nearest multiple of 10
                 -2 for rounding to the nearest multiple of 100
```

Notes:

- If the *`scale`* argument is present, the result usually has the same scale as the first argument, e.g.

  - ROUND(123.654, 1) returns 123.700 (not 123.7)
  - ROUND(8341.7, -3) returns 8000.0 (not 8000)
  - ROUND(45.1212, 0) returns 45.0000 (not 45)

  Otherwise, the result scale is 0:

  - ROUND(45.1212) returns 45

> **Important**
>
> - If the external function `ROUND` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).
> - If you are used to the behaviour of the external function `ROUND`, please notice that the *internal* function always rounds halves away from zero, i.e. downward for negative numbers.

RPAD()

Available in: DSQL, PSQL

Added in: 2.1

Description: Right-pads a string with spaces or with a user-supplied string until a given length is reached.

Result type: VARCHAR(32765) or BLOB

Syntax:

```
RPAD (str, endlen [, padstr])
```

- This function fully supports text BLOBs of any length and character set.
- If *str* is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(32765).
- If *padstr* is given and equals '' (empty string), no padding takes place.
- If *endlen* is less than the current string length, the string is truncated to *endlen*, even if *padstr* is the empty string.

Important

If the external function RPAD is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

Tip

With (VAR)CHARs, it is generally wise to CAST the result to a smaller size. The default result length of 32765 may, in combination with other output columns, lead to a "block size exceeds implementation restriction" error.

Examples:

```
rpad ('Hello', 12)                -- returns 'Hello       '
rpad ('Hello', 12, '-')           -- returns 'Hello-------'
rpad ('Hello', 12, '')            -- returns 'Hello'
rpad ('Hello', 12, 'abc')         -- returns 'Helloabcabca'
rpad ('Hello', 12, 'abcdefghij')  -- returns 'Helloabcdefg'
rpad ('Hello', 2)                 -- returns 'He'
rpad ('Hello', 2, '-')            -- returns 'He'
rpad ('Hello', 2, '')             -- returns 'He'
```

Warning

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

See also: LPAD()

SIGN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the sign of the argument: -1, 0 or 1.

Result type: SMALLINT

Syntax:

```
SIGN (number)
```

Important

If the external function `SIGN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SIN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns an angle's sine. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
SIN (angle)
```

- Any non-`NULL` result is – obviously – in the range [-1, 1].

Important

If the external function `SIN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SINH()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the hyperbolic sine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
SINH (number)
```

> **Important**
>
> If the external function `SINH` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SQRT()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the square root of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
SQRT (number)
```

> **Important**
>
> If the external function `SQRT` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

SUBSTRING()

Available in: DSQL, PSQL

Added in: 1.0

Changed in: 2.0, 2.1

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

Result type: VARCHAR(*n*) or BLOB

Syntax:

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

This function returns the substring starting at character position *`startpos`* (the first position being 1). Without the FOR argument, it returns all the remaining characters in the string. With FOR, it returns *`length`* characters or the remainder of the string, whichever is shorter.

In Firebird 1.x, *startpos* and *length* must be integer literals. In 2.0 and above they can be any valid integer expression.

Starting with Firebird 2.1, this function fully supports binary and text BLOBs of any length and character set. If *str* is a BLOB, the result is also a BLOB. For any other argument type, the result is a VARCHAR(*n*). Previously, the result type used to be CHAR(*n*) if the argument was a CHAR(*n*) or a string literal.

For non-BLOB arguments, the width of the result field is always equal to the length of *str*, regardless of *startpos* and *length*. So, `substring('pinhead' from 4 for 2)` will return a VARCHAR(7) containing the string `'he'`.

If any argument is `NULL`, the result is `NULL`.

Bugs

- If *str* is a BLOB and the *length* argument is not present, the output is limited to 32767 characters. Workaround: with long BLOBs, always specify char_length(*str*) – or a sufficiently high integer – as the third argument, unless you are sure that the requested substring fits within 32767 characters.
- A bug in Firebird 2.0 which caused the function to return “false emptystrings” if *startpos* or *length* was `NULL`, has been fixed.

Example:

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

Warning

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

TAN()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns an angle's tangent. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
TAN (angle)
```

Important

If the external function `TAN` is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

TANH()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the hyperbolic tangent of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
TANH (number)
```

- Due to rounding, any non-NULL result is in the range [-1, 1] (mathematically, it's <-1, 1>).

> **Important**
>
> If the external function TANH is declared in your database, it will override the internal function. To make the internal function available, DROP or ALTER the external function (UDF).

TRIM()

Available in: DSQL, PSQL

Added in: 2.0

Changed in: 2.1

Description: Removes leading and/or trailing spaces (or optionally other strings) from the input string. Since Firebird 2.1 this function fully supports text BLOBs of any length and character set.

Result type: VARCHAR(*n*) or BLOB

Syntax:

```
TRIM ([<adjust>] str)

<adjust>  ::=  {[where] [what]} FROM

where     ::=  BOTH | LEADING | TRAILING       /* default is BOTH */

what      ::=  The substring to be removed (repeatedly if necessary)
               from str's head and/or tail. Default is ' ' (space).
```

Examples:

```
select trim ('  Waste no space   ') from rdb$database
  -- returns 'Waste no space'
```

```
select trim (leading from '  Waste no space   ') from rdb$database
  -- returns 'Waste no space   '

select trim (leading '.' from '  Waste no space   ') from rdb$database
  -- returns '  Waste no space   '

select trim (trailing '!' from 'Help!!!!') from rdb$database
  -- returns 'Help'

select trim ('la' from 'lalala I love you Ella') from rdb$database
  -- returns ' I love you El'

select trim ('la' from 'Lalala I love you Ella') from rdb$database
  -- returns 'Lalala I love you El'
```

Notes:

- If `str` is a BLOB, the result is a BLOB. Otherwise, it is a VARCHAR(*n*) with *n* the formal length of `str`.
- The substring to be removed, if specified, may not be bigger than 32767 bytes. However, if this substring is *repeated* at `str`'s head or tail, the total number of bytes removed may be far greater. (The restriction on the size of the substring will be lifted in Firebird 3.)

> **Warning**
>
> When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

TRUNC()

Available in: DSQL, PSQL

Added in: 2.1

Description: Returns the integer part of a number. With the optional `scale` argument, the number can be truncated to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

Result type: INTEGER, (scaled) BIGINT or DOUBLE

Syntax:

```
TRUNC (<number> [, <scale>])

<number>  ::=  a numerical expression
<scale>   ::=  an integer specifying the number of decimal places
               toward which should be truncated, e.g.:
                  2 for truncating to a multiple of 0.01
                  1 for truncating to a multiple of 0.1
                  0 for truncating to a whole number
                 -1 for truncating to a multiple of 10
                 -2 for truncating to a multiple of 100
```

Notes:

- If the *scale* argument is present, the result usually has the same scale as the first argument, e.g.

  - TRUNC(789.2225, 2) returns 789.2200 (not 789.22)
  - TRUNC(345.4, -2) returns 300.0 (not 300)
  - TRUNC(-163.41, 0) returns -163.00 (not -163)

  Otherwise, the result scale is 0:

  - TRUNC(-163.41) returns -163

> **Important**
>
> If you are used to the behaviour of the external function TRUNCATE, please notice that the *internal* function TRUNC always truncates toward zero, i.e. upward for negative numbers.

UPPER()

Available in: DSQL, ESQL, PSQL

Added in: IB

Changed in: 2.0, 2.1

Description: Returns the upper-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are uppercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

Result type: (VAR)CHAR or BLOB

Syntax:

```
UPPER (str)
```

Examples:

```
select upper(_iso8859_1 'Débâcle')
from rdb$database
  -- returns 'DÉBÂCLE' (before Firebird 2.0: 'DéBâCLE')

select upper(_iso8859_1 'Débâcle' collate fr_fr)
from rdb$database
  -- returns 'DEBACLE', following French uppercasing rules
```

See also: LOWER

Chapter 13

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.

abs

Library: ib_udf

Added in: IB

Better alternative: Internal function ABS()

Description: Returns the absolute value of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
abs (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION abs
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_abs' MODULE_NAME 'ib_udf'
```

acos

Library: ib_udf

Added in: IB

Better alternative: Internal function ACOS()

Description: Returns the arc cosine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
acos (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION acos
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_acos' MODULE_NAME 'ib_udf'
```

addDay

Library: fbudf

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

Better alternative: Internal function DATEADD

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

Result type: TIMESTAMP

Syntax:

```
addday (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)

Better alternative: Internal function DATEADD

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

Result 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)

Better alternative: Internal function DATEADD

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

Result 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)

Better alternative: Internal function DATEADD

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

Result 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)

Better alternative: Internal function DATEADD

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

Result 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)

Better alternative: Internal function DATEADD

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

Result 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.

Result type: TIMESTAMP

Syntax:

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

Declaration:

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

The DATEADD alternative: The internal function DATEADD, which can replace all the other `add<DateTimePart>` functions, doesn't support WEEK yet. This will be realised in Firebird 2.5. Meanwhile, you can use DATEADD(7*number DAY TO *atimestamp*) – or stick with `addWeek`.

addYear

Library: fbudf

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

Better alternative: Internal function DATEADD

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

Result 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, 2.0

Better alternative: Internal function ASCII_CHAR()

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

Result type: VARCHAR(1)

Syntax (unchanged):

```
ascii_char (intval)
```

Declaration:

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

The declaration reflects 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 the result to the caller as a VARCHAR(1) though.

The **NULL** after INTEGER is an optional addition that became available in Firebird 2. When declared with the NULL keyword, the engine will pass a `NULL` argument unchanged to the function. This causes a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL` is passed to the function as 0 and the result is an empty string.

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- `ascii_char(0)` returns an empty string in all versions, not a character with ASCII value 0.
- Before Firebird 2.0, the result type was CHAR(1).

ascii_val

Library: ib_udf

Added in: IB

Better alternative: Internal function ASCII_VAL()

Description: Returns the ASCII code of the character passed in.

Result type: INTEGER

Syntax:

```
ascii_val (ch)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION ascii_val
   CHAR(1)
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_ascii_val' MODULE_NAME 'ib_udf'
```

Caution

Because CHAR fields are padded with spaces, an empty string argument will be seen as a space, and yield a result of 32. The internal function ASCII_VAL returns 0 in this case.

asin

Library: ib_udf

Added in: IB

Better alternative: Internal function ASIN()

Description: Returns the arc sine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
asin (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION asin
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_asin' MODULE_NAME 'ib_udf'
```

atan

Library: ib_udf

Added in: IB

Better alternative: Internal function ATAN()

Description: Returns the arc tangent of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
atan (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION atan
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_atan' MODULE_NAME 'ib_udf'
```

atan2

Library: ib_udf

Added in: IB

Better alternative: Internal function ATAN2()

Description: Returns the angle whose sine-to-cosine *ratio* is given by the two arguments, and whose sine and cosine *signs* correspond to the signs of the arguments. This allows results across the entire circle, including the angles -#/2 and #/2.

Result type: DOUBLE PRECISION

Syntax:

```
atan2 (num1, num2)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION atan2
   DOUBLE PRECISION, DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_atan2' MODULE_NAME 'ib_udf'
```

bin_and

Library: ib_udf

Added in: IB

Better alternative: Internal function BIN_AND()

Description: Returns the bitwise AND result of the arguments.

Result type: INTEGER

Syntax:

```
bin_and (num1, num2)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION bin_and
   INTEGER, INTEGER
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_bin_and' MODULE_NAME 'ib_udf'
```

bin_or

Library: ib_udf

Added in: IB

Better alternative: Internal function BIN_OR()

Description: Returns the bitwise OR result of the arguments.

Result type: INTEGER

Syntax:

```
bin_or (num1, num2)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION bin_or
   INTEGER, INTEGER
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_bin_or' MODULE_NAME 'ib_udf'
```

bin_xor

Library: ib_udf

Added in: IB

Better alternative: Internal function BIN_XOR()

Description: Returns the bitwise XOR result of the arguments.

Result type: INTEGER

Syntax:

```
bin_xor (num1, num2)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION bin_xor
   INTEGER, INTEGER
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_bin_xor' MODULE_NAME 'ib_udf'
```

ceiling

Library: ib_udf

Added in: IB

Better alternative: Internal function CEIL() / CEILING()

Description: Returns the smallest whole number that is greater than or equal to the argument.

Result type: DOUBLE PRECISION

Syntax:

```
ceiling (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION ceiling
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_ceiling' MODULE_NAME 'ib_udf'
```

cos

Library: ib_udf

Added in: IB

Better alternative: Internal function COS()

Description: Returns an angle's cosine. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
cos (angle)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION cos
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_cos' MODULE_NAME 'ib_udf'
```

cosh

Library: ib_udf

Added in: IB

Better alternative: Internal function COSH()

Description: Returns the hyperbolic cosine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
cosh (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION cosh
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_cosh' MODULE_NAME 'ib_udf'
```

cot

Library: ib_udf

Added in: IB

Better alternative: Internal function COT()

Description: Returns an angle's cotangent. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
cot (angle)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION cot
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_cot' MODULE_NAME 'ib_udf'
```

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.

Result 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)

Better alternative: Internal function POWER()

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

Result 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'
```

floor

Library: ib_udf

Added in: IB

Better alternative: Internal function FLOOR()

Description: Returns the largest whole number that is smaller than or equal to the argument.

Result type: DOUBLE PRECISION

Syntax:

```
floor (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION floor
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_floor' MODULE_NAME 'ib_udf'
```

getExactTimestamp

Library: fbudf

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

Better alternative: `CURRENT_TIMESTAMP` or `'NOW'`

Description: Returns the system time with milliseconds precision. This function was added because in pre-2.0 versions, `CURRENT_TIMESTAMP` always had `.0000` in the fractional part of the second. In Firebird 2.0 and up it is better to use `CURRENT_TIMESTAMP`, which now also defaults to milliseconds precision. To measure time intervals in PSQL modules, use `'NOW'`.

Result 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.

ln

Library: ib_udf

Added in: IB

Better alternative: Internal function LN()

Description: Returns the natural logarithm of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
ln (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION ln
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_ln' MODULE_NAME 'ib_udf'
```

log

Library: ib_udf

Added in: IB

Changed in: 1.5

Better alternative: Internal function LOG()

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

Result 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 use `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 wrong results.

log10

Library: ib_udf

Added in: IB

Better alternative: Internal function LOG10()

Description: Returns the 10-based logarithm of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
log10 (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION log10
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_log10' MODULE_NAME 'ib_udf'
```

lower

Library: ib_udf

Added in: IB

Changed in: 2.0

Better alternative: Internal function LOWER()

Description: Returns the lower-case version of the input string. Please notice that only ASCII characters are handled correctly. If possible, use the superior internal function LOWER instead. Just dropping the declaration of the `lower` UDF should do the trick, unless you gave it an alternative name.

Result type: VARCHAR(*n*)

Syntax:

```
"LOWER" (str)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION "LOWER"
   CSTRING(255) NULL
   RETURNS CSTRING(255) FREE_IT
   ENTRY_POINT 'IB_UDF_lower' MODULE_NAME 'ib_udf'
```

The above declaration is from the file `ib_udf2.sql`. "LOWER" has been surrounded by double-quotes to avoid confusion with the internal function LOWER.

The **NULL** after CSTRING(255) is an optional addition that became available in Firebird 2. When declared with the NULL keyword, the engine will pass a `NULL` argument unchanged to the function. This leads to a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL` is passed to the function as an empty string and the result is an empty string as well.

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- Before Firebird 2.0, the result type was CHAR(*n*).
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).

lpad

Library: ib_udf

Added in: 1.5

Changed in: 1.5.2, 2.0

Better alternative: Internal function LPAD()

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

Result type: VARCHAR(*n*)

Syntax:

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

Declaration:

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

The above declaration is from the file `ib_udf2.sql`. The **NULL**s after the CSTRING arguments are an optional addition that became available in Firebird 2. If an argument is declared with the NULL keyword, the engine will pass a `NULL` argument value unchanged to the function. This leads to a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL`s are passed to the function as empty strings and the result is a string with *endlengh* padchars (if *str* is `NULL`) or a copy of *str* itself (if *padchar* is `NULL`).

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- 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.
- If *endlength* is less than *str*'s length, *str* is truncated to *endlength*. If *endlength* is negative, the result is `NULL`.
- A `NULL` *endlength* is treated as if it were 0.
- If *padchar* is empty, or if *padchar* is `NULL` and the function has been declared without the NULL keyword after the last argument, *str* is returned unchanged (or truncated to *endlength*).
- Before Firebird 2.0, the result type was CHAR(*n*).
- A bug that caused an endless loop if *padchar* was empty or `NULL` has been fixed in 2.0.
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).

ltrim

Library: ib_udf

Changed in: 1.5, 1.5.2, 2.0

Better alternative: Internal function TRIM()

Description: Returns the input string with any leading space characters removed. In new code, you are advised to use the internal function TRIM instead, as it is both more powerful and more versatile.

Result type: VARCHAR(*n*)

Syntax (unchanged):

```
ltrim (str)
```

Declaration:

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

The above declaration is from the file `ib_udf2.sql`. The **NULL** after the argument is an optional addition that became available in Firebird 2. If the argument is declared with the NULL keyword, the engine will pass a `NULL` argument value unchanged to the function. This leads to a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL` is passed to the function as an empty string and the result is an empty string as well.

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- Before Firebird 2.0, the result type was CHAR(*n*).
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).
- In Firebird 1.0.x, this function returned `NULL` if the input string was either empty or `NULL`.

mod

Library: ib_udf

Added in: IB

Better alternative: Internal function MOD()

Description: Returns the remainder of an integer division.

Result type: DOUBLE PRECISION

Syntax:

```
mod (a, b)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION mod
   INTEGER, INTEGER
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_mod' MODULE_NAME 'ib_udf'
```

*nullif

Library: fbudf

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

Better alternative: Internal function NULLIF()

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`.

Result type: Varies, see declarations.

Syntax:

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

As from Firebird 1.5, use of the internal function NULLIF is preferred.

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)

Better alternative: Internal function COALESCE()

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.

Result type: Varies, see declarations.

Syntax:

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

As from Firebird 1.5, use of the internal function COALESCE is preferred.

> **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'
```

pi

Library: ib_udf

Added in: IB

Better alternative: Internal function PI()

Description: Returns an approximation of the value of #.

Result type: DOUBLE PRECISION

Syntax:

```
pi ()
```

Declaration:

```
DECLARE EXTERNAL FUNCTION pi
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_pi' MODULE_NAME 'ib_udf'
```

rand

Library: ib_udf

Changed in: 2.0

Better alternative: Internal function RAND()

Description: Returns a pseudo-random number. Before Firebird 2.0, this function would first seed the random number generator with the current time in seconds. Multiple `rand()` calls within the same second would therefore return the same value. If you want that old behaviour in Firebird 2 and up, use `srand()`.

Result type: DOUBLE PRECISION

Syntax:

```
rand ()
```

Declaration:

```
DECLARE EXTERNAL FUNCTION rand
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_rand' MODULE_NAME 'ib_udf'
```

right

See sright.

round, i64round

Library: fbudf

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

Changed in: 1.5, 2.1.3

Better alternative: Internal function ROUND()

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

Result type: INTEGER / NUMERIC(18,4)

Syntax:

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

Caution

Halves are always rounded upward, i.e. away from zero for positive numbers and toward zero for negative numbers. For instance, 3.5 is rounded to 4, but -3.5 is rounded to -3. The internal function ROUND, available since Firebird 2.1, rounds all halves away from zero.

Bug alert

In versions 2.1, 2.1.1 and 2.1.2, 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.

Fixed in 2.1.3.

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. From Firebird 2.0 onward you can also perform this update with ALTER EXTERNAL FUNCTION.

rpad

Library: ib_udf

Added in: 1.5

Changed in: 1.5.2, 2.0

Better alternative: Internal function RPAD()

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

Result type: VARCHAR(*n*)

Syntax:

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

Declaration:

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

The above declaration is from the file `ib_udf2.sql`. The **NULL**s after the CSTRING arguments are an optional addition that became available in Firebird 2. If an argument is declared with the NULL keyword, the engine will pass a `NULL` argument value unchanged to the function. This leads to a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL`s are passed to the function as empty strings and the result is a string with *endlengh* padchars (if *str* is `NULL`) or a copy of *str* itself (if *padchar* is `NULL`).

For more information about passing NULLs to UDFs, see the note at the end of this book.

Notes:

- 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.
- If *endlength* is less than *str*'s length, *str* is truncated to *endlength*. If *endlength* is negative, the result is NULL.
- A NULL *endlength* is treated as if it were 0.
- If *padchar* is empty, or if *padchar* is NULL and the function has been declared without the NULL keyword after the last argument, *str* is returned unchanged (or truncated to *endlength*).
- Before Firebird 2.0, the result type was CHAR(*n*).
- A bug that caused an endless loop if *padchar* was empty or NULL has been fixed in 2.0.
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).

rtrim

Library: ib_udf

Changed in: 1.5, 1.5.2, 2.0

Better alternative: Internal function TRIM()

Description: Returns the input string with any trailing space characters removed. In new code, you are advised to use the internal function TRIM instead, as it is both more powerful and more versatile.

Result type: VARCHAR(*n*)

Syntax (unchanged):

```
rtrim (str)
```

Declaration:

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

The above declaration is from the file ib_udf2.sql. The **NULL** after the argument is an optional addition that became available in Firebird 2. If the argument is declared with the NULL keyword, the engine will pass a NULL argument value unchanged to the function. This leads to a NULL result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), NULL is passed to the function as an empty string and the result is an empty string as well.

For more information about passing NULLs to UDFs, see the note at the end of this book.

Notes:

- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- Before Firebird 2.0, the result type was CHAR(*n*).
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).
- In Firebird 1.0.x, this function returned NULL if the input string was either empty or NULL.

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.

Result 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

sign

Library: ib_udf

Added in: IB

Better alternative: Internal function SIGN()

Description: Returns the sign of the argument: -1, 0 or 1.

Result type: INTEGER

Syntax:

```
sign (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION sign
   DOUBLE PRECISION
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_sign' MODULE_NAME 'ib_udf'
```

sin

Library: ib_udf

Added in: IB

Better alternative: Internal function SIN()

Description: Returns an angle's sine. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
sin (angle)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION sin
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_sin' MODULE_NAME 'ib_udf'
```

sinh

Library: ib_udf

Added in: IB

Better alternative: Internal function SINH()

Description: Returns the hyperbolic sine of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
sinh (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION sinh
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_sinh' MODULE_NAME 'ib_udf'
```

sqrt

Library: ib_udf

Added in: IB

Better alternative: Internal function SQRT()

Description: Returns the square root of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
sqrt (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION sqrt
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf'
```

srand

Library: ib_udf

Added in: 2.0

Description: Seeds the random number generator with the current time in seconds and then returns the first number. Multiple `srand()` calls within the same second will return the same value. This is exactly how `rand()` behaved before Firebird 2.0.

Result type: DOUBLE PRECISION

Syntax:

```
srand ()
```

Declaration:

```
DECLARE EXTERNAL FUNCTION srand
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_srand' MODULE_NAME 'ib_udf'
```

sright

Library: fbudf

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

Better alternative: Internal function RIGHT()

Description: Returns the rightmost *numchars* characters of the input string. Only works with 1-byte character sets.

Result type: VARCHAR(100)

Syntax:

```
sright (str, numchars)
```

Declaration:

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

string2blob

Library: fbudf

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

Better alternative: Internal function CAST()

Description: Returns the input string as a BLOB.

Result type: BLOB

Syntax:

```
string2blob (str)
```

Declaration:

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

strlen

Library: ib_udf

Added in: IB

Better alternatives: Internal functions BIT_LENGTH(), CHAR[ACTER]_LENGTH and OCTET_LENGTH()

Description: Returns the length of the argument string.

Result type: INTEGER

Syntax:

```
strlen (str)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION strlen
   CSTRING(32767)
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'IB_UDF_strlen' MODULE_NAME 'ib_udf'
```

substr

Library: ib_udf

Changed in: 1.0, 1.5.2, 2.0

Description: Returns a string's substring from *startpos* to *endpos*, inclusively. Positions are 1-based. If *endpos* is past the end of the string, substr returns all the characters from *startpos* to the end of the string. This function only works correctly with single-byte characters.

Result type: VARCHAR(*n*)

Syntax (unchanged):

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

Declaration:

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

The above declaration is from the file ib_udf2.sql. The **NULL** after the argument is an optional addition that became available in Firebird 2. If the argument is declared with the NULL keyword,

the engine will pass a `NULL` argument value unchanged to the function. This leads to a `NULL` result, which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL` is passed to the function as an empty string and the result is an empty string as well.

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- Before Firebird 2.0, the result type was CHAR(*n*).
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).
- In InterBase, `substr` returned `NULL` if *`endpos`* lay past the end of the string.

> **Tip**
>
> Although the function arguments are slightly different, consider using the internal SQL function SUBSTRING instead, for better compatibility and multi-byte character set support.

substrlen

Library: ib_udf

Added in: 1.0

Changed in: 1.5.2, 2.0

Better alternative: Internal function SUBSTRING()

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. This function only works correctly with single-byte characters.

Result type: VARCHAR(*n*)

Syntax:

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

Declaration:

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

The above declaration is from the file `ib_udf2.sql`. The **NULL** after the argument is an optional addition that became available in Firebird 2. If the argument is declared with the NULL keyword, the engine will pass a `NULL` argument value unchanged to the function. This leads to a `NULL` result,

which is correct. Without the NULL keyword (your only option in pre-2.0 versions), `NULL` is passed to the function as an empty string and the result is an empty string as well.

For more information about passing `NULL`s to UDFs, see the note at the end of this book.

Notes:

- Depending on how you declare it (see CSTRING note), this function can accept and return strings of up to 32767 characters.
- Before Firebird 2.0, the result type was CHAR(*n*).
- In Firebird 1.5.1 and below, the default declaration used CSTRING(80) instead of CSTRING(255).

> **Tip**
>
> Firebird 1.0 has also implemented the internal SQL function SUBSTRING, effectively rendering `substrlen` obsolete in the same version in which it was introduced. SUBSTRING also supports multi-byte character sets. In new code, use SUBSTRING.

tan

Library: ib_udf

Added in: IB

Better alternative: Internal function TAN()

Description: Returns an angle's tangent. The argument must be given in radians.

Result type: DOUBLE PRECISION

Syntax:

```
tan (angle)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION tan
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_tan' MODULE_NAME 'ib_udf'
```

tanh

Library: ib_udf

Added in: IB

Better alternative: Internal function TANH()

Description: Returns the hyperbolic tangent of the argument.

Result type: DOUBLE PRECISION

Syntax:

```
tanh (number)
```

Declaration:

```
DECLARE EXTERNAL FUNCTION tanh
   DOUBLE PRECISION
   RETURNS DOUBLE PRECISION BY VALUE
   ENTRY_POINT 'IB_UDF_tanh' MODULE_NAME 'ib_udf'
```

truncate, i64truncate

Library: fbudf

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

Changed in: 1.5, 2.1.3

Better alternative: Internal function TRUNC()

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

Result type: INTEGER / NUMERIC(18)

Syntax:

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

> **Caution**
>
> Both functions round to the nearest whole number that is lower than or equal to the argument. This means that negative numbers are also “truncated” downward. For instance, `truncate(-2.37)` returns `-3`. The internal function TRUNC, available since Firebird 2.1, always truncates toward zero.

> **Bug alert**
>
> Contrary to what's mentioned above, in versions 2.1, 2.1.1 and 2.1.2 anything between -1 and 0 is truncated to 0. This anomaly has been corrected in Firebird 2.1.3 and above.

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. From Firebird 2.0 onward you can also perform this update with ALTER EXTERNAL FUNCTION.

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` – specify 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.

Passing NULL to UDFs in Firebird 2

If a pre-2.0 Firebird engine must pass an SQL NULL argument to a user-defined function, it always converts it to a zero-equivalent, e.g. a numerical 0 or an empty string. The only exception to this rule are UDFs that make use of the "BY DESCRIPTOR" mechanism introduced in Firebird 1. The fbudf library uses descriptors, but the vast majority of UDFs, including those in Firebird's standard ib_udf library, still use the old style of parameter passing, inherited from InterBase.

As a consequence, most UDFs can't tell the difference between NULL and zero input.

Firebird 2 comes with a somewhat improved calling mechanism for these old-style UDFs. The engine will now pass NULL input as a null pointer to the function, **if** the function has been declared to the database with a NULL keyword after the argument(s) in question, e.g. like this:

```
declare external function ltrim
  cstring(255) null
  returns cstring(255) free_it
  entry_point 'IB_UDF_ltrim' module_name 'ib_udf';
```

This requirement ensures that existing databases and their applications can continue to function like before. Leave out the NULL keyword and the function will behave like it did under Firebird 1.5 and earlier.

Please note that you can't just add NULL keywords to your declarations and then expect every function to handle NULL input correctly. Each function has to be (re)written in such a way that NULLs are dealt with correctly. Always look at the declarations provided by the function implementor. For the functions in the ib_udf library, consult ib_udf2.sql in the Firebird UDF directory. Notice the 2 in the file name; the old-style declarations are in ib_udf.sql.

These are the ib_udf functions that have been updated to recognise NULL input and handle it properly:

- ascii_char
- lower
- lpad and rpad
- ltrim and rtrim
- substr and substrlen

Most ib_udf functions remain as they were; in any case, passing NULL to an old-style UDF is never possible if the argument isn't of a referenced type.

On a side note: don't use lower, .trim and substr* in new code; use the internal functions LOWER, TRIM and SUBSTRING instead.

"Upgrading" ib_udf functions in an existing database

If you are using an existing database with one or more of the functions listed above under Firebird 2, and you want to benefit from the improved NULL handling, run the script ib_udf_upgrade.sql against your database. It is located in the Firebird misc\upgrade\ib_udf directory.

Maximum number of indices in different Firebird versions

Between Firebird 1.0 and 2.0 there have been quite a few changes to the maximum number of indices per database table. The table below sums them all up.

Table A.2. Max. indices per table in Firebird 1.0 – 2.0

Page size	Firebird version(s)											
	1.0, 1.0.2			1.0.3			1.5.x			2.0.x		
	1 col	2 cols	3 cols	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	50	35	27
2048	65	65	65	126	101	84	126	101	84	101	72	56
4096	65	65	65	254	203	169	254	203	169	203	145	113
8192	65	65	65	510	408	340	257	257	257	408	291	227
16384	65	65	65	1022	818	681	257	257	257	818	584	454

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

0.9	10 Jul 2009	PV	First publication, based on the *Firebird 2.0 Language Reference Update* with almost all the changes for 2.1 added (roughly adding 50% to the size).
1.0	9 Dec 2010	PV	GLOBAL: Renamed all “Deprecated in” section headers to “Better alternative”. This also required editing the text immediately following the header and in some cases additional text in the section (if the “deprecation” was discussed in the section body). *Bookinfo*: Added 2.1.4 to covered versions. *Introduction :: Subject matter*: Added “Aggregate functions” to first list. *Introduction :: Versions covered*: Added 2.1.4. *Introduction :: Authorship*: Edited first paragraph. Added Frank Ingermann and Vlad Khorsun to contributor list. *Introduction*: Removed sections *Completeness* and *Miscellaneous notes*. *Data types and subtypes :: BLOB data type :: Text BLOB compatibility with VARCHAR*: Replaced this subsection, which was incorrect, with *Text BLOB support in functions and operators*. *Data types and subtypes :: BLOB data type :: Various enhancements*: Added information on `binary` mnemonic (new in 2.0) + extra example. *Data types and subtypes :: New collations :: A note on the UTF8 collations*: Added information on UNICODE_CI. *DDL statements :: COLLATION :: DROP COLLATION*: Edited Description. *DDL statements :: DATABASE :: CREATE DATABASE*: Moved Syntax one level up and added DIFFERENCE FILE clause. Added new subsection *DIFFERENCE FILE parameter*. *DDL statements :: DATABASE :: ALTER DATABASE*: Merged difference file clauses onto one line in Syntax. *DDL statements :: DOMAIN :: ALTER DOMAIN*: Added Warning about changing domains referred in PSQL code. *DDL statements :: FILTER :: DECLARE FILTER*: Edited Description. Added *`user_defined`* to Syntax. Added more info under Syntax block and made it an itemizedlist. Converted Tip to formalpara *User-defined mnemonics*. *DDL statements :: PROCEDURE :: CREATE PROCEDURE*: Added NOT NULL to syntax block; added comment about character sets to syntax block.

DDL statements :: PROCEDURE :: CREATE PROCEDURE :: Domains instead of datatypes: Renamed to *Domains supported in parameter and variable declarations.* Added Warning about changing domain definitions.
DDL statements :: PROCEDURE :: CREATE PROCEDURE: Added subsection *NOT NULL in variable and parameter declarations*.
DDL statements :: PROCEDURE :: ALTER PROCEDURE :: Domains instead of datatypes: Renamed to *Domains supported in parameter and variable declarations.*
DDL statements :: PROCEDURE :: ALTER PROCEDURE: Added subsection *NOT NULL in variable and parameter declarations*.
DDL statements :: TABLE :: CREATE TABLE :: GENERATED ALWAYS AS: Added Note about it not being supported in index definitions.
DDL statements :: TABLE :: CREATE TABLE: Added subsection *FOREIGN KEY without target column references PK*.
DDL statements :: TABLE :: ALTER TABLE: Added subsection *FOREIGN KEY without target column references PK*.
DDL statements :: TABLE :: ALTER TABLE: Added subsection *GENERATED ALWAYS AS*.
DDL statements :: TRIGGER :: CREATE TRIGGER: Added subsection *NOT NULL in variable declarations*.
DDL statements :: TRIGGER :: ALTER TRIGGER: Added subsection *NOT NULL in variable declarations*.
DDL statements :: VIEW :: CREATE VIEW :: Full SELECT syntax supported: Mentioned that in Fb 2.5 the column list becomes optional also for union views.
DDL statements :: VIEW :: CREATE VIEW :: PLAN subclause disallowed in 1.5: Changed title to *PLAN subclause disallowed in 1.5, reallowed in 2.0*.
DML statements :: DELETE: Corrected formal syntax (`columns -> values`). Corrected syntax note about WHERE CURRENT OF.
DML statements :: DELETE: Added subsection *COLLATE subclause for text BLOB columns*.
DML statements :: DELETE: Added subsection *Relation alias makes real name unavailable*.
DML statements :: DELETE :: RETURNING: Improved Description.
DML statements :: EXECUTE BLOCK: Added NOT NULL support for in/out/local PSQL vars (added “Changed in” formalpara, updated syntax block, added subsection).
DML statements :: INSERT: Corrected formal syntax (`column_list -> value_list`).
DML statements :: INSERT :: RETURNING: Improved Description.
DML statements :: MERGE: Mentioned CTE in description and created links.
DML statements :: SELECT :: Common Table Expressions: Edited Description, Syntax and Notes.
DML statements :: SELECT :: Table alias must be used if present: Renamed to *Relation alias makes real name unavailable* and moved to before *ROWS* subsection. Also changed Description and paragraph before last example.
DML statements :: UPDATE: Corrected formal syntax (`columns -> values`). Corrected syntax note about WHERE CURRENT OF.

DML statements :: UPDATE: Added subsection *COLLATE subclause for text BLOB columns*.
DML statements :: UPDATE: Added subsection *Relation alias makes real name unavailable*.
DML statements :: UPDATE :: RETURNING: Improved Description.
DML statements :: UPDATE OR INSERT: Corrected formal syntax (`columns -> values`). Edited first two subitems of second Note.
PSQL statements :: DECLARE: Added NOT NULL to Syntax. Added Syntax note about including a character set.
PSQL statements :: DECLARE :: DECLARE ... CURSOR: Edited first Note and placed it last. Added a subsequent note about the effects of variable changes during loop execution.
PSQL statements :: DECLARE :: DECLARE with DOMAIN instead of datatype: Added Warning about changing domain definitions.
PSQL statements :: DECLARE: Added subsection *NOT NULL in variable declaration*.
PSQL statements :: FOR SELECT ... INTO ... DO: Edited Syntax note and added a second note about the effects of variable changes during loop execution.
Context variables :: `CURRENT_CONNECTION`: Improved Description.
Context variables :: `CURRENT_TIME`: Edited description. Removed Note and added Notes formalpara.
Context variables :: `CURRENT_TIMESTAMP`: Edited description. Removed Note and added Notes formalpara.
Context variables :: `CURRENT_TRANSACTION`: Improved Description.
Context variables :: `'NOW'`: Removed Note and added Notes formalpara.
Operators and predicates :: || (string concatenator): New subsections *Text BLOB concatenation* and *Result type VARCHAR or BLOB*.
Operators and predicates :: || (string concatenator) :: Overflow checking: Corrected “Changed in” and Description.
Aggregate functions :: LIST(): Extended 1st and 2nd second listitems under Syntax. Inserted new listitem about BLOB support in 3rd position. Edited 5th (previously 4th) listitem. Added warning on truncation bug.
Aggregate functions :: MAX(): New section.
Aggregate functions :: MIN(): New section.
Internal functions: Replaced all occurrences of “obfuscate” in the function sections with “override”.
Internal functions :: ASCII_VAL(): Edited Syntax. Added listitem about `NULL`. Altered last listitem.
Internal functions :: ATAN2(): Replaced argument names `num1` and `num2` with `y` and `x`, respectively. Changed wording of 3rd Syntax note. Added two Notes.
Internal functions :: BIT_LENGTH(): Added formalparas “Changed in” and “BLOB support”. Edited Note after Syntax block and placed it after Description.
Internal functions :: CAST(): Edited *Changed in*, *Description* and *Syntax*. Worked BLOB into table. Added paragraphs and examples re. casting to a domain. Added formalpara “Casting BLOBs”.

Internal functions :: CHAR_LENGTH(), CHARACTER_LENGTH(): Added formalparas “Changed in” and “BLOB support”. Edited Note after Syntax block and placed it after Description.
Internal functions :: EXTRACT(): Edited Result type and everything following the Syntax block, except the *WEEK* subsection.
Internal functions :: HASH(): Mentioned full text BLOB support in Description.
Internal functions :: LEFT(): Edited Result type. Edited first listitem and inserted a new listitem before it (about BLOB support).
Internal functions :: LOWER(): Added “Changed in”. Edited Description (BLOB support, removed epithet “new”). Added BLOB as result type and corrected VAR(CHAR) -> (VAR)CHAR. Removed “new” from text under Declaration.
Internal functions :: LPAD(): Replaced `''` with “the empty string” in 2nd (now 4th) Syntax note. Inserted two new Syntax notes concerning BLOB support. Changed 1st sentence of Tip. Added Warning about possible high memory usage.
Internal functions :: MAXVALUE(): Mentioned full text BLOB support in Description.
Internal functions :: MINVALUE(): Mentioned full text BLOB support in Description.
Internal functions :: OCTET_LENGTH(): Added formalparas “Changed in” and “BLOB support”. Edited Note after Description.
Internal functions :: OVERLAY(): Edited Result type. Edited first listitem under Syntax and inserted another one before it, about BLOB support. Also added a listitem about `NULL` arguments. Added Warning about possible high memory usage.
Internal functions :: POSITION(): Added “(1-based)” to Description. Added two listitems after Syntax. Added Warning about possible high memory usage.
Internal functions :: RDB$GET_CONTEXT(): Replaced “general” with “global” (4x) in System namespace table.
Internal functions :: REPLACE(): Edited Result type. Inserted new first listitem under Syntax and edited the previous first (now 2nd) listitem. Added example with `NULL` first argument. Aligned arguments in examples. Corrected last `constant` element, which accidentally spanned three lines instead of a single word. Added Warning about possible high memory usage.
Internal functions :: RIGHT(): Edited Result type. Edited first listitem and inserted a new listitem before it (about BLOB support and bug). Added Warning about possible high memory usage.
Internal functions :: RPAD(): Replaced `''` with “the empty string” in 2nd (now 4th) Syntax note. Inserted two new Syntax notes concerning BLOB support. Changed 1st sentence of Tip. Added Warning about possible high memory usage.
Internal functions :: SUBSTRING(): Added 2.1 to “Changed in”. Changed Result type. Edited Syntax. Rewrote most everything between Syntax and Examples. Added Warning about possible high memory usage.
Internal functions :: TRIM(): Added “Changed in”. Edited Description and Syntax. Corrected and extended Result type. Added Notes formalpara. Added Warning about possible high memory usage.

Internal functions :: UPPER(): Added 2.1 to “Changed in”. Edited Description (BLOB support). Added BLOB as result type and corrected VAR(CHAR) -> (VAR)CHAR.
External functions :: `addDay`, `addHour`, `addMilliSecond`, `addMinute`, `addMonth`, `addSecond`, `addYear`: Added “Better alternative: Internal function DATEADD” formalpara.
External functions :: `addWeek`: Added formalpara “The DATEADD alternative”.
External functions :: `getExactTimestamp`: Edited “Better alternative” and Description.
External functions :: `log`: Changed `log` -> `log(x,y)` in Description.
External functions :: `rand`: Removed “the new function” from Description.
External functions :: `string2blob`: Added “Better alternative” formalpara.
Notes :: Understanding the WITH LOCK clause :: Syntax and behaviour: In table, aligned 1st column left, all rows top, and added periods to sentences in first two rows.
License Notice: Added Frank Ingermann and Vlad Khorsun as contributors. (C) end year now 2010.

Appendix C: License notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the “License”); you may only use this Documentation if you comply with the terms of this License. Copies of the License are available at http://www.firebirdsql.org/pdfmanual/pdl.pdf (PDF) and http://www.firebirdsql.org/manual/pdl.html (HTML).

The Original Documentation is titled *Firebird 2.1 Language Reference Update*.

The Initial Writers of the Original Documentation are: Paul Vinkenoog et al.

Writers and Editors of included PDL-licensed material (the “al.”) are: J. Beesley, Helen Borrie, Arno Brinkman, Frank Ingermann, Vlad Khorsun, Alex Peshkov, Nickolay Samofatov, Adriano dos Santos Fernandes, Dmitry Yemanov.