Firebird 2.5 – Обновление справочника языка SQL

Всё новое в языке SQL Firebird начиная с InterBase 6

Paul Vinkenoog

8 октября 2011 года, версия 1.1 — охватывает Firebird 2.5 и 2.5.1
(*Примечание переводчика:* русская версия охватывает Firebird до версии 2.5.2 включительно)

Содержание

- CREATE TABLE
- ALTER TABLE
- RECREATE TABLE

TRIGGER
- CREATE TRIGGER
- ALTER TRIGGER
- CREATE OR ALTER TRIGGER
- DROP TRIGGER
- RECREATE TRIGGER

VIEW
- CREATE VIEW
- ALTER VIEW
- CREATE OR ALTER VIEW
- RECREATE VIEW

7. Операторы DML

DELETE
- Использование COLLATE для столбцов с текстовым BLOB
- ORDER BY
- PLAN
- Использование алиаса делает недоступным использование полного имени таблицы
- RETURNING
- ROWS

EXECUTE BLOCK
- COLLATE в объявлениях переменных и параметров
- NOT NULL в объявлениях переменных и параметров
- Домены вместо типа данных
- TYPE OF COLUMN в объявлениях параметров и переменных

EXECUTE PROCEDURE

INSERT
- INSERT ... DEFAULT VALUES
- Предложение RETURNING
- Разрешено использовать UNION в операторе SELECT при вставке

MERGE

SELECT
- Агрегатные функции: Расширенный функционал
- Использование COLLATE для столбцов с текстовым BLOB
- Общие табличные выражения (“WITH ... AS ... SELECT”)
- Производные таблицы (“SELECT FROM SELECT”)
- FIRST и SKIP
- GROUP BY
- HAVING: Более строгие правила
- JOIN

CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSACTION
CURRENT_USER
DELETING
GDSCODE
INSERTING
NEW
'NOW'
OLD
ROW_COUNT
SQLCODE
SQLSTATE
UPDATING

12. Операторы и предикаты

Разрешено использование NULL как операнда

|| - конкатенация (сцепление) строк

Конкатенация текстовых BLOB

Результат типа VARCHAR или BLOB

Проверка переполнения

ALL

Разрешено использование NULL

UNION как подзапрос

ANY/SOME

Разрешено использование NULL

UNION как подзапрос

IN

Разрешено использование NULL

UNION как подзапрос

IS [NOT] DISTINCT FROM

NEXT VALUE FOR

SIMILAR TO

Создание регулярных выражений

SOME

13. Агрегатные функции

LIST()
MAX()
MIN()

14. Встроенные функции

ABS()
ACOS()
ASCII_CHAR()

ASCII_VAL()
ASIN()
ATAN()
ATAN2()
BIN_AND()
BIN_OR()
BIN_SHL()
BIN_SHR()
BIN_XOR()
BIT_LENGTH()
CAST()
CEIL(), CEILING()
CHAR_LENGTH(), CHARACTER_LENGTH()
CHAR_TO_UUID()
COALESCE()
COS()
COSH()
COT()
DATEADD()
DATEDIFF()
DECODE()
EXP()
EXTRACT()
    MILLISECOND
    WEEK
FLOOR()
GEN_ID()
GEN_UUID()
HASH()
IIF()
LEFT()
LN()
LOG()
LOG10()
LOWER()
LPAD()
MAXVALUE()
MINVALUE()
MOD()
NULLIF()
OCTET_LENGTH()
OVERLAY()
PI()

POSITION()
POWER()
RAND()
RDB$GET_CONTEXT()
RDB$SET_CONTEXT()
REPLACE()
REVERSE()
RIGHT()
ROUND()
RPAD()
SIGN()
SIN()
SINH()
SQRT()
SUBSTRING()
TAN()
TANH()
TRIM()
TRUNC()
UPPER()
UUID_TO_CHAR()

15. Внешние функции (UDF)
    abs
    acos
    addDay
    addHour
    addMilliSecond
    addMinute
    addMonth
    addSecond
    addWeek
    addYear
    ascii_char
    ascii_val
    asin
    atan
    atan2
    bin_and
    bin_or
    bin_xor
    ceiling
    cos
    cosh

Список таблиц

Глава 1

Введение

- Операторы PSQL (Procedural SQL — процедурный SQL, используется в хранимых процедурах, триггерах и выполнимых блоках);

Что содержит данный документ

Данный документ содержит в себе изменения в языке Firebird SQL начиная с InterBase 6 по Firebird 2.5.1. Он охватывает следующие области:

- Зарезервированные слова;
- Типы и подтипы данных;
- Операторы DDL (Data Definition Language - язык создания данных);
- Операторы DML (Data Manipulation Language - язык обращения с данными);
- Операторы управления транзакциями;
- Безопасность и операторы управления доступом;
- Контекстные переменные;
- Операторы и предикаты (утверждения);
- Агрегатные функции;
- Встроенные функции;
- UDF (User Defined Functions — функции, определённые пользователем. Также известные как внешние функции).

Полная документация по Firebird 2.5 SQL включает в себя:

- InterBase 6.0 beta SQL Reference (LangRef.pdf и/или SQLRef.html);
- Данный документ.

Вопросы, не связанные с SQL, не рассматриваются в данном документе. Это:

- Версии ODS;
- Список багов;
- Инсталляция и конфигурирование;
- Обновление, миграция и совместимость;
- Архитектура сервера;
- Функции API;
- Протоколы подключения;
- Приложения и утилиты.

Данные темы освещены в замечаниях к релизу (Release Notes). Вы можете найти их и другую документацию по следующей ссылке: Firebird Documentation Index (http://www.firebirdsql.org/en/documentation/).

Охватываемые версии

Этот документ охватывает все версии Firebird вплоть до 2.5.1.

Авторство

Большая часть этого документа написана основным автором. Остальное (2-3%) было взято из Release Notes (замечаниях к релизу) различных версий Firebird, которые, в свою очередь, содержат материалы из предшествующих источников, таких как Whatsnew (Что нового). Авторы и редакторы материала:

- J. Beesley;
- Helen Borrie;
- Arno Brinkman;
- Frank Ingermann;
- Vlad Khorsun;
- Alex Peshkov;
- Nickolay Samofatov;
- Adriano dos Santos Fernandes;
- Dmitry Yemanov.

Благодарности

Большую помощь по вопросам о новых возможностях Firebird оказали Vlad Khorsun, Adriano dos Santos Fernandes и Dmitry Yemanov. Электронная переписка, которую я вёл с ними, сделала этот справочник более читабельным и полным. Огромное Спасибо Вам!

Глава 2

Новое в Firebird 2.5

Данная глава содержит список дополнений и изменений к SQL в Firebird 2.5 и 2.5.1 для пользователей, обновляющихся с версии Firebird 2.1. Если у вас более ранняя версия Firebird или вы в первый раз устанавливаете его, то вам лучше пропустить этот раздел.

Зарезервированные и ключевые слова

Изменения начиная с Firebird 2.1:

- Новые зарезервированные слова: SIMILAR, SQLSTATE (2.5.1).
- Новые не зарезервированные слова: AUTONOMOUS, BIN_NOT, CALLER, CHAR_TO_UUID, COMMON, DATA, FIRSTNAME, GRANTED, LASTNAME, MAPPING, MIDDLENAME, OS_NAME, SOURCE, TWO_PHASE и UUID_TO_CHAR.
- Более не зарезервированные, но ключевые слова: ACTIVE, AFTER, ASC, ASCENDING, AUTO, BEFORE, COLLATION, COMMITTED, COMPUTED, CONDITIONAL, CONTAINING, CSTRING, DATABASE, DESC, DESCENDING, DESCRIPTOR, DO, DOMAIN, ENTRY_POINT, EXCEPTION, EXIT, FILE, GEN_ID, GENERATOR, IF, INACTIVE, INPUT_TYPE, ISOLATION, KEY, LENGTH, LEVEL, MANUAL, MODULE_NAME, NAMES, OPTION, OUTPUT_TYPE, OVERFLOW, PAGE, PAGE_SIZE, PAGES, PASSWORD, PRIVILEGES, PROTECTED, READ, RESERV, RESERVING, RETAIN, SCHEMA, SEGMENT, SHADOW, SHARED, SINGULAR, SIZE, SNAPSHOT, SORT, STABILITY, STARTING, STARTS, STATEMENT, STATISTICS, SUB_TYPE, SUSPEND, TRANSACTION, UNCOMMITTED, WAIT, WORK и WRITE.
- Более не зарезервированные и не ключевые слова: AUTODDL, BASE_NAME, BASED, BLOBEDIT, BUFFER, CHECK_POINT_LENGTH, COMPILETIME, CONTINUE, DB_KEY, DEBUG, DESCRIBE, DISPLAY, ECHO, EDIT, EVENT, EXTERN, FOUND, GOTO, GROUP_COMMIT_, HELP, IMMEDIATE, INDICATOR, INIT, INPUT, ISQL, LC_MESSAGES, LC_TYPE, LEV, LOG_BUFFER_SIZE, MAX_SEGMENT, MAXIMUM, MESSAGE, MINIMUM, NOAUTO, NUM_LOG_BUFFERS, OUTPUT, PAGELENGTH, PREPARE, PUBLIC, QUIT, RETURN, RUNTIME, SHELL, SHOW, SQLERROR, SQLWARNING, STATIC, TERMINATOR, TRANSLATE,

TRANSLATION, VERSION, WAIT_TIME и WHENEVER.

Дополнительно

Изменения начиная с Firebird 2.1:

- Шестнадцатеричные обозначения цифр
- Шестнадцатеричные обозначения «двоичных» строк

Типы и подтипы данных

Изменения начиная с Firebird 2.1:

- Тип данных SQL_NULL
- Набор символов GB18030, алиас для WIN_1258
- Сортировка UNICODE_CI_AI для UTF8, сортировка GB18030 для GB18030

Язык определения данных (DDL)

Изменения начиная с Firebird 2.1:

- ALTER CHARACTER SET (установка сортировки по умолчанию для набора символов)
- Параметр NUMERIC-SORT для юникодовых сортировок
- Сортировка по умолчанию для базы данных
- Сервер классической архитектуры: изменения хранимых процедур становятся сразу же видимыми для всех активных клиентов
- ALTER COLUMN для вычисляемых (computed) столбцов
- ALTER COLUMN ... TYPE — выполняется, даже если столбец используется в триггере или хранимой процедуре
- Представления могут содержать запросы из хранимых процедур
- Представления могут выводить имена столбцов из производных таблиц (derived tables) или включённых в оператор GROUP BY
- Список столбцов для представлений с оператором UNION теперь не обязателен
- ALTER VIEW
- CREATE OR ALTER VIEW

Язык обращения с данными (DML)

Изменения начиная с Firebird 2.1:

- Оператор UPDATE: изменение семантики SET

Операторы PSQL

Изменения начиная с Firebird 2.1:

- TYPE OF COLUMN в объявлениях переменных и параметров
- EXECUTE STATEMENT
  - улучшение производительности
  - WITH {AUTONOMOUS|COMMON} TRANSACTION
  - WITH CALLER PRIVILEGES
  - ON EXTERNAL [DATA SOURCE]
  - AS USER, PASSWORD и ROLE
  - параметризованные запросы
- IN AUTONOMOUS TRANSACTION
- Подзапросы как PSQL выражения

Безопасность и управление доступом к данным

Изменения начиная с Firebird 2.1:

- ALTER ROLE
- Предложение GRANTED BY
- REVOKE ALL ON ALL
- Роль RDB$ADMIN
- AUTO ADMIN MAPPING
- Команды управления пользователями в SQL
  - CREATE USER
  - ALTER USER
  - DROP USER

Контекстные переменные

Изменения начиная с Firebird 2.1:

- Коды SQLCODE устарели (2.5.1)
- Контекстная переменная SQLSTATE (2.5.1)

Операторы и предикаты

Изменения начиная с Firebird 2.1:

- SIMLAR TO: регулярные выражения

Агрегатные функции

Изменения начиная с Firebird 2.1:

- Оператор LIST() может содержать любой текстовый разделитель

Встроенные функции

Изменения начиная с Firebird 2.1:

- CAST() как TYPE OF COLUMN
- DATEADD: Новый элемент WEEK (неделя). Разрешено использование этого элемента с типом данных DATE
- DATEDIFF: Новый элемент WEEK. Разрешено использование этого элемента с типом данных DATE
- CHAR_TO_UUID()
- Улучшено выполнение LOG()
- Улучшено выполнение LOG10()
- LPAD() возвращает тип VARCHAR правильной длины
- RPAD() возвращает тип VARCHAR правильной длины
- UUID_TO_CHAR()

Глава 3

Зарезервированные и ключевые слова

Зарезервированные слова являются частью языка SQL Firebird. Они не могут использоваться в качестве идентификаторов (также, как и в имени таблицы или хранимой процедуры). Исключение составляют квотированные (заключённые в двойные кавычки) имена в 3-м диалекте — однако, вы должны избегать этого, если не имеете веских оснований.

Ключевые слова также являются частью языка. У них есть особое значение при использовании в определённом контексте, но они не зарезервированы для собственного и монопольного использования Firebird. Их можно использовать в качестве идентификаторов без двойных кавычек.

Далее приводятся изменения начиная с InterBase 6. Полный перечень зарезервированных и ключевых слов в Firebird 2.5 можно найти в Приложении B.

Добавления начиная с InterBase 6

Вновь зарезервированные слова

Зарезервированные слова, добавленные в Firebird:

BIGINT
BIT_LENGTH
BOTH
CASE
CLOSE
CONNECT
CROSS
CURRENT_CONNECTION
CURRENT_ROLE
CURRENT_TRANSACTION
CURRENT_USER
DISCONNECT
FETCH
GLOBAL
INSENSITIVE
LEADING
LOWER

OPEN
RECREATE
RECURSIVE
ROW_COUNT
ROWS
SAVEPOINT
SENSITIVE
SIMILAR
SQLSTATE (2.5.1)
START
TRAILING
TRIM

Новые не зарезервированные ключевые слова

Приведённые ниже слова добавлены в Firebird как не зарезервированные ключевые слова. Более половины из них составляют имена внутренних функций, добавленных начиная с версии Firebird 2.1.

ABS
ACCENT
ACOS
ALWAYS
ASCII_CHAR
ASCII_VAL
ASIN
ATAN
ATAN2
AUTONOMOUS
BACKUP
BIN_AND
BIN_OR
BIN_NOT
BIN_SHL
BIN_SHR
BIN_XOR
BLOCK
BREAK
CALLER
CEIL
CEILING
CHAR_TO_UUID
COALESCE

COLLATION
COMMENT
COMMON
COS
COSH
COT
DATA
DATEADD
DATEDIFF
DECODE
DELETING
DIFFERENCE
EXP
FIRST
FIRSTNAME
FLOOR
GEN_UUID
GENERATED
GRANTED
HASH
IIF
INSERTING
LAST
LASTNAME
LEAVE
LIST
LN
LOCK
LOG
LOG10
LPAD
MAPPING
MATCHED
MATCHING
MAXVALUE
MIDDLENAME
MILLISECOND
MINVALUE
MOD
NEXT
NULLIF
NULLS
OS_NAME

OVERLAY
PAD
PI
PLACING
POWER
PRESERVE
RAND
REPLACE
RESTART
RETURNING
REVERSE
ROUND
RPAD
SCALAR_ARRAY
SEQUENCE
SIGN
SIN
SINH
SKIP
SOURCE
SPACE
SQRT
SUBSTRING
TAN
TANH
TEMPORARY
TRUNC
TWO_PHASE
WEEK
UPDATING
UUID_TO_CHAR

Удалено начиная с InterBase 6

Более не резервируемые, но всё ещё ключевые слова

Приведённые ниже слова более не являются зарезервированными в Firebird 2.5, но по-прежнему являются ключевыми словами:

ACTION
ACTIVE
AFTER

ASC
ASCENDING
AUTO
BEFORE
CASCADE
COLLATION
COMMITTED
COMPUTED
CONDITIONAL
CONTAINING
CSTRING
DATABASE
DESC
DESCENDING
DESCRIPTOR
DO
DOMAIN
ENTRY_POINT
EXCEPTION
EXIT
FILE
FREE_IT
GEN_ID
GENERATOR
IF
INACTIVE
INPUT_TYPE
ISOLATION
KEY
LENGTH
LEVEL
MANUAL
MODULE_NAME
NAMES
OPTION
OUTPUT_TYPE
OVERFLOW
PAGE
PAGE_SIZE
PAGES
PASSWORD
PRIVILEGES
PROTECTED

READ
RESERV
RESERVING
RESTRICT
RETAIN
ROLE
SCHEMA
SEGMENT
SHADOW
SHARED
SINGULAR
SIZE
SNAPSHOT
SORT
STABILITY
STARTING
STARTS
STATEMENT
STATISTICS
SUB_TYPE
SUSPEND
TRANSACTION
TYPE
UNCOMMITTED
WAIT
WEEKDAY
WORK
WRITE
YEARDAY

Более не резервируемые, не ключевые слова

Приведённые ниже слова более не являются зарезервированными в Firebird 2.5 и не являются ключевыми словами:

AUTODDL
BASE_NAME
BASED
BASENAME
BLOBEDIT
BUFFER
CACHE
CHECK_POINT_LEN

CHECK_POINT_LENGTH
COMPILETIME
CONTINUE
DB_KEY
DEBUG
DESCRIBE
DISPLAY
ECHO
EDIT
EVENT
EXTERN
FOUND
GOTO
GROUP_COMMIT_
GROUP_COMMIT_WAIT
HELP
IMMEDIATE
INDICATOR
INIT
INPUT
ISQL
LC_MESSAGES
LC_TYPE
LEV
LOG_BUF_SIZE
LOG_BUFFER_SIZE
LOGFILE
MAX_SEGMENT
MAXIMUM
MESSAGE
MINIMUM
NOAUTO
NUM_LOG_BUFFERS
NUM_LOG_BUFS
OUTPUT
PAGELENGTH
PREPARE
PUBLIC
QUIT
RAW_PARTITIONS
RETURN
RUNTIME
SHELL

SHOW
SQLERROR
SQLWARNING
STATIC
TERMINATOR
TRANSLATE
TRANSLATION
VERSION
WAIT_TIME
WHENEVER

Некоторые из этих слов до сих пор имеют особое значение в ESQL и/или ISQL.

Что может быть зарезервировано в будущих версиях

Приведённые ниже слова не являются зарезервированными в Firebird 2.5, но лучше избегать их использования в качестве идентификаторов, так как они, вероятно, станут зарезервированными — или будут добавлены как ключевые слова в будущих версиях:

BOOLEAN
FALSE
TRUE
UNKNOWN

Глава 4

Элементы языка

-- Однострочный комментарий

Доступно: DSQL, PSQL

Добавлено: 1.0

Изменено: 1.5

Описание: Строка, начинающаяся с "--" (двух тире), является комментарием и будет проигнорирована. Это также даёт возможность легко и быстро закомментировать строку SQL.

Начиная с версии Firebird 1.5 "--" может находиться в любом месте строки, например, после оператора SQL. Весь текст начиная с двойного тире до конца строки игнорируется.

Пример:

```
-- Таблица для хранения информации о клиентах:
CREATE TABLE CUSTOMERS (
NAME VARCHAR(32),
ADDED_BY VARCHAR(24),
CUSTNO VARCHAR(8),
PURCHASES INTEGER              -- количество покупок
)
```

Примечание: Второй комментарий разрешается только в версиях Firebird 1.5 и выше.

Шестнадцатеричные обозначения для цифр

Доступно: DSQL, PSQL

Добавлено: 2.5

Описание: Начиная с версии Firebird 2.5 целые числа могут быть записаны в шестнадцатеричном виде. Числа с 1-8 шестнадцатеричными цифрами будут интерпретироваться как INTEGER, а числа с 9-16 шестнадцатеричными цифрами как BIGINT.

Синтаксис:

0{x|X}<hexdigits>
<hexdigits> ::= 1–16 of <hexdigit>
<hexdigit> ::= one of 0..9, A..F, a..f

Пример:

```
SELECT 0X6FAA0D3 FROM RDB$DATABASE
    -- возвращает 117088467

SELECT 0X4F9 FROM RDB$DATABASE
    -- возвращает 1273

SELECT 0X6E44F9A8 FROM RDB$DATABASE
    -- возвращает 1850014120

SELECT 0X9E44F9A8 FROM RDB$DATABASE
    -- возвращает 1639646808 (INTEGER)

SELECT 0X09E44F9A8 FROM RDB$DATABASE
    -- возвращает 2655320488 ( BIGINT)

SELECT 0X28ED678A4C987 FROM RDB$DATABASE
    -- возвращает 720001751632263

SELECT 0XFFFFFFFFFFFFFFFF FROM RDB$DATABASE
    -- возвращает -1
```

Диапазоны значений:

- Шестнадцатеричные числа в диапазоне 0**..**7FFF FFFF являются положительными числами INTEGER с десятичными значениями 0**..**2147483647. Вы можете привести эти числа к типу BIGINT, добавляя спереди нули, чтобы довести общее число шестнадцатеричных цифр до девяти и выше, но это только меняет их тип, а не значения.
- Шестнадцатеричные числа в диапазоне 8000 0000**..**FFFF FFFF требуют особого внимания:

  - Числа с восемью шестнадцатеричными цифрами, например, в 0x9E44F9A8, интерпретируются как 32-битные целые (INTEGER) значения. Так как старший бит (бит знака) установлен, то они

соответствуют отрицательным десятичным числам в диапазоне -2147483648**..**-1.

- При добавлении одного или нескольких нулей, например, 0x09E44F9A8, они будут интерпретироваться как 64-разрядные числа BIGINT в диапазоне 0000 0000 8000 0000**..**0000 0000 FFFF FFFF. Так как знаковый бит не установлен, они соответствуют положительным десятичным числам в диапазоне 2147483648**..**4294967295.

Таким образом в этом диапазоне (и только в нём) добавление впереди ничего не значащего нуля радикально меняет значение числа. Вы должны помнить об этом.

- Шестнадцатеричные числа в диапазоне 0000 0000**..**7FFF FFFF FFFF FFFF соответствуют положительным значениям BIGINT.
- Шестнадцатеричные числа в диапазоне 8000 0000 0000 0000**..**FFFF FFFF FFFF FFFF соответствуют отрицательным значениям BIGINT.

Шестнадцатеричные обозначения для «двоичных» строк

Доступно: DSQL, PSQL

Добавлено: 2.5

Описание: Начиная с версии Firebird 2.5 строковые литералы (буквы) могут быть записаны в шестнадцатеричном виде. Каждая пара шестнадцатеричных цифр определяет байт в строке. Строка, записанная в таком виде, по умолчанию имеет набор символов OCTETS, но вы можете заставить сервер интерпретировать её иначе с помощью введённого в употреблении синтаксиса.

Синтаксис:

```
<hexstring> ::= an even number of <hexdigit>
<hexdigit> ::= one of 0..9, A..F, a..f
```

Пример:

```
SELECT x'4E657276656E' FROM RDB$DATABASE
    -- возвращает 4E657276656E, а 6-байтовая 'binary' строка

SELECT _ASCII x'4E657276656E' FROM RDB$DATABASE
    -- возвращает 'Nerven' (та же самая последовательность, интерпретированная как ASCII текст)

SELECT _ISO8859_1 x'53E46765' FROM RDB$DATABASE
    -- возвращает 'Säge' (4 символа, 4 байта)
```

```
SELECT _UTF8 x'53C3A46765' FROM RDB$DATABASE
    -- возвращает 'Säge' (4 символа, 5 байт)
```

Примечания:

- В примерах приведён результат запроса до вывода на клиентский интерфейс в виде двоичных строк, выведенных на экран пользователю. Isql, со своей стороны, использует прописные буквы A-F. FlameRobin использует строчные буквы. Другие клиентские программы могут иметь свои правила отображения, например, с пробелами между байтами: '4E 65 72 76 65 6E'.
- Шестнадцатеричное представление позволяет вставлять любой байт (включая 00) в любом месте строки. Однако, если надо привести такую строку к иному набору символов, чем OCTETS, вы должны чётко знать, что последовательность байта допустима для целевого набора символов.

Сокращённое приведение типов даты и времени (datetime)

Доступно: DSQL, ESQL, PSQL

Добавлено: IB (InterBase)

Описание: При преобразовании строки в тип DATE, TIME или TIMESTAMP, Firebird позволяет использовать сокращённое “C-style” приведение типов. Эта функция уже существовала в InterBase 6, но так и не была должным образом документирована.

Синтаксис:

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

Пример:

```
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';
```

Примечание: Обратите внимание, что эти сокращённые выражения вычисляются сразу же во время синтаксического анализа, т.е. как будто оператор

уже подготовлен к выполнению. Таким образом, даже если запрос выполняется несколько раз, значение, например, для “timestamp 'now'” не изменится, независимо от того, сколько времени проходит. Если вам нужно получать нарастающее значение времени (т.е. оно должно быть оценено при каждом вызове), используйте полный синтаксис оператора CAST.

- См. также: CAST()

Конструкция CASE

Доступно: DSQL, PSQL

Добавлено: 1.5

Описание: Оператор CASE возвращает только одно значение из нескольких возможных. Есть два синтаксических варианта:

- Простой CASE, сравнимый с Pascal case или C switch;
- Поисковый CASE, который работает как серия операторов “if ... else if ... else if”.

Простой CASE

Синтаксис:

```
CASE <test-expr>
    WHEN <expr> THEN result
    [WHEN <expr> THEN result ...]
[ELSE defaultresult]
END
```

При использовании этого варианта *<test-expr>* сравнивается с *<expr> 1*, *<expr> 2* и т.д. до тех пор, пока не будет найдено совпадение, и тогда возвращается соответствующий результат. Если совпадений не найдено, то возвращается *defaultresult* из ветви ELSE. Если нет совпадений и ветвь ELSE отсутствует, возвращается значение NULL.

Совпадение эквивалентно оператору «=», т.е. если *<test-expr>* имеет значение NULL, то он не соответствует ни одному из <expr>, даже тем, которые имеют значение NULL.

Полученные результаты не должны быть буквальным значением: они могут быть полями или именами переменных, сложными выражениями, или иметь значение NULL.

Сокращённый вид простого оператора CASE используется в функции DECODE, доступной начиная с версии Firebird 2.1.

Пример:

```
SELECT NAME,
    AGE,
    CASE UPPER(SEX)
        WHEN 'M' THEN 'Male'
        WHEN 'F' THEN 'Female'
    ELSE 'Unknown'
    END SEX,
    RELIGION
FROM PEOPLE
```

Поисковый CASE

Синтаксис:

```
CASE
    WHEN <bool_expr> THEN result
    [WHEN <bool_expr> THEN result …]
[ELSE defaultresult]
END
```

Здесь <bool_expr> выражение, которое даёт тройной логический результат: TRUE, FALSE или NULL. Первое выражение, возвращающее TRUE, определяет результат. Если нет выражений, возвращающих TRUE, то в качестве результата берётся *defaultresult* из ветви ELSE. Если нет выражений, возвращающих TRUE, и ветвь ELSE отсутствует, результатом будет NULL.

Как и в простом операторе CASE, результаты не должны быть буквальным значением: они могут быть полями или именами переменных, сложными выражениями, или иметь значение NULL.

Пример:

```
CANVOTE = CASE
            WHEN AGE >= 18 THEN 'Yes'
            WHEN AGE < 18 THEN 'No'
          ELSE 'Unsure'
          END;
```

Глава 5

Типы и подтипы данных

Тип данных BIGINT

Добавлено: 1.5

Описание: BIGINT это SQL99-совместимый 64 битный целочисленный тип данных. Он доступен только в 3-м диалекте.

Числа типа BIGINT находятся в диапазоне -2^{63} .. $2^{63} - 1$, или -9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807.

Начиная с Firebird 2.5 числа типа BIGINT могут быть заданы в шестнадцатеричном виде с 9 — 16 шестнадцатеричными цифрами. Более короткие шестнадцатеричные числа интерпретируются как тип данных INTEGER.

Пример:

```
 CREATE TABLE WHOLELOTTARECORDS (
        ID BIGINT NOT NULL PRIMARY KEY,
        DESCRIPTION VARCHAR(32)
)

INSERT INTO MYBIGINTS VALUES (
        -236453287458723,
        328832607832,
        22,
        -56786237632476,
        0X6F55A09D42, -- 478177959234
        0X7FFFFFFFFFFFFFFF, -- 9223372036854775807
        0XFFFFFFFFFFFFFFFF, -- -1
        0X80000000, -- -2147483648, т.е. INTEGER
        0X080000000, -- 2147483648, т.е. BIGINT
        0XFFFFFFFF, -- -1, т.е. INTEGER
        0X0FFFFFFFF -- 4294967295, т.е. BIGINT
)
```

Шестнадцатеричный INTEGER автоматически приводится к типу BIGINT перед вставкой в таблицу. Однако это происходит *после* установки численного значения, так `0x80000000` (8 цифр) и `0x080000000` (9 цифр) будут сохранены в разных форматах. Более подробную информацию об этом смотрите в разделе Шестнадцатеричные обозначения для цифр в параграфе Элементы языка .

Примечание переводчика: значение `0x80000000` (8 цифр) будет сохранено в формате INTEGER, а `0x080000000` (9 цифр) как BIGINT.

Тип данных BLOB

Поддержка текстовых BLOB в функциях и операторах

Изменено: 2.1, 2.1.5, 2.5.1

Описание: Текстовые BLOB любой длины и с любым набором символов (включая multi-byte) теперь могут быть использованы практически в любыми встроенными функциями и операторами. В некоторых случаях существуют ограничения или ошибки («*баги*»).

Уровень поддержки:

- Полная поддержка для операторов:
  - = (присвоение);
  - =, <>, <, <=, >, >= (сравнение);
  - || (конкатенация);
  - BETWEEN, IS [NOT] DISTINCT FROM, IN, ANY|SOME и ALL.

- Поддержка для STARTING [WITH], LIKE и CONTAINING:
  - В версиях 2.1–2.1.4 и 2.5 ошибка возникает, если второй операнд имеет размер 32 КБ или больше, или если первый операнд BLOB с набором символов NONE, а второй операнд BLOB любой длины и и с другим набором символов;
  - В версии 2.5.1 и выше (а также 2.1.5 и выше), каждый операнд может быть BLOB любой длины и с любым набором символов.

- SELECT DISTINCT, ORDER BY и GROUP BY работают с BLOB ID, а не с содержимым блоба. Это улучшает выполнение этих операторов, хотя использование BLOB ID и бесполезно, за исключением того, что SELECT DISTINCT отсеивает многократные NULL, если они присутствуют. Оператор GROUP BY ведет себя странно при объединении одинаковых строк, если они

являются смежными, но не тогда, когда они обособленны друг от друга.

- Проблемы с BLOB во внутренних и агрегатных функциях рассматриваются в соответствующих разделах.

Различные улучшения

Изменено: 2.0

Описание: В Firebird 2.0 были реализованы несколько усовершенствований для текстовых BLOB:

- Предложение DML COLLATE больше не поддерживается;
- Операция равенства при сравнении может быть выполнена на все содержимое BLOB;
- Преобразования набора символов возможны при назначении BLOB для BLOB или строки для BLOB. При определении двоичных BLOB теперь может быть использован мнемонический двоичный тип данных вместо целого.

Примеры:

```
SELECT NAMEBLOB
    FROM MYTABLE
WHERE NAMEBLOB COLLATE PT_BR = 'Joậo'

CREATE TABLE MYPICTURES (
);
    ID INTEGER NOT NULL PRIMARY KEY,
    TITLE VARCHAR(40),
    DESCRIPTION VARCHAR(200),
    PICTURE BLOB SUB_TYPE BINARY
);
```

Тип данных SQL_NULL

Добавлено: 2.5

Описание: Тип данных SQL_NULL представляет небольшой или даже вообще не представляет интереса для конечного пользователя. Он не может содержат данные, только состояние: NULL или NOT NULL. Кроме того он не

может быть использован при объявлении полей таблицы, переменных или PSQL параметров. В настоящее время его единственная цель заключается в поддержке синтаксиса “? IS NULL” в SQL операторах с позиционными параметрами. Разработчики приложений могут использовать это при построении запросов, которые содержат один или несколько дополнительных условий для фильтрации.

Синтаксис: Если запрос, содержащий следующий предикат подготовлен:

? <*op*> NULL

Firebird опишет параметр ('?') как тип SQL_NULL. <op> может быть любым оператором сравнения, но единственный, который имеет смысл на практике это "IS" (и, возможно, в некоторых редких случаях, "NOT IS").

Обоснование

Сам по себе запрос с "WHERE ? IS NULL " не имеет много смысла. Вы можете использовать такой параметр, как вкл./выкл., но это вряд ли обосновывает введение нового типа данных. В конце концов, такие переключатели могут также быть созданы с CHAR, SMALLINT или другим типом параметра. Причина добавления типа SQL_NULL состоит в том, что разработчики приложений, инструментов подключения к БД, драйверов и т.д. хотят поддерживать запросы с дополнительными фильтрами, такими, как эти:

```
SELECT
        AU.MAKE, AU.MODEL, AU.WEIGHT,
        AU.PRICE, AU.IN_STOCK
FROM AUTOMOBILES AU
WHERE (AU.MAKE = :MAKE OR :MAKE IS NULL)
        AND (AU.MODEL = :MODEL OR :MODEL IS NULL)
        AND (AU.PRICE <= :MAXPRICE OR :MAXPRICE IS NULL)
```

Идея состоит в том, что конечный пользователь может дополнительно ввести варианты для параметров *:make*, *:model* и *:maxprice*. Там, где сделан выбор, должен быть применен соответствующий фильтр. Везде, где значение параметра не установлено (NULL), никакой фильтрации по этому атрибуту не должно быть. Если все параметры не установлены, то должны быть показана вся таблица AUTOMOBILES.

К сожалению именованные параметры, такие как *:make*, *:model* и *:maxprice*. существует только на уровне приложений. Прежде чем запрос передается серверу Firebird для подготовки, он должен быть преобразован в такую форму:

```
SELECT
        AU.MAKE, AU.MODEL, AU.WEIGHT,
```

```
        AU.PRICE, AU.IN_STOCK
FROM AUTOMOBILES AU
WHERE (AU.MAKE = ? OR ? IS NULL)
        AND (AU.MODEL = ? OR ? IS NULL)
        AND (AU.PRICE <= ? OR ? IS NULL)
```

Вместо трех именованных параметров, каждый из которых используется два раза, мы теперь имеем шесть позиционных параметров. Нет никакого способа, которым Firebird может определить, относятся ли некоторые из них фактически к тому же параметру прикладного уровня. (Тот факт, что, как в этом примере, они оказались в пределах одной пары скобок ничего не значит). Это, в свою очередь, означает, что Firebird не может определить тип данных параметра “? is null”. Эта последняя задача может быть решена путем приведения типов:

```
SELECT
        AU.MAKE, AU.MODEL, AU.WEIGHT,
        AU.PRICE, AU.IN_STOCK
FROM AUTOMOBILES AU
WHERE (AU.MAKE = ? OR CAST(? AS TYPE OF COLUMN
AU.MAKE) IS NULL)
        AND (AU.MODEL = ? OR CAST(? AS TYPE OF COLUMN
AU.MODEL) IS NULL)
        AND (AU.PRICE <= ? OR CAST(? AS TYPE OF COLUMN
AU.PRICE) IS NULL)
```

Но это довольно громоздко. И еще один вопрос: там, где параметр для фильтра не NULL, его значение будет передано серверу два раза: один раз в параметр для сравнения с данными столбца таблицы, и второй раз для проверки на NULL. А это небольшие, но потери производительности. Но единственной альтернативой является создание не менее восьми отдельных запросов (2 в степени числа дополнительных фильтров), т.е. ещё большие потери производительности. Для решения этой проблемы и был введён тип данных SQL_NULL.

Практическое использование

Примечание: Следующее обсуждение предполагает знакомство с Firebird API и принятия параметров через XSQLVAR структуру. Читатели без этих знаний в любом случае не будут иметь дело с типом данных SQL_NULL и могут пропустить этот раздел.

Обычно приложение передает параметризованные запросы на сервер в виде «?». Это делает невозможным слияние пары «одинаковых» параметров в один. Так, например, для двух фильтров (двух именованных параметров) необходимо четыре

позиционных параметра:

```
SELECT
        SH.SIZE, SH.COLOUR, SH.PRICE
FROM SHIRTS SH
WHERE (SH.SIZE = ? OR ? IS NULL)
        AND (SH.COLOUR = ? OR ? IS NULL)
```

После выполнения isc_dsql_describe_bind() sqltype 2-го и 4-го параметров устанавливается в SQL_NULL. Как уже говорилось выше, сервер Firebird не имеет никакой информации об их связи с 1-м и 3-м параметрами - это полностью прерогатива программиста. Как только значения для 1-го и 3-го параметров были установлены (или заданы как NULL) и запрос подготовлен, каждая пара XSQLVARs должна быть заполнена следующим образом:

Пользователь задал параметры

- Первый параметр (сравнение значений): set *sqldata в переданное значение и *sqlind в 0 (для NOT NULL);
- Второй параметр (проверка на NULL): set *sqldata в NULL (не SQL_NULL) и *sqlind в 0 (для NOT NULL).

Пользователь не задал параметры (NULL)

- Оба параметра (проверка на NULL): set *sqldata в NULL (не SQL_NULL) и *sqlind в -1 (индикация NULL).

Другими словами: Значение параметра сравнения всегда устанавливается как обычно. SQL_NULL параметр устанавливается также, за исключением случая, когда sqldata передаётся как NULL.

Новые наборы символов

Добавлено: 1.0, 1.5, 2.0, 2.1, 2.5

Ниже приведена таблица с добавленными в Firebird наборами символов

Таблица 5.1. Новые наборы символов в Firebird

Название	Макс. байт/символ	Язык	Добавлено в
CP943C	2	Japanese	2.1

DOS737	1	Greek	1.5
DOS775	1	Baltic	1.5
DOS858	1	= DOS850 plus € sign	1.5
DOS862	1	Hebrew	1.5
DOS864	1	Arabic	1.5
DOS866	1	Russian	1.5
DOS869	1	Modern Greek	1.5
GB18030	4	Chinese	2.5
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	Все	2.0
WIN1255	1	Hebrew	1.5
WIN1256	1	Arabic	1.5
Название	**Макс. байт/символ**	**Язык**	**Добавлено в**
WIN1257	1	Baltic	1.5
WIN1258	1	Vietnamese	2.0
WIN_1258 (алиас для WIN1258)	1	Vietnamese	2.5

(*) В Firebird 1.5 UTF8 является псевдонимом для UNICODE_FSS. Этот набор символов имеет некоторые внутренние проблемы. Начиная с Firebird 2.0 UTF-8 представляет собой самостоятельный набор символов без недостатков UNICODE_FSS.

Изменение обработки набора символов NONE

Изменено: 1.5.1

Описание: В Firebird 1.5.1 улучшено преобразование набора символов NONE в и из полей или переменных с другим набором символов, что уменьшило количество ошибок транслитерации. Более детальная информация приведена в разделе Примечания Набор символов NONE воспринимается «как есть» (“as is”) в конце этого документа.

Новое в сортировках

Добавлено: 1.0, 1.5, 1.5.1, 2.0, 2.1, 2.5

В таблице 5.2 приведены сортировки, добавленные в Firebird. Информация в колонке «Подробно» взята из Release Notes (Примечаниях к релизу) и других документов. Информация в этой колонке, вероятно, неполная; сортировки в пустых полях этой колонки могут быть регистрочувствительными (case insensitive — ci), акцентно нечувствительными / диакритически чувствительными (accent insensitive — ai) или словарно отсортированными (dictionary-sorted - dic).

Обратите внимание на то, что для новых наборов символов умолчательный порядок сортировки - двоичный - не приводится, т.к. это не добавило бы значимой информации.

Таблица 5.2. Новые сортировки в Firebird

Набор символов	Сортировка	Язык	Подробно	Добавлено в
CP943C	CP943C_UNICODE	Japanese		2.1
GB18030	GB18030_UNICODE	Chinese		2.5
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	Все		2.0
	UNICODE		dic	2.0
	UNICODE_CI		ci	2.1
	UNICODE_CI_AI		ci, ai	2.5
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, 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
	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

Примечание к сортировкам UTF8

Сортировка UCS_BASIC сортирует Юникод в следующем порядке: A, B, a, b, á... То есть так же, как и без указания сортировки UTF8. UCS_BASIC был добавлен для соответствия со стандарту SQL.

Сортировка UNICODE использует алгоритм UCA (Unicode Collation Algorithm): a, A, á, b, B...

UNICODE_CI является регистрочувствительной. При поиске, напрмер, 'Apple' также будут найдены 'apple', 'APPLE' и 'aPPLe'.

UNICODE_CI_AI диакритически чувствительна. При использовании этой сортировки 'APPLE' эквивалентен 'Applé'.

Юникод сортировки для всех наборов символов

Добавлено: 2.1

Firebird теперь поддерживает UNICODE сортировки для всех стандартных наборов символов. Однако, за исключением перечисленных в таблице новых сортировок в предыдущем разделе, эти параметры сортировки не становятся автоматически доступными в базах данных. Для использования они должны быть добавлены при помощи оператора CREATE COLLATION, например:

CREATE COLLATION **ISO8859_1_UNICODE** FOR **ISO8859_1**

Имена всех новых юникодных сортировок получаются добавлением к имени набора символов _UNICODE. (Встроенные юникодные сортировки для UTF8 для сортировки являются исключением из этого правила). Они определены, наряду с другими параметрами сортировки, в файле манифеста fbintl.conf подкаталога intl сервера Firebird.

Сортировки также можно зарегистрировать и под другим, заданным вами, именем, например:

CREATE COLLATION **RU_UNI** FOR **WIN1251** FROM EXTERNAL ('**WIN1251_UNICODE**')

Более полную информацию смотрите в разделе CREATE COLLATION.

Глава 6

Операторы DDL

Операторы в этой главе сгруппированы по типу объектов базы данных, на которые они воздействуют. Например, ALTER DATABASE, CREATE DATABASE и DROP DATABASE находится в разделе *DATABASE*; DECLARE EXTERNAL FUNCTION и ALTER EXTERNAL FUNCTION в *EXTERNAL FUNCTION* и т.д.

CHARACTER SET

ALTER CHARACTER SET

Доступно: DSQL

Добавлено: 2.5

Описание: С помощью оператора ALTER CHARACTER SET можно изменить умолчательный (дефолтный) набор символов. Это повлияет в будущем на использование набора символов, кроме случаев, когда явно переопределена сортировка COLLATE. Сортировка существующих доменов, столбцов и переменных PSQL при этом не будет изменена.

Синтаксис:

ALTER CHARACTER SET *charset* SET DEFAULT COLLATION *collation*

Пример:

ALTER CHARACTER SET UTF8 SET DEFAULT COLLATION UNICODE_CI_AI

Примечания:

- При использовании SET DEFAULT COLLATION на набор символов базы данных по умолчанию, то Вы установили (или изменили) параметры сортировки по умолчанию для базы данных.

- При использовании SET DEFAULT COLLATION на набор символов при подключении к БД строковые константы будет интерпретироваться в соответствии с новыми параметрами сортировки (если набор символов и/или сортировка переопределяются). В большинстве случаев это не будет иметь никакого значения, но операции сравнения могут иметь другой результат при изменении сортировки.

COLLATION

CREATE COLLATION

Доступно: DSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Добавляет сортировку для БД. Язык сортировки уже должен присутствовать в вашей системе (как правило, в библиотечном файле) и сортировки должны быть должным образом зарегистрированы в конфигурационном файле fbintl.conf в подкаталоге intl директории установки сервера Firebird. Вы также можете использовать уже присутствующие в базе данных сортировки.

Синтаксис:

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

['<specific-attributes>']
collname            ::= имя, используемое для новой сортировки
charset             ::= набор символов, присутствующий в БД
basecoll            ::= сортировка, уже присутствующая в БД
extname             ::= имя сортировки из конфигурационного файла
.conf
<specific-attributes> ::= <attribute> [; <attribute> ...]
```

<*attribute*> ::= *attrname=attrvalue*

- При отсутствии предложения FROM Firebird ищет в. конфигурационном файле в подкаталоге intl директории установки сервера сортировку с именем, указанным сразу после CREATE COLLATION. (За исключением случая явного задания сортировки предложением “FROM EXTERNAL ('*collname*')”);
- Имя сортировки в одиночных кавычках чувствительно к регистру и должно в точности совпадать с именем сортировки в конфигурационном файле.

Специфичные атрибуты: В таблице 6.1 приведён список доступных специфичных атрибутов. Не все эти атрибуты применимы ко всем сортировкам. Если атрибут не применим к сортировке, но указан при её создании, то это не вызывает ошибки. “*1 bpc*” в таблице указывает на то, что атрибут действителен для сортировок наборов символов, использующих 1 байт на символ (так называемый узкий набор символов), а “*UNI*” - для юникодных сортировок.

Таблица 6.1. Специфичные атрибуты сортировок

Имя	Значение	Валидность	Комментарий
DISABLE-COMPRESSIONS	0, 1	1 bpc	Отключает сжатия (иначе сокращения). Сжатия заставляют определенные символьные последовательности быть сортированными как атомарные модули, например, испанские c+h как единственный символ ch.
DISABLE-EXPANSIONS	0, 1	1 bpc	Отключение расширений. Расширения позволяют определенные символы (например, лигатуры или гласные умляуты) рассматривать как последовательности символов и соответственно сортировать
ICU-VERSION	default или *M.m*	UNI	Задаёт для использования версию библиотеки ICU. Допустимые значения определены в соответствующих элементах

			<intl_module> в файле intl/fbintl.conf. Формат: либо строка "default" или основной + дополнительный номер версии, как "3.0" (оба без кавычек)
LOCALE	*xx_YY*	UNI	Задает параметры сортировки языкового стандарта. Требуется полная версия библиотеки ICU. Формат строки: "du_NL" (без кавычек)
MULTI-LEVEL	0, 1	1 bpc	Использование нескольких уровней сортировки
NUMERIC-SORT	0, 1	UNI	Обрабатывает непрерывные группы десятичных цифр в строке как атомарные модули и сортирует их в цифровой форме. (известна как *естественная сортировка*)
SPECIALS-FIRST	0, 1	1 bpc	Сортирует специальные символы (пробелы и т.д.) до буквенно-цифровых символов

Примечание: Атрибут NUMERIC-SORT добавлен в Firebird 2.5.

Пример:

Простейшая форма: использует имя, найденное в файле fbintl.conf (регистро-чувствительно):

```
CREATE COLLATION ISO8859_1_UNICODE FOR ISO8859_1
```

Использование специального (заданного пользователем) названия. Обратите внимание, что «external» имя должно точно соответствовать имени в файле fbintl.conf:

```
CREATE COLLATION LAT_UNI
    FOR ISO8859_1
    FROM EXTERNAL ('ISO8859_1_UNICODE')
```

Сортировка, уже присутствующая в БД:

```
CREATE COLLATION ES_ES_NOPAD_CI
    FOR ISO8859_1
    FROM ES_ES
    NO PAD
    CASE INSENSITIVE
```

Со специфическими атрибутами (регистро-чувствительно!):

```
CREATE COLLATION ES_ES_CI_COMPR
    FOR ISO8859_1
    FROM ES_ES
    CASE INSENSITIVE
    'DISABLE-COMPRESSIONS=0'
```

Совет

Если Вы хотите добавить в базу данных новый набор символов с его умолчательной сортировкой, то зарегистрируйте и выполните хранимую процедуру *sp_register_character_set(name, max_bytes_per_character)* из подкаталога *misc/intl.sql* установки Firebird. Напоминание: Для нормальной работы с набором символов он должен присутствовать в Вашей операционной системе и зарегистрирован в файле fbintl.conf поддиректории intl.

DROP COLLATION

Доступно: DSQL

Добавлено: 2.1

Описание: Удаляет сортировку из БД. Удалить сортировку может только добавлявший её пользователь.

Синтаксис:

DROP COLLATION *name*

Совет

Если Вы хотите удалить в базе данных набор символов со всеми его сортировками, то зарегистрируйте и выполните хранимую процедуру *sp_unregister_character_set(name)* из подкаталога *misc/intl.sql* установки Firebird.

COMMENT

Доступно: DSQL

Добавлено: 2.0

Описание: Позволяет добавлять комментарии для метаданных. Комментарии при этом сохраняются в виде текстового блоба в поля RDB$DESCRIPTION системных таблиц (из этих полей клиентское приложение может просмотреть комментарии).

Синтаксис:

COMMENT ON <*object*> IS {'*текст комментария*' | 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

Замечание

Если Вы вводите пустой комментарий (''), то он будет сохранён в базе данных как NULL.

Пример:

```
COMMENT ON DATABASE IS 'Это тестовая (''my.fdb'') БД';

COMMENT ON TABLE METALS IS 'Справочник металлов';

COMMENT ON COLUMN METALS.ISALLOY is '0 = чистый металл,
1 = сплав';

COMMENT ON INDEX IX_SALES IS 'Сделайте меня неактивным
при массовой вставке :-)';
```

DATABASE

CREATE DATABASE

Доступно: DSQL, ESQL

Синтаксис (неполный):

```
CREATE {DATABASE | SCHEMA}
    ...
    [PAGE_SIZE [=] size]
    ...
    [DEFAULT CHARACTER SET charset [COLLATION collation]]
    ...
    [DIFFERENCE FILE 'filepath']

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

- При вводе размера страницы БД меньшего, чем 4096, он будет автоматически изменён на 4096. Другие числа (не равные 4096, 8192 или 16384) будут изменены на ближайшее меньшее из поддерживаемых значений.

Поддерживается размер страницы 16 кБ, а 1 кБ и 2 кБ теперь не поддерживаются.

Изменено: 1.0, 2.1

Описание: Начиная с Firebird 1.0 максимальный размер страницы увеличен с 8192 до 16384 байтов. В Firebird 2.1 и выше не поддерживаются размеры страниц 1024 и 2048 из за их не эффективности. Firebird не создаёт БД с размерами страниц 1024 и 2048, но без всяких проблем работает с уже существующими (старыми) БД с такими размерами страниц.

Сортировка по умолчанию для БД

Добавлено: 2.5

Описание: Начиная с Firebird 2.5 Вы можете задать сортировку для набора символов по умолчанию (см. пример ниже). В этом случае сортировка станет умолчательной для набора символов по умолчанию (т.е. для всей БД за исключением случаев использования других наборов символов).

Пример:

```
CREATE DATABASE "my.fdb"
    DEFAULT CHARACTER WIN1251
    COLLATION WIN1251_UA
```

Обратите внимание: Здесь используется ключевое слово COLLATION, а не обычный COLLATE.

Параметр DIFFERENCE FILE

Добавлено: 2.0

Описание: Параметр DIFFERENCE FILE добавлен в Firebird 2.0, но в настоящее время не документирован. Полное описание данного параметра см. ниже (ALTER DATABASE : ADD DIFFERENCE FILE).

ALTER DATABASE

Доступно: DSQL, ESQL

Описание: Изменяет структуру файлов базы данных или переключает её в "безопасное для копирование" состояние.

Синтаксис:

```
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]]
```

Операторы DIFFERENCE FILE и BACKUP добавлены в Firebird 2.0 и недоступны в ESQL.

BEGIN BACKUP

Доступно: DSQL

Добавлено: 2.0

Описание: «Замораживает» основной файл базы данных (*От переводчика: переводит его в режим «read only»*), что позволяет безопасно делать резервную копию БД средствами файловой системы, даже если пользователи подключены и выполняют операции с данными. При этом все изменения, вносимые пользователями в базу данных, будут записаны в отдельный файл, так называемый *дельта файл* (*delta file*). Обратите внимание: оператор BEGIN BACKUP, несмотря на синтаксис его использования (см. пример), не начинает резервное копирование БД, а лишь создаёт для него условия.

Пример:

```
ALTER DATABASE BEGIN BACKUP
```

END BACKUP

Доступно: DSQL

Добавлено: 2.0

Описание: Объединяет файл дельты с основным файлом базы данных и восстанавливает нормальное состояние работы, таким образом заканчивая создание безопасной резервной копии БД средствами файловой системы. (При этом безопасное резервное копирование с помощью GBAK остаётся доступным).

Пример:

```
ALTER DATABASE END BACKUP
```

Совет

Вместо использования операторов BEGIN и END BACKUP возможно использование утилиты nbackup Firebird: она также может «замораживать» и «размораживать» файл БД, а также создавать полный и инкрементные резервные копии БД. Руководство по использованию утилиты nbackup доступно по адресу Firebird Documentation Index.

ADD DIFFERENCE FILE

Доступно: DSQL

Добавлено: 2.0

Описание: Задаёт путь и имя дельта файла, в который будут записываться изменения, внесённые в БД после перевода её в режим «безопасного копирования» (“copy-safe”) путём выполнения команды ALTER DATABASE BEGIN BACKUP.

Пример:

```
ALTER DATABASE
    ADD DIFFERENCE FILE '/usr/opt/firebird/db/my.delta'
```

Примечания:

- Этот оператор в действительности не добавляет файла. Он просто переопределяет умолчательные имя и путь файла дельты, который будет создан при переводе БД в режим «безопасного копирования»;
- При задании относительного пути или только имени файла дельты он будет создаваться в текущем каталоге сервера. Для операционных систем Windows это системный каталог;
- Для изменения существующих установок сначала надо удалить созданные данные для файла дельты (см. ниже описание оператора DROP DIFFERENCE FILE), а затем задать новое описание файла дельты;
- Если не переопределять путь и имя файла дельты, то он будет иметь тот же путь и имя, что и БД, но с расширением .delta.

DROP DIFFERENCE FILE

Доступно: DSQL

Добавлено: 2.0

Описание: Удаляет описание (путь и имя) файла дельты, заданное ранее командой ALTER DATABASE ADD DIFFERENCE FILE. На самом деле при выполнении этого оператора файл не удаляется. Он только удаляет путь и имя файла дельты и при последующем переводе БД в режим «безопасного копирования» будут использоваться умолчательные значения (т.е. тот же путь и имя, что и БД, но с расширением .delta).

Пример:

```
ALTER DATABASE DROP DIFFERENCE FILE
```

DOMAIN

CREATE DOMAIN

Доступно: DSQL, ESQL

Контекстная переменная по умолчанию

Изменено: IB

Описание: Любая контекстная переменная, тип которой совместим с типом данных нового домена, может использовать его умолчательное значение. Это было введено ещё в InterBase 6, но в его Language Reference упоминается только контекстная переменная USER.

Пример:

```
CREATE DOMAIN DDATE AS
    DATE
    DEFAULT CURRENT_DATE
    NOT NULL
```

ALTER DOMAIN

Доступно: DSQL, ESQL

Предупреждение

При изменении описания домена существующий код PSQL, использующий этот домен, может стать некорректным. Информация о том, как это обнаружить, находится в Приложении A (раздел Поле RDB$VALID_BLR).

Переименование домена

Добавлено: IB

Описание: Переименование домена возможно с помощью предложения TO. Введено в InterBase 6, но не описано в его Language Reference.

Пример:

```
ALTER DOMAIN DDATE TO D_DATE
```

- Предложение TO может использоваться совместно с другими, но при этом не должна быть первой.

SET DEFAULT для контекстных переменных

Изменено: IB

Описание: Любая контекстная переменная, тип которой совместим с типом данных домена, может использовать его умолчательное значение. Это было введено ещё в InterBase 6, но в его Language Reference упоминается только контекстная переменная USER.

Пример:

```
ALTER DOMAIN D_DATE
      SET DEFAULT CURRENT_DATE
```

EXCEPTION

CREATE EXCEPTION

Доступно: DSQL, ESQL

Увеличение длины сообщения

Изменено: 2.0

Описание: Начиная с Firebird 2.0 максимальная длина сообщения исключения увеличена с 78 до 1021 байтов.

Пример:

```
CREATE EXCEPTION EX_TOOMANYMANAGERS
```

'Закреплено больше трёх менеджеров на один заказ.
Сократите количество ответственных за заказ менеджеров до трёх.
Если есть необходимость в закреплении больше трёх менеджеров,
то обратитесь к руководителю отдела продаж с просьбой увеличить
это ограничение в таблице лимита менеджеров на заказ.'

Примечание

Максимальная длина сообщения исключения зависит от ODS базы данных. Следовательно для баз данных, созданных в версиях Firebird, меньших, чем 2.0, для создания или изменения сообщений исключений с длиной до 1021 байта необходимо сделать процедуру резервного копирования и затем восстановления под Firebird версии 2.0 или выше.

CREATE OR ALTER EXCEPTION

Доступно: DSQL

Добавлено: 2.0

Описание: Если исключения не существует, то оно будет создано таким же образом, как и при использовании оператора CREATE EXCEPTION (см. выше). Уже существующее исключение будет изменено, при этом существующие зависимости исключения будут сохранены.

Синтаксис: Точно такой же, как и для оператора CREATE EXCEPTION (см. выше).

RECREATE EXCEPTION

Доступно: DSQL

Добавлено: 2.0

Описание: Создаёт или пересоздаёт исключение. Если исключение с таким именем уже существует, то оператор RECREATE EXCEPTION попытается удалить его и создать новое исключение. При наличии зависимостей для существующего исключения оператор RECREATE EXCEPTION не выполнится.

Синтаксис: Точно такой же, как и для оператора CREATE EXCEPTION (см. выше).

Примечание

При использовании оператора RECREATE EXCEPTION для существующих исключений с зависимостями вы не получите сообщение об ошибке до тех пор, пока не попытаетесь подтвердить транзакцию.

EXTERNAL FUNCTION

DECLARE EXTERNAL FUNCTION

Доступно: DSQL, ESQL

Описание: Регистрирует внешнюю функцию (UDF) в базе данных (т.е. делает её доступной для использования).

Синтаксис:

```
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)
```

Ограничения

- Вызов UDF с помощью метода BY DESCRIPTOR не поддерживается в ESQL.

Вы можете сами свободно выбирать название *localname*: это имя, под которым к функции можно будет обращаться в базе данных. Вы также можете изменять длину аргумента CSTRING (Подробнее о CSTRING см. в разделе Замечания к параметрам типа CSTRING в Приложении А).

Передача параметров BY DESCRIPTOR

Доступно: DSQL

Добавлено: 1.0

Описание: Firebird представляет возможность передавать параметры через дескриптор (BY DESCRIPTOR): этот механизм облегчает обработку значений NULL. Отметим, что это объявление работает только в том случае, если внешняя функция поддерживает его. Простое добавление "BY DESCRIPTOR" к существующей декларации не заставит его работать - наоборот! Всегда используйте объявление блока, обеспеченное функционалом внешней функции.

RETURNS PARAMETER *n*

Доступно: DSQL, ESQL

Добавлено: IB 6

Описание: При возврате из внешней функции BLOB должен быть объявлен дополнительный входной параметр и предложение “RETURNS PARAMETER *n*”, где n позиция возвращаемого параметра. Введено в InterBase 6, но не описано в его Language Reference (хотя документировано в Developer's Guide).

ALTER EXTERNAL FUNCTION

Доступно: DSQL

Добавлено: 2.0

Описание: Изменяет имя внешней функции и/или точку входа. Существующие зависимости сохраняются.

Синтаксис:

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

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

Пример:

```
ALTER EXTERNAL FUNCTION PHI MODULE_NAME 'NewUdfLib'
```

FILTER

DECLARE FILTER

Доступно: DSQL, ESQL

Изменено: 2.0

Описание: Делает доступными в базе данных фильтры BLOB.

Синтаксис:

```
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
```

- Начиная с Firebird 2 в базе данных не может быть двух и более фильтров BLOB с одинаковыми комбинациями входных и выходных типов. Объявление фильтра с уже существующими комбинациями входных и выходных типов BLOB приведёт к ошибке. Восстановление базы данных, созданной в версиях Firebird ниже 2.0 и имеющей такие «дубликаты» фильтров BLOB, невозможно;
- В Firebird 2 была добавлена возможность указать типы BLOB с их мнемоникой (текстовое название типа BLOB) вместо числового представления. Также в Firebird 2 *binary* мнемоника для подтипа 0. Предопределенные мнемоники чувствительны к регистру.

Пример:

```
DECLARE FILTER FUNNEL
    INPUT_TYPE blr OUTPUT_TYPE text
    ENTRY_POINT 'blr2asc' MODULE_NAME 'myfilterlib'
```

Пользовательские мнемоники: Если вы хотите определить мнемоники для собственных подтипов BLOB, Вы можете добавить их в системную таблицу

RDB$TYPES, как показано ниже. После подтверждения транзакции мнемоники могут быть использованы для декларации при создании новых фильтров.

```
INSERT INTO RDB$TYPES
    (RDB$FIELD_NAME, RDB$TYPE, RDB$TYPE_NAME)
VALUES
    ('RDB$FIELD_SUB_TYPE', -33, 'MIDI');
```

Значение поля rdb$field_name всегда должно быть 'RDB$FIELD_SUB_TYPE'. Если Вы определяете мнемоники в верхнем регистре, то можете использовать их без учета регистра и без кавычек при объявлении фильтра.

INDEX

CREATE INDEX

Доступно: DSQL, ESQL

Описание: Создаёт индекс таблицы для ускорения поиска, сортировки и/или группирования.

Синтаксис:

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

<*col*> ::= не для столбцов таблицы типов ARRAY, BLOB или COMPUTED BY

Уникальный индекс (UNIQUE INDEX) теперь поддерживает значение NULL

Изменено: 1.5

Описание: В соответствии со стандартом SQL-99 в столбцах, по которым построен уникальный индекс, допускается хранить значение NULL (в т.ч. и в нескольких строках таблицы). Более подробно этот вопрос изложен в разделе *CREATE NABLE :: Уникальное ограничение (UNIQUE constraints) теперь поддерживает значение NULL.* Правила в отношении NULL для уникальных

индексов точно такие же, как и для уникальных ключей.

Вычисляемый индекс

Добавлено: 2.0

Описание: При создании индекса вместо одного или нескольких столбцов Вы также можете указать одно выражение, используя предложение COMPUTED BY. Вычисляемый индексы используются в запросах, в которых условие в предложениях WHERE, ORDER BY или GROUP BY в точности совпадает с выражением в определении индекса. Многосегментный вычисляемый индекс не поддерживается, но само выражение в вычисляемом индексе может использовать несколько столбцов таблицы.

Примеры:

```
CREATE INDEX NAME_UPPPER ON PERSONS
        COMPUTED BY (UPPER(NAME));
COMMIT;

-- Эти запросы используют индекс NAME_UPPPER:

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;

-- Эти запросы используют индекс IX_EVENTS_YT:
```

```
SELECT *
FROM MYEVENTS
ORDER BY EXTRACT(YEAR FROM STARTDATE) || TOWN DESC;
```

Увеличена максимальная длина ключа индекса

Изменено: 2.0

Описание: Максимальная используемая длина ключа индекса, ранее установленная в 252 байта, теперь равна 1/4 размера страницы, т.е. ль 256 до 4096 байтов. Максимальная длина индексируемой строки на 9 байтов меньше, чем максимальная длина ключа. В таблице 6.2 приведены данные для максимальной длины индексируемой строки (в символах) в зависимости от размера страницы и набора символов.

Таблица 6.2 Максимальная длина индексируемого (VAR)CHAR

Размер страницы	**Максимальная длина индексируемой строки для набора символов**			
	1 байт/символ	2 байта/символ	3 байта/символ	4 байта/символ
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

Увеличено максимальное число индексов в таблице

Изменено: 1.0.3, 1.5, 2.0

Описание: Максимальное число индексов в таблице: Firebird 1.0.3 — 65; Firebird 1.5 — 257; вновь увеличено в Firebird 2.0.

Начиная с Firebird 2.0 это уже не «жёстко» заданное число: максимальное число индексов в таблице зависит теперь от размера страницы и числа столбцов в индексе (таблица 6.3).

Таблица 6.3 Макс. число индексов в таблице, начиная с Firebird 2.0

Размер страницы	Число индексов в зависимости от кол-ва столбцов в индексе		
	1	**2**	**3**
1024	50	35	27
2048	101	72	56
4096	203	145	113
8192	408	291	227
16384	818	584	454

Помните, что при нормальных обстоятельствах даже 50 индексов в таблице слишком много и резко снижают скорость восстановления из резервной копии и выполнения массовых операций вставки записей. Ограничение по максимальному числу индексов было снято для работы с хранилищами данных приложений и т.п., , которые при выполнении массовых операций с таблицами временно отключают (деактивируют) индексы.

Полная таблица для максимального числа индексов в таблице, включающая в себя версии Firebird 1.0-1.5, приведена в Приложении А

PROCEDURE

Хранимая процедура (ХП) представляет из себя программный модуль, который может быть выполнен на клиенте, вызван из другой ХП, выполнимого блока (executable block) или триггера. Хранимые процедуры, выполнимые блоки и триггера пишутся на процедурном языке SQL (PSQL). Большинство операторов SQL доступно и в PSQL, иногда с ограничениями или расширениями. Заметными исключениями являются DDL и операторы управления транзакциями.

Хранимые процедуры могут принимать и возвращать множество параметров.

CREATE PROCEDURE

Доступно: DSQL, ESQL

Описание: Создаёт хранимую процедуру.

Синтаксис:

```
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 | TYPE OF COLUMN rel.col
<declarations> ::= См. описание точного синтаксиса в раз деле PSQL::DECLARE
```

/* Если Isql_*datatype* имеет тип строки, то он может включать в себя набор символов */

TYPE OF COLUMN в объявлениях параметров и переменных

Добавлено: 2.5

Описание: По аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Используется только тип данных, а в случае строковых типов ещё и набор символов и параметры сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Пример:

/* Исходя из автоподтверждения DDL и кодировки подключения UTF8 */

```
CREATE DOMAIN DPHRASE AS
    VARCHAR(200) CHARACTER SET UTF8
    COLLATE UNICODE_CI_AI;

CREATE TABLE PHRASES (PHRASE DPHRASE);
```

```
SET TERM ^;
CREATE PROCEDURE EQUALPHRASES (
        A TYPE OF COLUMN PHRASES.PHRASE,
        B TYPE OF COLUMN PHRASES.PHRASE)
RETURNS (RES VARCHAR(30))
AS
BEGIN
        IF (:A = :B) THEN
                RES = 'Yes';
        ELSE
                RES = 'No';
        SUSPEND;
END^
SET TERM ;^

SELECT RES FROM EQUALPHRASES('Appel', 'appel');

-- результат 'Yes'
```

Предупреждения

- Для текстовых типов в TYPE OF COLUMN используются набор символов и порядок сортировки — так же, как и при использовании [TYPE OF] *<domain>*. Однако, из-за ошибки, сортировки не всегда принимаются во внимание при выполнении операции сравнения (например, на равенство). В случаях, когда сортировка имеет важное значение, необходимо тщательно тестировать свой код перед использованием! Эта ошибка исправлена в Firebird 3;
- Если тип столбца позднее изменяется, PSQL код с использованием этого столбца может вызывать ошибки. Информация о том, как это обнаружить, находится в Приложении A (раздел Поле RDB$VALID_BLR).

Поддержка доменов при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование доменов вместо типов данных SQL при объявлении входных и выходных параметров и локальных переменных. Если перед названием домена дополнительно используется предложение "TYPE OF", то используется только тип данных домена — не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию. Если домен текстового типа, то всегда используется его набор символов и порядок сортировки.

Пример:

```
CREATE DOMAIN BOOL3
        SMALLINT
        CHECK (VALUE IS NULL OR VALUE IN (0,1));

CREATE DOMAIN BIGPOSNUM
        BIGINT
        CHECK (VALUE >= 0);

/* Определение кратности A на B: */
SET TERM ^;
CREATE PROCEDURE ISMULTIPLE (
        A BIGPOSNUM, B BIGPOSNUM)
RETURNS (RES BOOL3)
AS
-- Отношение является BIGINT
  DECLARE RATIO TYPE OF BIGPOSNUM;

– Такой же остаток
DECLARE REMAINDER TYPE OF BIGPOSNUM;
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; -- Целочисленное деление!
                        REMAINDER = :A - :B*:RATIO;
                        IF (:REMAINDER = 0) THEN
                                RES = 1;
                        ELSE
                                RES = 0;
                END
END^
SET TERM;^
```

Предупреждение

Если тип домена позднее изменяется, PSQL код с использованием этого домена может вызывать ошибки. Информация о том, как это обнаружить, находится в Приложении A (раздел Поле RDB$VALID_BLR).

Использование сортировок при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование предложения COLLATE при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
CREATE PROCEDURE SPANISHTODUTCH(
    ES_1 VARCHAR(20)
        CHARACTER SET WIN1251 COLLATE WIN1251_UA,
    ES_2 MY_CHAR_DOMAIN COLLATE WIN1251_UA)
RETURNS(
    NL_1 VARCHAR(20)
        CHARACTER SET WIN1251 COLLATE WIN1251_UA,
    NL_2 MY_CHAR_DOMAIN COLLATE DU_NL)
AS
    DECLARE S_TEMP VARCHAR(100)
        CHARACTER SET UTF8 COLLATE UNICODE;
BEGIN
    ...
    ...
END
```

NOT NULL при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование NOT NULL ограничения при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
CREATE PROCEDURE REGISTERORDER(
        ORDER_NO INTEGER NOT NULL,
        DESCRIPTION VARCHAR(200) NOT NULL)
RETURNS(
        TICKET_NO INTEGER NOT NULL)
AS
 DECLARE TEMP INTEGER NOT NULL;
BEGIN
        ...
        ...
END
```

Значения параметров по умолчанию

Добавлено: 2.0

Описание: Теперь есть возможность использовать для параметров хранимых процедур значения по умолчанию, позволяя при её вызове пропустить один или несколько элементов (или даже все) из конца списка аргументов.

Описание: Параметры хранимых процедур могут использовать значения по умолчанию, если при её вызове пропустить один или несколько элементов (или даже все) из конца списка аргументов.

Синтаксис:

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

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

Важно: Если для параметра задано значение по умолчанию, то Вы должны задать умолчательные значения для всех параметров, следующих за ним.

Блок BEGIN...END может быть пустым

Изменено: 1.5

Описание: Начиная с Firebird 1.5 блок BEGIN...END может быть пустым, т.о. создаётся своеобразная «заглушка», позволяющая избежать написания фиктивных операторов.

Пример:

```
CREATE PROCEDURE GRAB_INTS (
        A INTEGER, B INTEGER)
AS
BEGIN
END
```

ALTER PROCEDURE

Доступно: DSQL, ESQL

Значения параметров по умолчанию

Добавлено: 2.0

Описание: Параметры хранимых процедур могут использовать значения по умолчанию, если при её вызове пропустить один или несколько элементов (или даже все) из конца списка аргументов. Детальная информация и синтаксис приведены выше (CREATE PROCEDURE).

Пример:

```
ALTER PROCEDURE TESTPROC (
        A INTEGER, B INTEGER DEFAULT 1007, S VARCHAR(12) = '-')
        ...
```

Classic Server: Измененная процедура сразу же видима для всех клиентов

Изменено: 2.5

Описание: До Firebird версии 2.5 при изменении процедуры в Classic Server видели и выполняли старую версию процедуры до момента завершения соединения. Эта проблема устранена в Firebird 2.5. Теперь все клиенты сразу же видят новую версию хранимой процедуры, как только изменения будут подтверждены.

Использование сортировок при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование предложения COLLATE при объявлении входных и выходных параметров и локальных переменных. Детальная информация и синтаксис приведены выше (CREATE PROCEDURE).

Поддержка доменов при объявлении переменных и параметров

Описание: Начиная с Firebird 2.1 поддерживается использование доменов вместо типов данных SQL при объявлении входных и выходных параметров и локальных переменных. Детальная информация и синтаксис приведены выше (CREATE PROCEDURE).

NOT NULL при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование NOT NULL ограничения при объявлении входных и выходных параметров и локальных переменных. Детальная информация и синтаксис приведены выше (CREATE PROCEDURE).

Ограничение на изменение хранимых процедур

Изменено: 2.0, 2.0.1

Описание: Только в Firebird 2.0 существует ограничение, запрещающее удалять, изменять или пересоздавать триггера или хранимые процедуры, если они использовались со времени открытия базы данных. Это ограничение было снято в Firebird 2.0.1. Однако выполнение этих операций на «живой» (имеющей подключения) базе данных потенциально опасно и они должны быть сделаны с предельной осторожностью.

TYPE OF COLUMN в объявлениях параметров и переменных

Добавлено: 2.5

Описание: по аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Детальная информация и синтаксис приведены выше (CREATE PROCEDURE).

CREATE OR ALTER PROCEDURE

Доступно: DSQL

Добавлено: 1.5

Описание: Если хранимая процедура не существует, то она будет создана с использованием предложения CREATE PROCEDURE . Если она уже существует, то она будет изменена и перекомпилирована, при этом сохраняются существующие привилегии и зависимости.

Синтаксис: Точно такой же, как и у предложения CREATE PROCEDURE .

DROP PROCEDURE

Доступно: DSQL, ESQL

Ограничение на удаление используемых хранимых процедур

Изменено: 2.0, 2.0.1

Описание: Только в Firebird 2.0 существует ограничение, запрещающее удалять, изменять или пересоздавать триггера или хранимые процедуры, если они использовались со времени открытия базы данных. Это ограничение было снято в Firebird 2.0.1. Однако выполнение этих операций на «живой» (имеющей подключения) базе данных потенциально опасно и они должны быть сделаны с предельной осторожностью.

RECREATE PROCEDURE

Доступно: DSQL

Добавлено: 1.0

Описание: Создаёт или пересоздаёт хранимую процедуру. Если процедура с таким именем уже существует, то RECREATE POCEDURE попытается удалить её и создать новую процедуру. RECREATE POCEDURE невозможно, если существующая процедура используется.

Синтаксис: Точно такой же, как и у предложения CREATE PROCEDURE .

SEQUENCE или GENERATOR

CREATE SEQUENCE

Доступно: DSQL

Добавлено: 2.0

Описание: Создаёт новую последовательность или генератор. SEQUENCE представляет собой SQL-совместимый аналог известного в InterBase и Firebird генератора. Оператор CREATE SEQUENCE полностью эквивалентен оператору CREATE GENERATOR и является рекомендуемым синтаксисом начиная с Firebird 2.0.

Синтаксис:

CREATE SEQUENCE *sequence-name*

Пример:

```
CREATE SEQUENCE SEQ_TEST
```

Поскольку по своему функционалу последовательности и генераторы аналогичны *(Примечание переводчика: в силу этого в дальнейшем в тексте будет использоваться термин генератор)*, Вы можете свободно использовать генераторы и последовательности даже при работе на одном объекте метаданных. Однако это не рекомендуется.

Независимо от диалекта базы данных последовательности (или генераторы) всегда хранятся как 64-битные целые значения. Однако отметим, что:

- Если *клиент* использует 1 диалект, то сервер передает ему значения генератора, усеченные до 32-битного значения;
- Если значения генератора передаются в 32-разрядное поле или переменную, то до тех пор, пока текущее значение генератора не вышло за границы для 32-битного числа, ошибок не будет. В момент выхода значения генератора за этот диапазон база данных 3-го диалекта выдаст сообщение об ошибке, а база данных 1-го диалекта будет молча обрезать значения (что также может привести к ошибке — например, если поле, заполняемое генератором, является первичным или уникальным).

См. также ALTER SEQUENCE , NEXT VALUE FOR , DROP SEQUENCE

CREATE GENERATOR

Доступно: DSQL, ESQL

Наилучшая альтернатива: CREATE SEQUENCE

Предпочтительно использовать CREATE SEQUENCE

Изменено: 2.0

Описание: Начиная с Firebird 2.0 предпочтительным для создания генераторов является SQL-совместимое предложение CREATE SEQUENCE.

Максимальное количество генераторов существенно увеличено

Изменено: 1.0

Описание: InterBase резервировал для генераторов одну страницу в базе данных, ограничивая их количество 123 (для БД с размером страницы 1 кБ) до 1019 (размер страницы БД 8 кБ). В Firebird снято это ограничение: Вы можете создать в базе данных более 32 000 генераторов.

ALTER SEQUENCE

Доступно: DSQL

Добавлено: 2.0

Описание: Устанавливает значение последовательности или генератора в заданное значение. Последовательность является SQL-совместимым термином того, что в InterBase и Firebird всегда называли генератором. “ALTER SEQUENCE ... RESTART WITH ...” полностью эквивалентно предложению “SET GENERATOR ... TO ...” и является рекомендуемым синтаксисом начиная с Firebird 2.0.

Синтаксис:

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

<*newval*> ::= 64-битное целое значение.

Пример:

```
ALTER SEQUENCE SEQ_TEST RESTART WITH 0
```

Предупреждение

Неосторожное использование ALTER SEQUENCE (изменение значения генератора) может привести к потере логической целостности данных в базе данных или к ошибкам.

Примечание переводчика

Ошибки из-за изменения начального значения последовательности (генератора) могут быть, например, такими:

- *если генератор используется в качестве первичного или уникального ключа, то при вставке новой записи со значением, уже существующим в БД, Вы получите ошибку;*
- *если у Вас несколько БД с разнесённым шагом первичных ключей (например, шаг генератора 100, в 1-й БД генератор GEN_1 стартует с 1, во второй с 2 и т.д.) то при репликации Вы также получите ошибку при вставке записи из другой БД в случае совпадения значений первичного ключа. Это приведёт к не получению («потере») информации из, например, 2-й БД во всех остальных. Гораздо хуже случай обновления записей — при совпадении значения первичного ключа Вы перезапишите информацию, т.е. безвозвратно потеряете старые данные.*

См. также CREATE SEQUENCE

SET GENERATOR

Доступно: DSQL, ESQL

Наилучшая альтернатива: ALTER SEQUENCE

Описание: Устанавливает значение последовательности или генератора в заданное значение. Начиная с Firebird 2.0 предпочтительнее использовать SQL-совместимый оператор ALTER SEQUENCE.

Синтаксис:

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

<*new-value*> *::= 64-битное целое значение.*

Предупреждение

После того, как генератор или последовательность созданы, Вы не должны менять их значения (за исключением получения следующего значения с помощью GEN_ID или NEXT VALUE FOR), за исключением случаев, когда Вы точно уверены в своих действиях (см. 72).

DROP SEQUENCE

Доступно: DSQL

Добавлено: 2.0

Описание: Удаляет последовательность или генератор из базы данных. После цикла резервного копирования-восстановления в БД освобождается очень маленький размер памяти. Последовательность является SQL-совместимым термином того, что в InterBase и Firebird всегда называли генератором. Оператор DROP SEQUENCE полностью эквивалентен оператору DROP GENERATOR и начиная с Firebird 2.0 является рекомендуемым синтаксисом.

Синтаксис:

DROP SEQUENCE *sequence-name*

Пример:

DROP SEQUENCE SEQ_TEST

См. также CREATE SEQUENCE

DROP GENERATOR

Доступно: DSQL

Добавлено: 1.0

Наилучшая альтернатива: DROP SEQUENCE

Описание: Удаляет последовательность или генератор из базы данных. После цикла резервного копирования-восстановления в БД освобождается очень маленький размер памяти.

Синтаксис:

DROP GENERATOR *generator-name*

Начиная с Firebird 2.0 рекомендуемым синтаксисом является SQL-совместимый оператор DROP SEQUENCE .

TABLE

CREATE TABLE

Доступно: DSQL, ESQL

Глобальные временные таблицы - Global Temporary Tables (GTT)

Добавлено: 2.1

Описание: Глобальные временные таблицы (в дальнейшем сокращённо GTT) так же, как и обычные таблицы, являются постоянными метаданными, но данные в них ограничено по времени существования транзакцией (значение по умолчанию) или соединением с БД. Каждая транзакция или соединение имеет свой собственный экземпляр GTT с данными, изолированный от всех остальных. Экземпляры создаются только при условии обращения к GTT, и данные в ней удаляются при подтверждении транзакции или отключении от БД. Для модификации или удаления GTT используются операторы ALTER TABLE и DROP TABLE.

Синтаксис:

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

- ON COMMIT DELETE ROWS создаёт а GTT транзакционного уровня (по умолчанию), ON COMMIT PRESERVE ROWS GTT уровня соединения с БД;

- Предложение EXTERNAL [FILE] нельзя использовать для глобальной временной таблицы.

Ограничения: GTT-таблицы могут иметь такие же атрибуты, как и обычные таблицы (ключи, ссылки, индексы, триггеры ...), но есть несколько ограничений:

- GTT и обычные таблицы не могут ссылаться друг на друга;
- GTT уровня соединения (“PRESERVE ROWS”) GTT не могут ссылаться на GTT транзакционного уровня (“DELETE ROWS”);
- Доменные ограничения не могут использоваться в GTT;
- Уничтожения экземпляра GTT в конце своего жизненного цикла не вызывает срабатывания триггеров до/после удаления.

Пример:

```
CREATE GLOBAL TEMPORARY TABLE MYCONNGTT (
    ID INTEGER NOT NULL PRIMARY KEY,
    TXT VARCHAR(32),
    TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
ON COMMIT PRESERVE ROWS;

COMMIT;

CREATE GLOBAL TEMPORARY TABLE MYTXGTT (
    ID INTEGER NOT NULL PRIMARY KEY,
    PARENT_ID INTEGER NOT NULL REFERENCES
MYCONNGTT(ID),
    TXT VARCHAR(32),
    TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

COMMIT;
```

Совет

В существующей базе данных не всегда отличить обычную таблицу от GTT, или GTT на уровне транзакций от GTT уровня подключения. Для получения типа таблицы можно использовать следующий запрос:

```
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'
```

Или для получения полного списка Ваших таблиц:

```
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
ORDER BY 2, 1
```

GENERATED ALWAYS AS

Добавлено: 2.1

Описание: Вместо предложения COMPUTED [BY] Вы можете также использовать SQL-2003-совместимый эквивалент GENERATED ALWAYS AS для вычисляемых файлов.

Синтаксис:

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

Пример:

```
CREATE TABLE PERSONS (
    ID INTEGER PRIMARY KEY,
    FIRSTNAME VARCHAR(24) NOT NULL,
    MIDDLENAME VARCHAR(24),
    LASTNAME VARCHAR(24) NOT NULL,
    FULLNAME VARCHAR(74) GENERATED ALWAYS AS
        (LASTNAME ||
         COALESCE(' ' || FIRSTNAME, ' ') ||
         COALESCE(' ' || MIDDLENAME, ''),
    STREET VARCHAR(32),
    ...
    ...
);
```

Примечание: поля с GENERATED ALWAYS AS в настоящее время не поддерживаются для создания индекса.

CHECK допускает результат NULL

Изменено: 2.0

Описание: Если ограничение CHECK возвращает значение NULL, то Firebird вплоть до версии 2.0 отклоняет введённые данные. Следуя стандарту SQL начиная с Firebird 2.0 и выше NULL в ограничении CHECK принимается и проверка ограничения срабатывает, только если её результат является ложным (*false*).

Пример:

Такие проверки, как эти:

```
CHECK (VALUE > 10000)

CHECK (TOWN LIKE 'Amst%')

CHECK (UPPER(VALUE) IN ( 'A', 'B', 'X' ))

CHECK (MINIMUM <= MAXIMUM)
```

все они *не пройдут* проверку в версиях Firebird ниже 2..0, если значением для проверки является NULL. Начиная с Firebird 2.0 и выше эти проверки будут *успешными*.

Предупреждение

Новое поведение CHECK в существующих базах данных можно использовать только после миграции на версию Firebird 2.0 или выше. Внимательно изучите структуру Ваших таблиц и предикатов “and XXX is not null” в ограничениях CHECK, если они по-прежнему не допускают результат NULL в ограничениях.

Контекстные переменные как значение по умолчанию

Изменено: IB

Описание: Любая контекстная переменная может использоваться в качестве значения по умолчанию для столбца таблицы при условии совместимости по типу данных. Это было введено ещё в InterBase 6, но в его Language Reference упоминается только контекстная переменная USER.

Пример:

```
CREATE TABLE MYDATA (
    ID INTEGER NOT NULL PRIMARY KEY,
```

```
        RECORD_CREATED TIMESTAMP
            DEFAULT CURRENT_TIMESTAMP,
        ...
);
```

Внешний ключ без ссылки на столбец с первичным ключом

Изменено: IB

Описание: При создании внешнего ключа (FOREIGN KEY) без указания целевого столбца он будет ссылаться на первичный ключ целевой таблицы. Такое поведение было введено в IB 6.0, но в его Language Reference ошибочно утверждается, что сервер ищет в целевой таблице столбец с таким же именем, как и у ссылающегося столбца.

Пример:

```
CREATE TABLE EIK (
    A INTEGER NOT NULL PRIMARY KEY,
    B INTEGER NOT NULL UNIQUE
);

CREATE TABLE BEUK (
    B INTEGER REFERENCES EIK
    ...
);

-- BEUK.B ссылается на  EIK.A, а не на EIK.B !
```

Создание внешнего ключа не требует монопольного доступа

Изменено: 2.0

Описание: Начиная с Firebird 2.0 и выше для создания ограничений внешнего ключа не требуется монопольный доступ к базе данных.

Для уникальных ограничений разрешено значение NULL

Изменено: 1.5

Описание: В соответствии со стандартом SQL-99 в столбцах, по которым построено уникальное (UNIQUE) ограничение, допускается хранить значение NULL (в т.ч. и в нескольких строках таблицы). Таким образом, можно определить

уникальный ключ на столбец, который не имеет ограничения NOT NULL.

Для уникальных ключей, содержащих несколько столбцов, логика немного сложнее:

- Во всех столбцах, входящих в уникальный ключ, нет ограничения NOT NULL;
- Разрешено хранение значения NULL в столбцах, входящих в уникальный ключ, в нескольких строках;
- Разрешены строки, имеющие в одном из столбцов уникального ключа значение NULL, а остальные столбцы заполнены значениями и эти значения различны хотя бы в одном из них;
- Запрещены строки, имеющие в одном из столбцов уникального ключа значение NULL, а остальные столбцы заполнены значениями и эти значения имеют совпадения хотя бы в одном из них.

Обобщая, можно сказать следующее: В принципе, все значения NULL считаются разными; Если две строки имеют столбцы уникального ключа со значениями в каких-либо столбцах, то столбец с NULL игнорируется, а определяющее значение имеют не NULL столбцы — как будто только они входят в состав уникального ключа.

Предложение USING INDEX

Доступно: DSQL

Добавлено: 1.5

Описание: Предложение USING INDEX может быть помещено в конце предложения определения первичного, уникального или внешнего ключа. Его задачи следующие:

- Задать определённое пользователем имя автоматически создаваемого индекса, связанного с создаваемым ограничением;
- Опционально определить какой это индекс - по возрастанию или по убыванию (по умолчанию по возрастанию).

Без предложения USING INDEX для индексов автоматически формируются внутренние (служебные) имена, например, для внешних ключей — RDB$FOREIGN1, RDB$FOREIGN2 и т.д., не позволяющие определить по их имени, к какой таблице относится ограничение. Начиная с Firebird 1.5 при явном указании имени ограничения индексу автоматически присваивается то же самое имя.

Примечание

Вы должны помнить, что имя индекса является уникальным, т.е. Вы должны давать для нового индекса *новое* имя.

Предложение USING INDEX можно применять для поля, таблицы и, в случае ALTER TABLE, с предложением ADD CONSTRAINT. Оно работает как с именованными, так и с не именованными ограничениями. Оно не работает с CHECK ограничениями, т.к. они не создают индексы.

Синтаксис:

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

Пример:

В первом примере создаётся ограничение - первичный ключ PK_CUST, использующий индекс с именем IX_CUSTNO:

```
CREATE TABLE CUSTOMERS (
    CUSTNO INTEGER NOT NULL
        CONSTRAINT PK_CUST PRIMARY KEY
        USING INDEX IX_CUSTNO,
    ...
```

Однако в случае не указания имени ограничения:

```
CREATE TABLE CUSTOMERS (
    CUSTNO INTEGER NOT NULL PRIMARY KEY
        USING INDEX IX_CUSTNO,
    ...
```

...Вы получите ограничение первичного ключа с именем INTEG_11 (или похожим именем - INTEG_XXX) и индекс IX_CUSTNO.

Ещё несколько примеров:

```
CREATE TABLE PEOPLE (
    ID INTEGER 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
```

Важно

При создании индекса по убыванию для первичного или уникального ключа убедитесь, что ссылающиеся на них внешние ключи также созданы с убывающими индексами.

ALTER TABLE

Доступно: DSQL, ESQL

ADD COLUMN: Контекстные переменные как значение по умолчанию

Изменено: IB

Описание: Любая контекстная переменная может использоваться в качестве значения по умолчанию для столбца таблицы при условии совместимости по типу данных. Это было введено ещё в InterBase 6, но в его Language Reference упоминается только контекстная переменная USER.

Пример:

```
ALTER TABLE MYDATA
    ADD MYDAY DATE DEFAULT CURRENT_DATE;
```

ALTER COLUMN и для генерируемых (вычисляемых) столбцов

Доступно: DSQL

Добавлено: 2.5

Описание: Firebird 2.5 поддерживает изменение генерируемых (вычисляемых) столбцов - в предыдущих версиях это было недоступно. Можно изменить только тип данных и выражение для вычисления — Вы не можете изменить обычный столбец на вычисляемый или наоборот.

Синтаксис:

```
ALTER TABLE tablename ALTER [COLUMN] gencolname
    [TYPE datatype]
    {GENERATED ALWAYS AS | COMPUTED BY}
        (expression)
```

Пример:

```
CREATE TABLE NUMS (
    A INTEGER,
    B GENERATED ALWAYS AS (3*A));
COMMIT;

ALTER TABLE NUMS
    ALTER B GENERATED ALWAYS AS (4*A + 7);
COMMIT;
```

Обратите внимание, что Вы можете использовать предложение GENERATED ALWAYS AS при изменении столбца, созданного с помощью COMPUTED BY, и наоборот.

ALTER COLUMN ... TYPE не приводит к ошибке, если столбец используется в триггере или хранимой процедуре

Изменено: 2.5

Описание: В предыдущих версиях, если на столбец таблицы была ссылка в хранимой процедуре или триггере, тип столбца не мог быть изменен, даже если это изменение не затрагивает код PSQL. Теперь такие изменения разрешены - даже если они *затрагивают* код.

Предупреждение

Это означает, что теперь вы можете вносить изменения, которые затрагивают хранимые процедуры или триггера, получая при этом только предупреждение! Для получения информации о том, как отследить недействительные PSQL модули после изменения типа столбца, пожалуйста,

прочитайте записку RDB $ VALID_BLR поле в конце документа.

ALTER COLUMN: DROP DEFAULT

Доступно: DSQL

Добавлено: 2.0

Описание: Начиная с Firebird 2.0 Вы можете удалить для столбца значение по умолчанию. Как только значение по умолчанию удалено, в столбце не будет существовать никакого значения по умолчанию или – если тип столбца ДОМЕН со значением по умолчанию – доменное значение по умолчанию перекроет это удаление.

Синтаксис:

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

Пример:

```
ALTER TABLE TREES ALTER GIRTH DROP DEFAULT
```

Если Вы используете предложение DROP DEFAULT для столбца, у которого нет значения по умолчанию, или чье значение по умолчанию основано на домене, то это приведёт к ошибке выполнения данного оператора.

ALTER COLUMN: SET DEFAULT

Доступно: DSQL

Добавлено: 2.0

Описание: Начиная с Firebird 2.0 Вы можете добавить/изменить для столбца значение по умолчанию. Если столбец уже имел умолчательное значение, то оно будет заменено новым. Значение по умолчанию для столбца всегда перекрывает доменное умолчательное значение.

Синтаксис:

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

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

Пример:

```
ALTER TABLE CUSTOMERS
    ALTER ENTEREDBY SET DEFAULT CURRENT_USER
```

Совет

Если Вы хотите отменить для столбца действие доменного значения по умолчанию, то установите умолчательное значение столбца в NULL.

ALTER COLUMN: POSITION теперь начинается с 1

Изменено: 1.0

Описание: При изменении позиции столбца сервер теперь интерпретирует новую позицию начиная с 1. Это соответствует стандарту SQL и документации InterBase, но на практике InterBase интерпретировал позицию начиная с 0.

Синтаксис:

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

<newpos> ::= целое число между 1 и количеством
столбцов
```

Пример:

```
ALTER TABLE STOCK ALTER QUANTITY POSITION 3
```

Примечание

Не путайте это с `POSITION` в CREATE/ALTER TRIGGER. Триггерные позиции были и и остаются на основе начала с 0.

CHECK допускает результат NULL

Изменено: 2.0

Описание: Если ограничение CHECK возвращает значение NULL, то Firebird вплоть до версии 2.0 отклоняет введённые данные. Следуя стандарту SQL начиная с Firebird 2.0 и выше NULL в ограничении CHECK принимается и проверка

ограничения срабатывает, только если её результат является ложным (*false*). Более подробную информацию смотрите в разделе CREATE TABLE .

Внешний ключ без ссылки на столбец с первичным ключом

Изменено: IB

Описание: При создании внешнего ключа (FOREIGN KEY) без указания целевого столбца он будет ссылаться на первичный ключ целевой таблицы. Такое поведение было введено в IB 6.0, но в его Language Reference ошибочно утверждается, что сервер ищет в целевой таблице столбец с таким же именем, как и у ссылающегося столбца.

Пример:

```
CREATE TABLE EIK (
        A INTEGER NOT NULL PRIMARY KEY,
        B INTEGER NOT NULL UNIQUE
);

CREATE TABLE BEUK (
        B INTEGER REFERENCES EIK
        ...
);

-- BEUK.B ссылается на  EIK.A, а не на EIK.B !
```

С**оздание внешнего ключа не требует монопольного доступа**

Изменено: 2.0

Описание: Начиная с Firebird 2.0 и выше для создания ограничений внешнего ключа не требуется монопольный доступ к базе данных.

GENERATED ALWAYS AS

Добавлено: 2.1

Описание: Вместо предложения COMPUTED [BY] Вы можете также использовать SQL-2003-совместимый эквивалент GENERATED ALWAYS AS для вычисляемых файлов.

Синтаксис:

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

Пример:

```
ALTER TABLE FRIENDS ADD FULLNAME VARCHAR(74)
    GENERATED ALWAYS AS
        (LASTNAME ||
         COALESCE(' ' || FIRSTNAME, ' ') ||
         COALESCE(' ' || MIDDLENAME, ''),
```

Для уникальных ограничений разрешено значение NULL

Изменено: 1.5

Описание: В соответствии со стандартом SQL-99 в столбцах, по которым построено уникальное (UNIQUE) ограничение, допускается хранить значение NULL (в т.ч. и в нескольких строках таблицы). Более подробную информацию смотрите в разделе CREATE TABLE .

Предложение USING INDEX

Доступно: DSQL

Добавлено: 1.5

Описание: Предложение USING INDEX может быть помещено в конце предложения определения первичного, уникального или внешнего ключа. Его задачи следующие:

- Задать определённое пользователем имя автоматически создаваемого индекса, связанного с создаваемым ограничением;
- Опционально определить какой это индекс - по возрастанию или по убыванию (по умолчанию по возрастанию).

Синтаксис:

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

Более подробную информацию и примеры смотрите в разделе CREATE TABLE .

RECREATE TABLE

Доступно: DSQL

Добавлено: 1.0

Описание: Создаёт или пересоздаёт таблицу. Если таблица с таким именем уже существует, то оператор RECREATE TABLE попытается удалить её (все данные в таблице будут удалены!) и затем создать новую таблицу. Оператор RECREATE TABLE не выполнится, если существующая таблица используется.

Синтаксис: Точно такой же, как и для CREATE TABLE .

TRIGGER

CREATE TRIGGER

Доступно: DSQL, ESQL

Добавлено: IB

Описание: Создаёт триггер — блок PSQL кода, который автоматически выполняется при наступлении определенных событий в базе данных или изменений данных в таблице или представлении.

Синтаксис:

```
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 (по умолчанию 0)

<*declarations*> ::= См. *PSQL::DECLARE* для информации о полном синтаксисе

- Объявления «Legasy» и «sql2003» для триггеров по своему смыслу идентичны и отличаются только своим синтаксисом;
- Триггера с меньшей позицией (POSITION) выполняются раньше. Позиция номера триггера не обязательно должна быть уникальной, но если два или более триггеров имеют одинаковые позиции, то очерёдность их срабатывания не определена;
- При создании триггера каждый тип события на его срабатывание (INSERT, UPDATE или DELETE) не должен упоминаться более одного раза.

Синтаксис SQL-2003-совместимых триггеров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать как альтернативу SQL-2003-совместимый синтаксис для триггеров на таблицы и представления. Вместо указания директивы “FOR *relationname*” перед типом события и дополнительными директивами Вы можете использовать “ON *relationname*” после них, как показано ранее в этой главе в описании синтаксиса.

Пример:

```
CREATE TRIGGER BIU_BOOKS
ACTIVE BEFORE INSERT OR UPDATE POSITION 10
ON BOOKS
AS
BEGIN
        IF (NEW.ID IS NULL)
                THEN NEW.ID = NEXT VALUE FOR GEN_BOOKIDS;
END
```

DATABASE триггера

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 введена возможность создавать триггера для базы данных, срабатывающие на события соединения (CONNECT, DISCONNECT) и транзакции (TRANSACTION START, TRANSACTION COMMIT и TRANSACTION ROLLBACK). Только SYSDBA или владелец базы данных могут создавать, изменять или удалять такие триггера.

Синтаксис:

```
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 (по умолчанию 0)

<declarations> ::= См. PSQL::DECLARE для информации о полном синтаксисе
```

Пример:

Первый пример показывает логирование подключённых пользователей:

```
CREATE OR ALTER TRIGGER DB_CONNECT
ACTIVE ON CONNECT POSITION 10
AS
    DECLARE VARIABLE REMOTE_PROTOCOL VARCHAR(8);
    DECLARE VARIABLE APP_NAME VARCHAR(253);
    DECLARE VARIABLE SERVER_PID INTEGER;
    DECLARE VARIABLE PC_USER_PID INTEGER;
BEGIN
    SELECT
        M_A.MON$REMOTE_PROTOCOL,
        M_A.MON$REMOTE_PROCESS,
        M_A.MON$SERVER_PID, M_A.MON$REMOTE_PID
    FROM MON$ATTACHMENTS M_A
    WHERE M_A.MON$ATTACHMENT_ID =
            CURRENT_CONNECTION
    INTO :REMOTE_PROTOCOL,
        :APP_NAME,
        :SERVER_PID, :PC_PID;

    INSERT INTO USER_CONNECTED (
        USER_LOGIN, CONNECT_ID,
        PC_IP,
        REMOTE_PROTOCOL, APP_NAME,
        SERVER_PID, PC_PID, DATE_START_CONNECT)
    VALUES (
        CURRENT_USER, CURRENT_CONNECTION,
        RDB$GET_CONTEXT('SYSTEM', 'CLIENT_ADDRESS'),
        :REMOTE_PROTOCOL, :APP_NAME,
        :SERVER_PID, :PC_USER_PID, CURRENT_TIMESTAMP);
END
```

Следующий пример показывает вызов исключения при подключении к базе данных нежелательного пользователя:

```
CREATE EXCEPTION EX_CONNECT
    'Соединение с базой данных для пользователя ';

CREATE OR ALTER TRIGGER TRG_CONN
ACTIVE ON CONNECT POSITION 5
```

```
AS
BEGIN
        IF (CURRENT_USER = 'BAD_USER')
                THEN EXCEPTION EX_CONNECT USER ||
                        CURRENT_USER || ' запрещено!';
END;
```

Выполнение триггеров базы данных и обработка исключений:

- Триггера на события CONNECT и DISCONNECT выполняются в специально созданной для этого транзакции. Если при обработке триггера не было вызвано исключение, то транзакция подтверждается. Не перехваченные исключения откатят транзакцию и:
  - В случае триггера на событие CONNECT соединение разрывается, а исключения возвращается клиенту;
  - Для триггера на событие DISCONNECT соединение разрывается, как это и предусмотрено, но исключения не возвращается клиенту.
- Триггера на событие TRANSACTION срабатывают при открытии транзакции, её подтверждении или отмене. Не перехваченные исключения обрабатываются в зависимости от типа события TRANSACTION:
  - Для события START исключение возвращается клиенту, а транзакция отменяется;
  - Для события COMMIT исключение возвращается клиенту, действия, выполненные триггером, и транзакция отменяются;
  - Для события ROLLBACK исключение не возвращается клиенту, а транзакция, как и предусмотрено, отменяется.
- Из вышеизложенного следует, что нет прямого способа узнать, какой триггер (DISCONNECT или ROLLBACK) вызвал исключение;
- Также ясно, что Вы не сможете подключиться к базе данных в случае исключения в триггере на событие CONNECT, а также отменяется старт транзакции при исключении в триггере на событие TRANSACTION START. В обоих случаях база данных эффективно блокируется, в то время как Вы собирались работать с ней. См. примечание ниже, где описан способ обойти такие ситуации (Catch-22).
- В случае двухфазных транзакций триггера на событие TRANSACTION START срабатывают в режиме подготовки выполнения, но их действия не подтверждаются до момента выполнения во всех базах данных, участвующих в транзакции.

Примечание

В некоторые утилиты командной строки Firebird были добавлены новые ключи для отключения триггеров на базу данных:

```
gbak -nodbtriggers
isql -nodbtriggers
nbackup -T
```

Эти ключи могут использоваться только SYSDBA или владельцем базы данных.

Декларация переменных с использованием TYPE OF COLUMN

Добавлено: 2.5

Описание: По аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Используется только тип данных, а в случае строковых типов ещё и набор символов и параметры сортировки. Ограничения и значения по умолчанию столбца никогда не используются. См. *PSQL::DECLARE* для информации о полном синтаксисе.

Домены вместо типов данных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать домены вместо SQL типов данных при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

COLLATE при декларации переменных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать предложение COLLATE при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

NOT NULL при декларации переменных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать ограничение NOT NULL при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

Триггеры на несколько типов событий

Добавлено: 1.5

Описание: Срабатывание триггеров можно теперь определять на несколько событий (INSERT и/или UPDATE и/или DELETE). Также были добавлены три новых контекстных переменных (INSERTING, UPDATING и DELETING), которые позволяют определить тип операции и в зависимости от него выполнять код в триггере.

Пример:

```
CREATE TRIGGER BIU_PARTS FOR PARTS
BEFORE INSERT OR UPDATE
AS
BEGIN
-- Код для вставки:
        IF (INSERTING AND NEW.ID IS NULL)
                THEN NEW.ID = GEN_ID(GEN_PARTREC_ID, 1);

-- Общий код для всех типов операций:
        NEW.PARTNAME_UPPER = UPPER(NEW.PARTNAME);
END
```

Примечание

В триггерах на несколько типов событий всегда доступны контекстные переменные OLD и NEW. Если Вы используете их неправильно (например, OLD при вставке или NEW при удалении), то происходит следующее:

- При чтении значения столбца возвращается NULL;
- Попытка присвоить значение столбцу вызовет исключение.

Блок BEGIN...END может быть пустым

Изменено: 1.5

Описание: Начиная с Firebird 1.5 блок BEGIN...END может быть пустым, т.о. создаётся своеобразная «заглушка», позволяющая избежать написания фиктивных операторов.

Пример:

```
CREATE TRIGGER BI_ATABLE FOR ATABLE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
END
```

Нет счётчика изменения метаданных для оператора CREATE TRIGGER

Изменено: 1.0

Описание: В отличие от InterBase, Firebird не увеличивает счетчик изменений метаданных в соответствующей таблице при использовании операторов CREATE, ALTER или DROP TRIGGER. Более подробно это описано в разделе ALTER TRIGGER .

В коде триггера разрешён оператор PLAN

Изменено: 1.5

Описание: До Firebird 1.5 триггер, содержащий предложение PLAN будет не компилировался. В настоящее время надлежаще оформленный план может быть включён в код триггера и будет использоваться.

ALTER TRIGGER

Доступно: DSQL, ESQL

Описание: Изменяет существующий триггер. Обычные триггеры не могут быть изменены в триггеры базы данных или наоборот.

Синтаксис:

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

[<*statements*>]
END]

• Подробное описание параметров приведено в разделе CREATE TRIGGER .

Триггеры базы данных

Добавлено: 2.1

Описание: Синтаксис оператора ALTER TRIGGER был расширен для поддержки триггеров базы данных. Полное описание приведено в разделе CREATE TRIGGER .

Декларация переменных с использованием TYPE OF COLUMN

Добавлено: 2.5

Описание: По аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Используется только тип данных, а в случае строковых типов ещё и набор символов и параметры сортировки. Ограничения и значения по умолчанию столбца никогда не используются. См. *PSQL::DECLARE* для информации о полном синтаксисе.

Домены вместо типов данных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать домены вместо SQL типов данных при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

COLLATE при декларации переменных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать предложение COLLATE при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

NOT NULL при декларации переменных

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать ограничение NOT NULL при декларации локальных переменных в триггерах. См. *PSQL::DECLARE* для информации о полном синтаксисе.

Триггеры на несколько типов событий

Добавлено: 1.5

Описание: Синтаксис оператора ALTER TRIGGER был расширен для поддержки триггеров, срабатывающих на несколько событий (INSERT и/или UPDATE и/или DELETE). Полное описание приведено в разделе CREATE TRIGGER .

Ограничение на изменение используемых триггеров

Изменено: 2.0, 2.1

Описание: Только в Firebird 2.0 существует ограничение, запрещающее удалять, изменять или пересоздавать триггера или хранимые процедуры, если они использовались со времени открытия базы данных. Это ограничение было снято в Firebird 2.0.1. Однако выполнение этих операций на «живой» (имеющей подключения) базе данных потенциально опасно и они должны быть сделаны с предельной осторожностью.

Нет счётчика изменения метаданных для оператора ALTER TRIGGER

Изменено: 1.0

Описание: В InterBase каждый раз, когда используется операторы CREATE, ALTER или DROP TRIGGER, увеличивается счетчик изменений метаданных в соответствующей таблице. После достижения счётчиком значения 255 дальнейшее изменение метаданных не возможно (хотя Вы все еще можете работать с данными). Для того, чтобы снова сбросить счётчик (тем самым вновь разрешив изменение метаданных) необходимо выполнить цикл резервного копирования и восстановления базы данных.

Хотя сама по себе операция сброса счётчика изменений метаданных сама по себе является полезной функцией, это также означает, что пользователи, которые регулярно использовали оператор ALTER TRIGGER, например, для отключения триггеров во время операций массового импорта вынуждены резервировать и восстанавливать базу данных гораздо чаще, чем это необходимо.

Поскольку изменения в триггерах не приводят к структурным изменениям в

самой таблице, Firebird больше не увеличивает счетчик изменений метаданных при выполнении операторов CREATE, ALTER или DROP TRIGGER. Хотя остаётся верным следующее: если счетчик изменений метаданных для триггера равен 255, то Вы больше не можете создавать, изменять или удалять триггеры для этой таблицы.

CREATE OR ALTER TRIGGER

Доступно: DSQL

Добавлено: 1.5

Описание: Если триггер не существует, то он создаётся с помощью оператора CREATE TRIGGER. Если триггер существует, то он изменяется и перекомпилируется, при этом существующие права и зависимости сохраняются.

Синтаксис: Такой же, как и для оператора CREATE TRIGGER .

DROP TRIGGER

Доступно: DSQL, ESQL

Ограничение на удаление используемого триггера

Изменено: 2.0, 2.1

Описание: Только в Firebird 2.0 существует ограничение, запрещающее удалять, изменять или пересоздавать триггера или хранимые процедуры, если они использовались со времени открытия базы данных. Это ограничение было снято в Firebird 2.0.1. Однако выполнение этих операций на «живой» (имеющей подключения) базе данных потенциально опасно и они должны быть сделаны с предельной осторожностью.

Нет счётчика изменения метаданных для оператора DROP TRIGGER

Изменено: 1.0

Описание: В отличие от InterBase, Firebird не увеличивает счетчик изменений метаданных в соответствующей таблице при использовании операторов CREATE, ALTER или DROP TRIGGER. Более подробно это описано в разделе ALTER TRIGGER .

RECREATE TRIGGER

Доступно: DSQL

Добавлено: 2.0

Описание: Создаёт или пересоздаёт триггер. Если триггер с таким именем существует, то оператор RECREATE TRIGGER попытается удалить его и создать новый. Если существующий триггер используется, то выполнение оператора RECREATE TRIGGER вызовет ошибку.

Синтаксис: Такой же, как и для оператора CREATE TRIGGER .

Ограничение на пересоздание используемого триггера

Изменено: 2.0, 2.1

Описание: Только в Firebird 2.0 существует ограничение, запрещающее удалять, изменять или пересоздавать триггера или хранимые процедуры, если они использовались со времени открытия базы данных. Это ограничение было снято в Firebird 2.0.1. Однако выполнение этих операций на «живой» (имеющей подключения) базе данных потенциально опасно и они должны быть сделаны с предельной осторожностью.

VIEW

CREATE VIEW

Доступно: DSQL

Синтаксис:

```
CREATE VIEW viewname [<full_column_list>]
AS
    <select_statement>
    [WITH CHECK OPTION]

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

Представления могут делать выборку из хранимых процедур

Изменено: 2.5

Описание: Начиная с Firebird 2.5 представления могут делать выборку данных их селективных хранимых процедур.

Пример:

```
CREATE VIEW LOW_BONES AS
    SELECT
        SP1.ID, SP1.NAME, SP1.DESCRIPTION
    FROM THEM_BONES('HUMAN') SP1
    WHERE NAME IN ('LEG_BONE', 'FOOT_BONE', 'TOE_BONE')
```

Представления могут выводить имена столбцов из связанных таблиц или GROUP BY

Изменено: 2.5

Описание: Начиная с Firebird 2.5 представления выводить имена столбцов из связанных таблиц (derived table) или включённых в предложение GROUP BY. Ранее надо было указать явным образом псевдонимы для таких столбцов (в столбце или в полном списке).

Пример:

```
CREATE VIEW TICKLE AS
    SELECT T
    FROM (SELECT T FROM TACKLE)

CREATE VIEW VSTOCKS AS
    SELECT KIND, SUM(STOCK) S
    FROM STOCKS
    GROUP BY KIND
```

Поддержка псевдонимов для каждого столбца при создании представления

Изменено: 2.1

Описание: Начиная с Firebird 2.1 Вы можете использовать псевдонимы (алиасы) в операторе SELECT. Вы можете использовать псевдонимы для всех или части столбцов (или вообще не использовать их), при этом каждый псевдоним становится именем соответствующего столбца представления.

Синтаксис (неполный):

```
CREATE VIEW viewname [<full_column_list>]
AS
SELECT <column_def> [, <column_def> ...]
…

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

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

Примечания:

- Если также существует полный список столбцов, то задание псевдонимов не имеет смысла, поскольку они будут переопределены именами в списке столбцов;
- Полный список столбцов, используемых в представлениях, обязателен для , оператора SELECT, содержащего выражения на основе столбцов или идентичные имена столбцов. Теперь Вы можете опустить полный список столбцов при условии, что укажите их в операторе SELECT.

Полная поддержка синтаксиса оператора SELECT

Изменено: 2.0, 2.5

Описание: Начиная с Firebird 2.0 представления считаются законченными операторами SELECT. Таким образом, допускается использование следующих элементов в представлениях: FIRST, SKIP, ROWS, ORDER BY, PLAN и UNION.

Примечание

Начиная с Firebird 2.5 не является необходимым составлять список столбцов, если представление основано на операторе UNUION:

```
CREATE VIEW VPLANES AS
        SELECT MAKE, MODEL FROM JETS
                UNION
        SELECT MAKE, MODEL FROM PROPS
```

```
UNION
SELECT MAKE, MODEL FROM GLIDERS
```

Имена столбцов будут взяты из объединения. При этом, конечно Вы можете сами задать список колонок.

Применение предложения PLAN, запрещённое в 1.5, вновь разрешено в 2.0

Изменено: 1.5, 2.0

Описание: Firebird версий 1.5x запрещал использование предложения PLAN в представлениях. Начиная с версии 2.0 использование предложения PLAN вновь разрешено.

Триггеры на обновляемые представления блокируют автоматическую прямую запись

Изменено: 2.0

Описание: В версиях до 2.0 Firebird часто не блокировал автоматическую прямую запись (auto-writethrough) в базовую таблицу, если у обновляемого представления имелись один или несколько триггеров. При этом изменения данных могли непреднамеренно выполняться дважды, иногда приводя к повреждению данных и другим ошибкам. Начиная с Firebird 2.0 это ошибочное поведение было исправлено: теперь при наличии триггеров на обновляемое представление изменения не будут автоматически попадать в таблицу — либо это делает триггер, либо запись не происходит. Старое поведение описано в InterBase 6 *Data Definition Guide under Updating views with triggers.*

Предупреждение

Отдельные разработчики применяют код, который рассчитывает или использует в своих интересах предшествующее поведение. Такой код должен быть исправлен для версий Firebird 2.0 и выше — без правки кода изменения не будут вноситься в таблицы.

Представление с не участвующими столбцами NOT NULL в базовой таблице может вставлять записи

Изменено: 2.0

Описание: Любое представление, базовая таблица которой содержит один

или несколько NOT NULL столбцов, даже если они отсутствуют в представлении, является представлением только для чтения. Его можно сделать обновляемым с помощью триггеров, но даже в этом случае операция INSERT в таких представлениях не пройдёт, т.к. ограничения NOT NULL проверяются до работы триггера. Начиная с Firebird 2.0 это поведение исправлено, если для представления существует триггер, обеспечивающий вставку записей.

Пример:

Представление, приведённое ниже, вызывает ошибку при любой попытке вставки записей в версиях Firebird 1.5 и ниже. В Firebird 2.0 вставка в такое представление разрешена.

```
CREATE TABLE BASE (
        X INTEGER NOT NULL,
        Y INTEGER NOT NULL);

CREATE VIEW V_BASE
AS
        SELECT X
        FROM BASE;

SET TERM ^ ;

CREATE TRIGGER BI_BASE FOR V_BASE
ACTIVE BEFORE INSERT
AS
BEGIN
        IF (NEW.X IS NULL)
                THEN NEW.X = 33;
        INSERT INTO BASE VALUES (NEW.X, 0);
END
^

SET TERM ;^
```

Примечания:

- Обратите внимание, что проблема, описанная выше, в примере произошла на NOT NULL столбце (Y), который *не входит* в столбцы представления;
- Как ни странно, такой проблемы не было бы, если бы базовая таблица имела триггер, проверяющий ввод значения NULL в столбец и присваивающий в этом случае допустимое значение. Но тогда существует опасность, что

вставка произойдёт два раза — это связано с ошибкой автоматической прямой записи, которая также исправлена в Firebird 2.

ALTER VIEW

Доступно: DSQL

Добавлено: 2.5

Описание: Начиная с Firebird 2.5 доступен оператор ALTER VIEW, позволяющий изменять представление без его предварительного удаления. Существующие зависимости сохраняются.

Синтаксис: Такой же, как и у CREATE VIEW

CREATE OR ALTER VIEW

Доступно: DSQL

Добавлено: 2.5

Описание: Оператор CREATE OR ALTER VIEW создаёт представление, если оно не существует. В противном случае он изменит представление с сохранением существующих зависимостей.

Синтаксис: Такой же, как и у CREATE VIEW

RECREATE VIEW

Доступно: DSQL

Добавлено: 1.5

Описание: Создаёт или пересоздаёт представление. Если представление с таким именем уже существует, то оператор RECREATE VIEW попытается удалить его и создать новое. Оператор RECREATE VIEW не выполнится, если существующее представление используется.

Синтаксис: Такой же, как и у CREATE VIEW

Глава 7

Операторы DML

DELETE

Доступно: DSQL, ESQL, PSQL

Описание: Удаляет строки из таблицы базы данных (или из одного или нескольких базовых таблиц представления) в зависимости от условий в предложениях WHERE и ROWS.

Синтаксис:

```
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> ::= Любые целые числа
<values> ::= value_expression [, value_expression ...]
<variables> ::= :varname [, :varname ...]
```

Ограничения

- Директива TRANSACTION доступна только в EQSL;
- В простой DSQL сессии заявление WHERE CURRENT OF не имеет особого смысла, так как не существует оператора DSQL для создания курсора;
- Предложения PLAN, ORDER BY и ROWS недоступны в ESQL;
- Предложение RETURNING недоступно в ESQL;
- подпункт “INTO <*variables*>” доступен только в PSQL;
- При возврате значения в контекстную переменную NEW этому имени не должно предшествовать двоеточие (“: ”).

Использование COLLATE для столбцов с текстовым BLOB

Добавлено: 2.0

Описание: Предложение COLLATE доступно теперь и для столбцов текстовых BLOB.

Пример:

```
DELETE FROM MYTABLE
    WHERE NAMEBLOB COLLATE PT_BR = 'Joa̧o'
```

ORDER BY

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Оператор DELETE теперь поддерживает предложение ORDER BY. Это имеет смысл в сочетании с предложением ROWS, но справедливо и без него.

PLAN

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Оператор DELETE теперь поддерживает предложение PLAN, так что пользователи могут оптимизировать его работу вручную.

Использование алиаса делает недоступным использование полного имени таблицы

Изменено: 2.0

Описание: Если в Firebird 2.0 или выше Вы дадите таблице или представлению псевдоним (алиас), то Вы должны везде использовать псевдоним, а

не имя таблицы, если Вы хотите корректно определять имя столбца.

Пример:

Корректное использование:

```
DELETE FROM CITIES
    WHERE NAME STARTING 'Vash'

DELETE FROM CITIES
    WHERE CITIES.NAME STARTING 'Vash'

DELETE FROM CITIES C
    WHERE NAME STARTING 'Vash'

DELETE FROM CITIES C
    WHERE C.NAME STARTING 'Vash'
```

Теперь невозможно:

```
DELETE FROM CITIES C
    WHERE CITIES.NAME STARTING 'Vash'
```

RETURNING

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Оператор DELETE, удаляющий самое большее одну строку, может дополнительно включать предложение RETURNING для возврата значений удаляемой строки. Предложение, если присутствует, не должно содержать все столбцы и может также содержать другие столбцы или выражения.

Пример:

```
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;
```

Примечания:

- В DSQL оператор с предложением RETURNING всегда возвращает одну строку. Если не было удалённой записи, то все поля возвращаются со значением NULL. Такое поведение может измениться в более поздней версии Firebird. В PSQL, если ни одна строка не была удалена, ничего не возвращается, и значения переменных сохраняют существующие значения.

ROWS

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Ограничивает количество строк для удаления на указанное число или диапазон.

Синтаксис:

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

<*m*>, <*n*> ::= Любые целые числа или выражение, приводимое к целому числу

С единственным параметром **m** удаление ограничено первыми **m** строками набора данных, определенного таблицей или представлением и дополнительными условиями в предложениях WHERE и ORDER BY.

Необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то будет удален весь набор;
- Если **m = 0**, то ничего не будет удалено;
- Если **m < 0**, то оператор удаления выдаст ошибку.

С двумя параметрами **m** и **n** удаление ограничено строками начиная с **m** и до **n** включительно. Номера строк начинаются с 1.

Для двух аргументов необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то ничего не будет удалено;
- Если число **m** не превышает общего количества строк в наборе данных, а **n** превышает, то будут удалены строки от **m** до конца набора данных;
- Если **m < 1** и **n < 1**, то оператор удаления выдаст ошибку;
- Если **n = m -1**, то ничего не будет удалено;
- Если **n < m -1**, то оператор удаления выдаст ошибку.

Предложение ROWS также можно использовать с операторами SELECT и UPDATE .

EXECUTE BLOCK

Доступно: DSQL

Добавлено: 2.0

Изменено: 2.1, 2.5

Описание: Выполняет блок кода PSQL, как будто это хранимая процедура, опционально со входными и выходными параметрами и объявлениями переменных. Это позволяет пользователю выполнять «на лету» PSQL код в контексте DSQL.

Синтаксис:

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

<inparams> ::= <param_decl> = ? [, <inparams> ]
<outparams> ::= <param_decl> [, <outparams>]
<param_decl> ::= paramname <type> [NOT NULL]
                    [COLLATE collation]
<type> ::= sql_datatype | [TYPE OF] domain | TYPE OF COLUMN rel.col
<declarations> ::= См. PSQL::DECLARE для информации о полном
```

синтаксисе.

Пример:

В данном примере в таблицу ASCIITABLE вставляются числа от 0 до 127 и соответствующие им ASCII символы:

```
EXECUTE BLOCK
AS
        DECLARE I INTEGER = 0;
BEGIN
        WHILE (I < 128) DO
                BEGIN
                        INSERT INTO ASCIITABLE VALUES (
                                :I, ASCII_CHAR(:I));
                        I = I + 1;
                END
END
```

В следующем примере EXECUTE BLOCK вычисляет среднее геометрическое двух чисел и возвращает его пользователю:

```
EXECUTE BLOCK (
        X DOUBLE PRECISION = ?,
        Y DOUBLE PRECISION = ?)
RETURNS (GMEAN DOUBLE PRECISION)
AS
BEGIN
        GMEAN = SQRT(:X*:Y);
        SUSPEND;
END
```

Поскольку этот блок имеет входные параметров, он должен быть подготовлен. После этого можно задать значения параметров и выполнить блок. Как это будет сделано (если это возможно вообще – см. примечания ниже) зависит от клиентского программного обеспечения.

В следующем примере входными параметрами блока являются два целых числа — наименьшее и наибольшее. В цикле по возрастанию от наименьшего и наибольшему числу вычисляются и выдаются клиенту квадрат, куб и четвёртая степень числа:

```
EXECUTE BLOCK (
        SMALLEST INTEGER = ?,
```

```
        LARGEST INTEGER = ?)
RETURNS (
        NUMBER INTEGER,
        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
```

Опять же, способ присвоения значений параметрам зависит от клиентского программного обеспечения.

Пример, выбирающий из системных таблиц имена таблиц, ограничений, индексов, флаг активности и статистику индекса:

```
EXECUTE BLOCK RETURNS(
        RELATION_NAME VARCHAR(32),
        CONSTRAINT_NAME VARCHAR(32),
        INDEX_NAME VARCHAR(32),
        INDEX_ACTIVE VARCHAR(10),
        INDEX_STATISTIC DOUBLE PRECISION)
AS
BEGIN
        FOR SELECT
                CAST(RI.RDB$RELATION_NAME AS VARCHAR(31))
                        RELATION_NAME,
                CAST(RC.RDB$CONSTRAINT_NAME AS VARCHAR(31))
                        CONSTRAINT_NAME,
                CAST(RI.RDB$INDEX_NAME AS VARCHAR(31))
                        INDEX_NAME,
                CAST(
                        CASE COALESCE(RI.RDB$INDEX_INACTIVE, 0)
                                WHEN 0 THEN 'ACTIVE'
```

```
                    ELSE 'INACTIVE'
                    END AS VARCHAR(10)) INDEX_ACTIVE,
                RI.RDB$STATISTICS INDEX_STATISTIC
            FROM RDB$INDICES RI
                JOIN RDB$RELATION_CONSTRAINTS RC ON
            RC.RDB$RELATION_NAME = RI.RDB$RELATION_NAME
            WHERE RI.RDB$SYSTEM_FLAG = 0
            ORDER BY 1
            INTO
                :RELATION_NAME,
                :CONSTRAINT_NAME,
                :INDEX_NAME,
                :INDEX_ACTIVE,
                :INDEX_STATISTIC DO
            SUSPEND;
        END
```

Примечания:

- Некоторые клиенты, особенно те, которые разрешают пользователю подставить сразу несколько операторов, требуют заключения оператора EXECUTE BLOCK строками терминатора SET TERM:

```
SET TERM ^;
EXECUTE BLOCK (...)
AS
BEGIN
    STATEMENT1;
    STATEMENT2;
END
^
SET TERM ;^
```

В ISQL клиенте Firebird Вы должны установить разделитель на нечто иное, чем ";" до ввода оператора EXECUTE BLOCK. В противном случае ISQL, ориентированный на выполнении строк, оканчивающейся на точку с запятой, будет пытаться выполнить часть, которую вы ввели, как только он встречает первую точку с запятой.

- Выполнение блока без входных параметров должно быть возможно в любом клиенте Firebird, который позволяет пользователю вводить собственные операторы DSQL. Если есть входные параметры, все становится сложнее: эти параметры должны получить их значения перед подготовкой блока, но до

его выполнения. Это требует специального обеспечения, которое поддерживает не каждое клиентское приложение (Собственный ISQL клиент Firebird, к примеру, не поддерживает);

- Сервер принимает только вопросительные знаки (“?”) в качестве знака параметра, а не “:a”, “:MyParam” и т.д. или собственно значения вместо знака параметра. Клиентское программное обеспечение может поддерживать параметры в виде ":XXX", оно всё равно должно уметь преобразовать знаки параметров в “?” хотя перед отправкой их на сервер;
- Если блок имеет выходные параметры, Вы должны использовать внутри него оператор SUSPEND, в противном случае клиенту ничего не будет возвращено;
- Вывод данных всегда возвращается в виде результирующего набора, так же, как и у оператора SELECT. Вы не можете использовать предложения RETURNING_VALUES или INTO (за пределами блока) в заданные переменные, даже если возвращается только одна строка результата.

COLLATE в объявлениях переменных и параметров

Изменено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование предложения COLLATE при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
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 в объявлениях переменных и параметров

Изменено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование NOT NULL ограничения при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
EXECUTE BLOCK (
    A INTEGER NOT NULL = ?,
    B INTEGER 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 = 'PRODUCT меньше нуля!';
    ELSE IF (:PRODUCT > 0) THEN
        MESSAGE = 'PRODUCT больше нуля!';
    ELSE
        MESSAGE = 'PRODUCT равен нулю!';
    SUSPEND;
END
```

Домены вместо типа данных

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование доменов вместо типов данных SQL при объявлении входных и выходных параметров и локальных переменных. Если перед названием домена дополнительно используется предложение "TYPE OF", то используется только тип данных домена — не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию. Если домен текстового типа, то всегда используется его набор символов и порядок сортировки.

Пример:

```
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
```

Предупреждение

Для входных параметров при выполнении операции сравнения (например, тесты на равенство) не принимается во внимание сортировка, указанная в домене. Эта ошибка исправлена в Firebird 3.

TYPE OF COLUMN в объявлениях параметров и переменных

Добавлено: 2.5

Описание: По аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Используется только тип данных, а в случае строковых типов ещё и набор символов и параметры сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Пример:

```
CREATE TABLE NUMBERS (
    BIGNUM NUMERIC(18),
    SMALLNUM NUMERIC(9))

EXECUTE BLOCK (
    DIVIDEND TYPE OF COLUMN NUMBERS.BIGNUM = ?,
    DIVISOR TYPE OF COLUMN NUMBERS.SMALLNUM = ?)
RETURNS (
    QUOTIENT TYPE OF COLUMN NUMBERS.BIGNUM,
    REMAINDER TYPE OF COLUMN NUMBERS.SMALLNUM)
AS
BEGIN
    QUOTIENT = :DIVIDEND / :DIVISOR;
    REMAINDER = MOD (:DIVIDEND, :DIVISOR);
    SUSPEND;
```

```
END
```

Предупреждение

Для входных параметров при выполнении операции сравнения (например, тесты на равенство) не принимается во внимание сортировка, указанная в столбце таблицы. Для локальных переменных это не так. Эта ошибка исправлена в Firebird 3.

EXECUTE PROCEDURE

Доступно: DSQL, ESQL, PSQL

Изменено: 1.5

Описание: Выполняет хранимую процедуру. В Firebird 1.0.x, как и в InterBase, любые входные параметры для хранимых процедур должны быть представлены как литералы, переменные базового языка (в ESQL) или локальные переменные (в PSQL). Начиная с Firebird 1.5 входные параметры могут также быть выражениями (композитными), кроме статического ESQL.

Синтаксис:

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

<in_item> ::= <inparam> [<nullind>]
<out_item> ::= <outvar> [<nullind>]
<inparam> ::= выражение для входных параметров
<outvar> ::= базовый язык или переменная PSQL для возвращаемого значения
<nullind> ::= [INDICATOR]:host_lang_intvar
```

- Выражения в параметрах не поддерживаются в статическом ESQL и в версиях Firebird ниже 1.5;
- NULL индикаторы доступны только в коде ESQL. Они должны быть переменными базового языка целого типа;
- В ESQL именам переменных, используемых в качестве входных и выходных параметров должно предшествовать двоеточие (":"). В PSQL опциональное двоеточие, как правило, необязательно, но запрещено для

контекстных переменных OLD и NEW.

Пример:

PSQL (с опциональным двоеточием):

```
EXECUTE PROCEDURE MAKEFULLNAME(
    :FIRSTNAME, :MIDDLENAME, :LASTNAME)
RETURNING_VALUES :FULLNAME;
```

EQSL (с обязательным двоеточием):

```
EXEC SQL
    EXECUTE PROCEDURE MAKEFULLNAME(
        :FIRSTNAME, :MIDDLENAME, :LASTNAME)
    RETURNING_VALUES :FULLNAME;
```

В утилите командной строки **isql** (с уже введёнными параметрами):

```
EXECUTE PROCEDURE MAKEFULLNAME(
    'J', 'Edgar', 'Hoover');
```

Примечание: Не используйте в **isql** предложение RETURNING_VALUES. Результаты выполнения процедуры будут показаны автоматически.

Пример использования выражений в PSQL (в Firebird 1.5 и выше:

```
EXECUTE PROCEDURE MAKEFULLNAME(
    'Mr./Mrs. ' || FIRSTNAME, MIDDLENAME, UPPER(LASTNAME))
RETURNING_VALUES FULLNAME;
```

INSERT

Доступно: DSQL, ESQL, PSQL

Описание: Добавляет строку в таблицу базы данных или в одну или несколько таблиц, используемых представлением. Значения вставляемых столбцов могут быть указаны в предложении VALUES или вообще не указаны (в обоих случаях только при вставке одной строки); также они могут быть взяты из оператора SELECT - от 0 до любого количества строк.

Синтаксис:

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

<column_list> ::= colname [, colname ...]
<value_source> ::= VALUES (<value_list>) | <select_stmt>
<value_list> ::= value_expression [, value_expression ...]
<variables> ::= :varname [, :varname ...]
```

<*select_stmt*> ::= оператор SELECT, результирующий набор которого соответствует вставляемым столбцам

Ограничения

- Директива TRANSACTION доступна только в EQSL;
- Предложение RETURNING не доступно в ESQL;
- Предложение “INTO <*variables*>” доступно только в PSQL;
- При возврате значений в контекстную переменную NEW ей не должно предшествовать двоеточие (“:”);
- Начиная с версии Firebird 2.0 столбец не может упоминаться в списке для вставки более одного раза.

INSERT ... DEFAULT VALUES

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Предложение DEFAULT VALUES позволяет вставлять записи вообще без указания значения - ни прямо, ни через оператор SELECT. Это возможно только если каждый столбец с NOT NULL или проверкой (CHECK) таблицы имеет объявленное значение по умолчанию или получает значение в триггере до вставки (BEFORE INSERT). Кроме этого выражения в триггере, присваивающие значения таким столбцам, не должны зависеть от наличия входных значений.

Пример:

```
INSERT INTO JOURNAL
    DEFAULT VALUES
```

```
RETURNING ENTRY_ID
```

Предложение RETURNING

Доступно: DSQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: *При вставке одной строки* оператор INSERT опционально может использовать предложение RETURNING для возврата значений из вставленной строки. Предложение, если присутствует, не должно содержать все вставляемые столбцы, но также может содержать другие столбцы или выражения. Возвращаемые значения учитывают все изменения, сделанные в триггере до вставки (BEFORE INSERT), но не учитывают изменения, сделанные в триггере после вставки (AFTER INSERT).

Пример:

```
INSERT INTO SCHOLARS (
    FIRSTNAME, LASTNAME, ADDRESS, PHONE, EMAIL)
VALUES (
    'Henry', 'Higgins', '27A Wimpole Street', '+7 (100) 323-1212', 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;
```

Примечания:

- Предложение RETURNING поддерживается только при вставке и — начиная с версии Firebird 2.1 — при использовании для вставки одиночного (singleton), т.е. возвращающего только одну строку, оператора SELECT;
- В DSQL предложение RETURNING *всегда* возвращает одну строку. Если строка не была вставлена, то DSQL возвратит строку со значениями столбцов NULL. Это поведение может быть изменено в следующих версиях.

В PSQL , если строка не была вставлена, возвращаемые значения переменных останутся такими же, как и до операции вставки.

Примечание переводчика: Предложение RETURNING удобно использовать для получения значения первичного ключа, значение которого формируется в триггере до вставки.

Разрешено использовать UNION в операторе SELECT при вставке

Изменено: 2.0

Описание: Оператор SELECT, используемый для вставки, теперь может содержать предложение UNION.

Пример:

```
INSERT INTO MEMBERS (
        NUMBER, NAME)
SELECT NUMBER, NAME
        FROM NEWMEMBERS
        WHERE ACCEPTED = 1
UNION
SELECT NUMBER, NAME
        FROM SUSPENDEDMEMBERS
        WHERE VINDICATED = 1
```

MERGE

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Объединяет данные в таблицу или представление. Источником данных может быть таблица, представление или производная таблица (derived table), т.е. заключенный в скобки оператор SELECT или CTE. Каждая строка объединения будет использоваться для обновления одной или нескольких записей, вставки новой записи в целевую таблицу, но может и не использоваться. Действия зависят от объявленного условия и предложения WHEN в операторе MERGE. Условие, как правило, содержит сравнение столбцов исходной и целевой таблиц.

Синтаксис:

```
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 ...]
```

Примечание: Разрешается использовать только одно предложение WHEN.

Пример:

```
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)
```

Примечание

Предложение WHEN NOT MATCHED берёт за основу записи из источника данных, указанного в предложении USING. Это значит, что если у исходной записи нет соответствия в целевой таблице, то выполняется оператор INSERT. С другой стороны записи в целевой таблице, не имеющие соответствия в источнике данных, не вызывают никаких действий.

Предупреждение

Если присутствует предложение WHEN MATCHED и нескольким записям из источника данных соответствуют записи в целевой таблице, то операция обновления выполняется многократно; при этом каждое обновление перезаписывает предыдущее. Это поведение не соответствует стандарту: SQL - 2003 определяет, что в таком случае должно быть вызвано исключение .

SELECT

Доступно: DSQL, ESQL, PSQL

Агрегатные функции: Расширенный функционал

Изменено: 1.5

Описание: Начиная с Firebird 1.5 поддерживаются несколько типов смешивания (объединения) и вложения агрегатных функций. Эти типы будут рассмотрены далее — см. SELECT :: GROUP BY .

Смешивание агрегатных функций от различных контекстов

Начиная с Firebird 1.5 разрешено использовать агрегатные функции из разных контекстов внутри одного выражения.

Пример:

```
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
```

Этот “выдуманный” запрос показывает в первом столбце имя таблицы, а во втором индекс этой таблицы с максимальной селективностью и в скобках количество полей таблицы. Конечно, Вы, как правило, отображаете количество полей в отдельной колонке или в столбце с именем таблицы, но здесь целью было продемонстрировать возможность объединения агрегатов из разных контекстов в одном выражении.

Применение переводчика: Отметим, что в Firebird статистика индексов автоматически не пересчитывается. Она обновляется при выполнении оператора SET STATISTICS INDEX INDEX_NAME (также статистика индексов пересчитывается при восстановлении базы данных).

Предупреждение

Firebird версии 1.0 также выполнит данный запрос, но его результат будет неверным.

Агрегатные функции, подзапросы и предложение GROUP BY

Начиная с Firebird 1.5 разрешено использовать в подзапросе столбцы, содержащиеся в предложении GROUP BY.

Примеры:

Первый запрос возвращает для каждой таблицы её имя, ID и количество столбцов. Подзапрос ссылается в условии выборки на поле FLDS.RDB$RELATION_NAME, которое также содержится в предложении GROUP BY:

```
SELECT
        FLDS.RDB$RELATION_NAME RELATION_NAME,
        (SELECT RELS.RDB$RELATION_ID
                FROM RDB$RELATIONS RELS
```

```
            WHERE
                RELS.RDB$RELATION_NAME =
                FLDS.RDB$RELATION_NAME
        ) ID,
        COUNT(*) COUNT_FIELDS
    FROM RDB$RELATION_FIELDS FLDS
    GROUP BY FLDS.RDB$RELATION_NAME
```

Второй запрос возвращает для каждой таблицы её имя, имя её последнего столбца и его позицию:

```
    SELECT
        FLDS.RDB$RELATION_NAME TABLE_NAME,
        (SELECT FLDS2.RDB$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 LAST_FIELD_POS
    FROM RDB$RELATION_FIELDS FLDS
    GROUP BY 1
```

Подзапрос также, как и предложение GROUP BY, содержит поле FLDS.RDB$RELATION_NAME, но это не сразу очевидно, потому что GROUP BY обращается к нему по номеру столбца в запросе.

Агрегатные функции внутри подзапросов

Начиная с Firebird 1.5 разрешено использовать в одиночном (singleton - т.е. возвращающем одну запись) подзапросе агрегатные функции.

Пример:

```
    SELECT
        R.RDB$RELATION_NAME TABLE_NAME,
        SUM(
            (SELECT COUNT(*)
                FROM RDB$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
```

Вложенные вызовы агрегатных функций

Firebird 1.5 позволяет косвенное вложение агрегатных функций при условии, что внутренняя функция от более низкого контекста SQL. Прямое вложение вызовов агрегатной функции, например, “COUNT(MAX(PRICE))”, всё ещё запрещено и вызывает исключение.

Пример: См. выше Агрегатные функции внутри подзапросов , где COUNT(*) используется внутри SUM().

Агрегатные операторы: Более строгое использование HAVING и ORDER BY

Firebird версии 1.5 и выше более строги, чем предыдущие версии к тому, что может быть включено в предложения ORDER BY и HAVING. Если агрегатный оператор входит в состав предложений HAVING или ORDER BY и содержит имя столбца, то синтаксис правильный только при выполнении одного из следующих условий:

- Имя столбца содержится в вызове агрегатной функции (например, “HAVING MAX(SALARY) > 10000”);
- Операнд равен или основан на не агрегатной колонке, которая содержится в списке GROUP BY (по имени или номеру).

"Основан на" означает, что операнд не обязательно должен быть точно таким же, как имя столбца. Предположим, что это не агрегатный столбец "STR" в списке выбора.

Тогда можно использовать такие выражения, как "UPPER(STR)", STR || `!` или "SUBSTRING(STR FROM 4 FOR 2)" в предложении HAVING - даже если этих выражений нет в операторе SELECT или предложении GROUP BY.

Использование COLLATE для столбцов с текстовым BLOB

Добавлено: 2.0

Описание: Предложение COLLATE теперь поддерживается для текстовых BLOB.

Пример:

```
SELECT NAMEBLOB
    FROM MYTABLE
    WHERE NAMEBLOB COLLATE PT_BR = 'Joa̧o'
```

Общие табличные выражения (“WITH ... AS ... SELECT”)

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Общие табличные выражения (Common Table Expressions), сокращённо CTE, могут быть описаны как виртуальные таблицы или представления, определенных в преамбуле основного запроса, которые участвуют в основном запросе. Основной запрос может ссылаться на любое CTE из определенных в преамбуле, как и при выборке данных из обычных таблиц или представлений. CTE могут быть рекурсивными, т.е. ссылающимися сами на себя, но не могут быть вложенными.

Синтаксис:

<*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*> ::= любой оператор SELECT или UNION

<*main-query*> ::= основной оператор SELECT, который может ссылаться на

любое CTE из найденных в преамбуле.

*Примечание переводчика: Ниже приведён более наглядный синтаксис CTE (**http://www.ibase.ru/conf2007/ppt/Firebird.2.1.2007.ru.pdf**).*

WITH [RECURSIVE]**-- **новые ключевые слова

• **CTE_A** -- ***имя табличного выражения***
[(a1, a2, …)] -- ***алиасы полей, не обязательны***
AS (SELECT …), -- ***определение табличного выражения (запрос SELECT)***

• **CTE_B** -- ***второе табличное выражение***
[(b1, b2, …)]
AS (SELECT …), …

• **SELECT …** -- ***главный запрос, использующий***
FROM CTE_A, CTE_B, -- ***и табличные выражения,***
TAB1, TAB2 -- ***и обычные таблицы***
WHERE …

Пример:

```
WITH DEPT_YEAR_BUDGET AS (
    SELECT
        FISCAL_YEAR,
        DEPT_NO,
        SUM(PROJECTED_BUDGET) BUDGET
    FROM PROJ_DEPT_BUDGET
    GROUP BY FISCAL_YEAR, DEPT_NO
)
SELECT
    D.DEPT_NO, D.DEPARTMENT,
    DYB_2008.BUDGET 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)
```

Примечания:

- Определение CTE может содержать любые конструкции синтаксиса оператора SELECT, пока не закончится преамбула "WITH ...» (операторы WITH не могут быть вложенными);
- CTE могут использовать друг друга, но ссылки не должны иметь циклов;
- CTE могут быть использованы в любой части главного запроса или другого табличного выражения и сколько угодно раз;
- Основной запрос может ссылаться на CTE несколько раз, но с разными алиасами;
- CTE могут быть использованы в операторах INSERT, UPDATE и DELETE как подзапросы;
- CTE могут быть использованы и в PSQL (FOR WITH … SELECT … INTO …):

```
FOR WITH MY_RIVERS AS (
        SELECT *
                FROM RIVERS
                WHERE OWNER = 'me')
SELECT
        NAME, LENGTH
FROM MY_RIVERS
INTO :RNAME, :RLEN DO
        BEGIN
                ...
        END
```

Рекурсивные CTE

Рекурсивное (ссылающееся само на себя) CTE это ОБЪЕДИНЕНИЕ, у которого должен быть по крайней мере один не рекурсивный элемент, к которому привязываются остальные элементы объединения. Не рекурсивный элемент помещается в CTE первым. Рекурсивные члены отделяются от не рекурсивных и друг от друга с помощью UNION ALL. Объединение не рекурсивных элементов может быть любого типа.

Рекурсивное CTE требует наличия ключевого слова RECURSIVE справа от WITH. Каждый рекурсивный член объединения может сослаться на себя только один раз и это должно быть сделано в предложении FROM.

Главным преимуществом рекурсивных CTE является то, что они используют гораздо меньше памяти и процессорного времени, чем эквивалентные рекурсивные хранимые процедуры.

Выполнение рекурсивного CTE с точки зрения сервера Firebird можно описать следующим образом:

- Сервер начинает выполнение с не рекурсивного члена;
- Для каждой выбранной строки он начинает выполнять каждый рекурсивный элемент один за другим, используя текущие значения из не рекурсивного члена как параметры;
- Если во время выполнения экземпляр рекурсивного элемента не выдаёт строк, цикл выполнения переходит на предыдущий уровень и получает следующую строку от внешнего для него набора данных.

Пример рекурсивного CTE:

```
WITH RECURSIVE DEPT_YEAR_BUDGET AS (
    SELECT
        FISCAL_YEAR, DEPT_NO,
        SUM(PROJECTED_BUDGET) 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 H.HEAD_DEPT = D.DEPT_NO
)
SELECT
    D.DEPT_NO, D.INDENT || D.DEPARTMENT 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)
```

Примечания для рекурсивного CTE:

- В рекурсивных членах объединения не разрешается использовать агрегаты (DISTINCT, GROUP BY, HAVING) и агрегатные функции (SUM, COUNT, MAX и т.п.);
- Рекурсивная ссылка не может быть участником внешнего объединения OUTER JOIN;
- Максимальная глубина рекурсии составляет 1024.

Производные таблицы ("SELECT FROM SELECT")

Добавлено: 2.0

Описание: Производная таблица (derived table) это результат выборки оператора SELECT, использующего внешний SELECT, как будто это обычная таблица. Ранее это можно было сделать только при помощи подзапроса в предложении FROM.

Синтаксис:

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

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

Примеры:

Производная таблица в запросе ниже (жирным шрифтом) выводит список имён таблиц в базе данных и количество столбцов в них. Запрос к производной таблице выводит количество полей и количество таблиц с таким количеством полей.

```
SELECT
    FIELDCOUNT,
    COUNT(RELATION) AS NUM_TABLES
FROM (SELECT
            R.RDB$RELATION_NAME 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
```

Тривиальный пример, демонстрирующий использование псевдонима производной таблицы и списка псевдонимов столбцов (оба опциональные):

```
SELECT
    DBINFO.DESCR, DBINFO.DEF_CHARSET
FROM (SELECT *
        FROM RDB$DATABASE) DBINFO
        (DESCR, REL_ID, SEC_CLASS, DEF_CHARSET)
```

Примечания:

- Производные таблицы могут быть вложенными;
- Производные таблицы могут быть объединениями и использоваться в объединениях. Они могут содержать агрегатные функции, подзапросы и соединения, и сами по себе могут быть использованы в агрегатных функциях, подзапросах и соединениях. Они также могут быть хранимыми процедурами или запросами из них. Они могут иметь предложения WHERE, ORDER BY и GROUP BY, указания FIRST, SKIP или ROWS и т.д.;
- Каждый столбец в производной таблице *должен* иметь имя. Если этого нет по своей природе (например, потому что это — константа), то надо в обычном порядке присвоить псевдоним или добавить список псевдонимов столбцов в спецификации производной таблицы;
- Список псевдонимов столбцов опциональный, но если он присутствует, то должен быть полным (т.е. он должен содержать псевдоним для каждого столбца производной таблицы);
- Оптимизатор может обрабатывать производные таблицы очень эффективно. Однако, если производная таблица включена во внутреннее соединение и содержит подзапрос, то никакой порядок соединения не может быть использован оптимизатором.

FIRST и SKIP

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 1.5

Наилучшая альтернатива: ROWS

Описание: Указание FIRST ограничивает количество строк в выборке, а SKIP задаёт количество строк, пропускаемых в выборке.

Совет

В Firebird 2.0 и выше лучше использовать SQL-совместимый синтаксис ROWS.

Синтаксис:

SELECT [FIRST (<*INTEGER-expr*>)] [SKIP (<*INTEGER-expr*>)] <*columns*> FROM ...

<*INTEGER-expr*> ::= Любое выражение, приводимое к целому числу или просто целое число

<*columns*> ::= Обычные спецификации выбираемых в запросе столбцов

Примечание

Если <*INTEGER-expr*> целое число или параметр запроса, то можно не использовать скобки "()". С другой стороны, подзапросы требуют дополнительной пары круглых скобок.

Указания FIRST и SKIP являются опциональными. При совместном использовании “FIRST m SKIP n” в запросе первые **n** строк пропускаются и выбираются следующие **m** строк.

Указание SKIP 0 разрешено, но не имеет смысла. Начиная с Firebird 1.5 разрешено использовать указание FIRST 0, при этом возвращается пустой набор данных (в Firebird версии 1.0.x данное указание вызовет ошибку). Отрицательные числа в указаниях FIRST и SKIP всегда вызывают ошибку.

Если число в указании SKIP больше или равно количества записей в выборке, то возвращается пустой набор данных. Если число строк в выборке (или оставшихся после SKIP) меньше, чем значение, указанное для FIRST, то возвращается меньшее количество строк. Это нормальные результаты, а не ошибка.

Примеры:

Возврат первых десяти строк:

```
SELECT
    FIRST 10 ID, NAME
FROM PEOPLE
ORDER BY NAME ASC
```

Возврат всех записей, кроме первых 10:

```
SELECT
    SKIP 10 ID, NAME
FROM PEOPLE
ORDER BY NAME ASC
```

Возврат последних 10 строк. Обратите внимание на двойные круглые скобки:

```
SELECT
    SKIP ((SELECT COUNT(*) - 10 FROM PEOPLE)) ID, NAME
FROM PEOPLE
ORDER BY NAME ASC
```

Возврат строк с 81-й по 100-ю:

```
SELECT
    FIRST 20 SKIP 80 ID, NAME
FROM PEOPLE
ORDER BY NAME ASC
```

Два Глюка с FIRST в подзапросах

- Этот запрос

```
DELETE FROM MYTABLE
WHERE ID IN (SELECT FIRST 10 ID FROM MYTABLE)
```

удалит ВСЕ записи из таблицы.
Подзапрос каждый раз при удалении выбирает 10 строк, удаляет их — и так повторяется до тех пор, пока таблица не станет пустой. Знайте об этом! Или лучше: использовать указание ROWS, доступное начиная с Firebird 2.0.

- Такие запросы, как

```
...WHERE F1 IN (SELECT FIRST 5 F2 FROM TABLE2 ORDER BY 1 DESC)
```

не будут работать, как ожидалось, т.к. оптимизатор сервера преобразует предикт IN в предикт EXISTS, как показано ниже. Очевидно, что в этом случае использование указания FIRST N не имеет никакого смысла:

```
...WHERE EXISTS
        (SELECT FIRST 5 F2
           FROM TABLE2
           WHERE TABLE2.F2 = TABLE1.F2
          ORDER BY 1 DESC)
```

GROUP BY

Описание: GROUP BY объединяет строки, у которых есть та же комбинация значений, и/или NULL в списке элемента в единственную строку. Любые агрегатные функции в списке выбора применяются к каждой группе в отдельности, а не к набору данных в целом.

Синтаксис:

SELECT ... FROM ...
GROUP BY <*item*> [, <*item*> ...]
…

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

• Только не отрицательные литералы (числа) интерпретируются как позиция колонки. Если они находятся за пределами диапазона от 1 до количества столбцов, то возникает ошибка. Целочисленные значения, полученные из выражений или в качестве параметра являются просто постоянными величинами и будут использоваться как таковые в группировании. Они не будут иметь никакого эффекта, хотя их значение используется для каждой строки;
• Элемент GROUP BY не может ссылаться на агрегатную функцию (включая те, которые участвуют в выражении) из того же контекста;
• Список выборки не может содержать выражения, у которых могут быть различные значения в группе. Избежать этого можно, включая каждый не агрегатный столбец из списка выборки в указание GROUP BY (используя псевдонимы или позиции).

Примечание: Если группировка выполняется по позиции столбца, то выражение в этой позиции будет использоваться сервером из списка выборки. Если речь идет о подзапрос, то подзапрос будет выполняться по крайней мере два раза.

Группировка по псевдонимам, позиции и выражениям

Изменено: 1.0, 1.5, 2.0

Описание: В списке столбцов указания GROUP BY в дополнение к именам столбцов Firebird 2 позволяет использовать псевдонимы столбцов, позиции колонок и произвольные допустимые выражения.

Примеры:

Три запроса, выдающие одинаковый результат:

```
SELECT
    CHAR_LENGTH(LASTNAME) LEN_NAME, COUNT(*)
FROM PEOPLE
GROUP BY LEN_NAME

SELECT
    CHAR_LENGTH(LASTNAME) AS LEN_NAME, COUNT(*)
FROM PEOPLE
GROUP BY 1

SELECT
    CHAR_LENGTH(LASTNAME) AS LEN_NAME, COUNT(*)
FROM PEOPLE
GROUP BY CHAR_LENGTH(LASTNAME)
```

История: Группировка по результатам UDF была добавлена в Firebird 1.0. Группировка по позиции столбца, результатов CASE и ограниченного числа встроенных функций в Firebird 1.5. В Firebird 2 добавили для GROUP BY псевдонимы столбцов и выражения в целом ("выражения в целом" включает UDF, CASE и много встроенных функций).

HAVING: Более строгие правила

Изменено: 1.5

Описание: См. Агрегатные операторы: Более строгое использование HAVING и ORDER BY

JOIN

Запрещены неоднозначные имена столбцов

Изменено: 1.0

Описание: InterBase 6 принимает и выполняет операторы, как в примере ниже, в которых есть обращение к "неполному" (без алиаса или указания имени таблицы) имени столбца даже при том, что это имя существует в обеих таблицах, участвующих в соединении (JOIN):

```
SELECT
      BUSES.NAME, GARAGES.NAME
FROM BUSES JOIN
       GARAGES ON BUSES.GARAGE_ID = GARAGE.ID
WHERE NAME = 'Phideaux III'
```

Результаты выполнения такого запроса непредсказуемы. Диалект 3 в Firebird вызовет ошибку при наличии "неполных" имён полей в таких запросах. Диалект 1 выдаст предупреждение, но в любом случае выполнит запрос.

CROSS JOIN

Добавлено: 2.0

Описание: начиная с Firebird 2.0 поддерживается предложение , которое выполняет перекрёстное соединение (или декартово произведение) таблиц. Раньше это достигалось путём объединения на основе всегда выполняемого условия или с помощью неявного соединения (синтаксис объединения с запятой), в настоящее время осуждаемого.

Синтаксис:

```
SELECT ...
    FROM <relation> CROSS JOIN <relation>
...
<relation> ::= {table | view | cte | (select_stmt)} [[AS] alias]
```

Примечание: При использовании предложения CROSS JOIN Вы не должны использовать предикат ON.

Пример:

```
SELECT *
    FROM MEN CROSS JOIN WOMEN
ORDER BY MEN.AGE, WOMEN.AGE

-- Старый синтаксис:
--–   SELECT *
--          FROM MEN JOIN
--                  WOMEN ON 1 = 1
--    ORDER BY MEN.AGE, WOMEN.AGE

-- Неявное соединение:
--    SELECT *
--          FROM MEN, WOMEN
--    ORDER BY MEN.AGE, WOMEN.AGE
```

Именованные столбцы в JOIN

Добавлено: 2.1

Описание: Именованное соединение по столбцам — это объединение по эквивалентности на столбцах, названных в предложении USING. Эти столбцы должны присутствовать в обоих таблицах.

Синтаксис:

```
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]

Пример:

```
SELECT *
    FROM BOOKS JOIN SHELVES
        USING (SHELF, BOOKCASE)
```

Это эквивалентно традиционному соединению:

```
SELECT *
    FROM BOOKS B JOIN
        SHELVES S ON
            (B.SHELF = S.SHELF) AND
            (B.BOOKCASE = S.BOOKCASE)
```

Примечания:

- Столбцы в предложении USING могут быть выбраны без спецификаторов. Имейте, однако, в виду, что в такое внешнее соединение не всегда дает тот же результат, как при выборе left.colname или right.colname. Один из них может иметь значение NULL, в то время как другой нет; соединение по *colname* в таких случаях всегда возвращает not-NULL значения;
- SELECT * в соединениях по именованным столбцам возвращает каждый столбец из предложения USING только один раз. Во внешних соединениях такие столбцы всегда содержит not-NULL значения, за исключением строк, в которых столбец имеет значение NULL в обеих таблицах.

Естественный JOIN

Добавлено: 2.1

Описание: Естественное соединение это автоматическое объединение по эквивалентности всех столбцов, существующих в отношениях. Если нет общих названий столбцов, то производится перекрёстное соединение CROSS JOIN .

Синтаксис:

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

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

Пример:

```
SELECT *
    FROM PUPILS NATURAL LEFT JOIN TUTORS
```

Эквивалентный традиционный синтаксис в предположении, что таблицы PUPILS и TUTORS имеют два общих поля (TUTOR и CLASS), выглядит так:

```
SELECT *
    FROM PUPILS P LEFT JOIN
        TUTORS T ON
            (P.TUTOR = T.TUTOR) AND
            (P.CLASS = T.CLASS)
```

Примечания:

- Общие столбцы в естественном соединении могут быть выбраны без спецификаторов. Имейте, однако, в виду, что в такое внешнее соединение не всегда дает тот же результат, как при выборе left.colname или right.colname. Один из них может иметь значение NULL, в то время как другой нет; соединение по *colname* в таких случаях всегда возвращает not-NULL значения;
- SELECT * в естественных соединениях возвращает каждый столбец из предложения USING только один раз. Во внешних соединениях такие столбцы всегда содержит not-NULL значения, за исключением строк, в которых столбец имеет значение NULL в обеих таблицах.

ORDER BY

Синтаксис:

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}]

Сортировка по псевдонимам столбцов

Добавлено: 2.0

Описание: Начиная с Firebird 2.0 поддерживается сортировка по псевдонимам столбцов.

Пример:

```
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
```

Сортировка по номеру столбца вызывает расширение *

Изменено: 2.0

Описание: В случае сортировки по номеру столбца для запроса вида "SELECT " сервер теперь раскрывает звёздочку (*) для определения сортируемых столбцов.

Пример:

Такая сортировка была невозможна до версии Firebird 2.0:

```
SELECT *
    FROM RDB$COLLATIONS
ORDER BY 3, 2
```

В версиях Firebird до 2.0 сортировка в запросе, приведённом ниже, производилась по полю FILMS.DIRECTOR. Начиная с Firebird 2.0 сортировка происходит по второму столбцу таблицы BOOKS:

```
SELECT
    BOOKS.*,
    FILMS.DIRECTOR
FROM BOOKS, FILMS
```

```
ORDER BY 2
```

Сортировка по выражениям

Добавлено: 1.5

Описание: В Firebird 1.5 добавилась возможность использовать выражения для сортировки. Обратите внимание на то, что выражения, результатом вычисления которых должны быть целые неотрицательные числа, будут интерпретироваться как номер столбца и вызовут исключение, если они не будут в диапазоне от 1 до числа столбцов.

Пример:

```
SELECT
      X, Y, NOTE
FROM PAIRS
ORDER BY X+Y DESC
```

Примечания:

- Число, возвращаемое функцией или процедурой из UDF или хранимой процедуры, непредсказуемо, независимо от того, определена сортировка самим выражением или номером столбца;
- Только неотрицательные целые числа интерпретируются как номер столбца. Целое число, полученное *однократным* вычислением выражения или заменой параметра, запоминается как целочисленная постоянная величина, так как это значение одинаково для всех строк.

Размещение NULL

Изменено: 1.5, 2.0

Описание: В Firebird 1.5 ввели директивы NULLS FIRST и NULLS LAST для указания, где показывать значения NULL при сортировке столбца. В Firebird 2было изменено размещение значения NULL по умолчанию.

Если для сортировки не указаны директивы NULLS FIRST или NULLS LAST, то значения NULL в упорядоченных столбцов размещаются следующим образом:

- В Firebird 1.0 или 1.5 в конце независимо от порядка сортировки — по убыванию или по возрастанию;
- Начиная с Firebird 2.0 в *начале* при сортировке по возрастанию и в *конце* при сортировке по убыванию.

В таблице 7.1 приведены различия сортировок значения NULL в зависимости от версии сервера.

Таблица 7.1. Размещение NULL при сортировке столбцов

Сортировка	Размещение NULLs		
	Firebird 1	Firebird 1.5.x	Firebird 2.x
order by Field [asc]	внизу	внизу	вверху
order by Field desc	внизу	внизу	внизу
order by Field [asc | desc] nulls first	-	вверху	вверху
order by Field [asc | desc] nulls last	-	внизу	внизу

Примечания

• Корректная сортировка NULL в сервере Firebird 2.0 и выше для существующих баз данных достигается проведением цикла резервного копирования и восстановления (*Примечание переводчика: это необходимо сделать для смены ODS базы данных*);

• Индексы не будет использоваться для столбцов, у которых NULL не является значением по умолчанию. Это верно в случае сортировки: В Firebird 1.5 с NULLS FIRST; В версии Firebird 2.0 и выше с NULLS LAST по возрастанию и NULLS FIRST по убыванию.

Примеры:

```
SELECT *
      FROM MSG
ORDER BY PROCESS_TIME DESC NULLS FIRST

SELECT *
      FROM DOCUMENT
ORDER BY CHAR_LENGTH(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
```

Строгие правила сортировки для агрегатных операторов

Изменено: 1.5

Описание: См. Агрегатные операторы: Более строгое использование HAVING и ORDER BY

PLAN

Доступно: DSQL, ESQL, PSQL

Описание: Определяет пользовательский план выполнения запроса, переопределяя план, автоматически сгенерированный оптимизатором.

Синтаксис:

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* ...]

Улучшение обработки пользовательского плана

Изменено: 2.0

Описание: В Firebird 2 реализованы следующие улучшения обработки пользовательских планов:

- Фрагменты плана распространяются и на вложенные уровни соединений, что делает возможным ручную оптимизацию сложных внешних соединений;
- Пользовательские планы теперь проверяются на корректность внешних соединений;
- Добавлена упрощённая оптимизация пользовательских планов;
- Пользовательский план можно использовать для любого оператора или предложения, основанного на SELECT.

ORDER с INDEX

Изменено: 2.0

Описание: Одиночный элемент плана может теперь содержать директивы и ORDER, и INDEX (именно в таком порядке).

Пример:

```
PLAN (MYTABLE ORDER IX_MYFIELD INDEX (IX_THIS, IX_THAT))
```

PLAN должен включать все таблицы

Изменено: 2.0

Описание: Начиная с Firebird 2.0 в предложении PLAN должны быть включены все таблицы из запроса. Предыдущие версии иногда принимали неполные планы, но теперь это не разрешается.

Использование алиаса делает недоступным использование полного имени таблицы

Изменено: 2.0

Описание: Если в Firebird 2.0 или выше Вы дадите таблице или представлению псевдоним (алиас), то Вы должны везде использовать псевдоним, а не имя таблицы, если Вы хотите корректно определять имя столбца.

Пример:

Корректное использование:

```
SELECT PEARS
    FROM FRUIT

SELECT FRUIT.PEARS
    FROM FRUIT

SELECT PEARS
    FROM FRUIT F

SELECT F.PEARS
    FROM FRUIT F
```

Теперь невозможно:

```
SELECT FRUIT.PEARS
    FROM FRUIT F
```

ROWS

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Ограничивает количество строк, возвращенных оператором SELECT, на указанное число или диапазон.

Синтаксис:

Для простого SELECT:

```
SELECT <columns> FROM ...
    [WHERE ...]
    [ORDER BY ...]
    ROWS <m> [TO <n>]
```

<*columns*> ::= Обычная спецификация выходных столбцов
<*m*>, <*n*> ::= Любое выражение или число целого типа

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

<*m*>, <*n*> ::= Любые целые числа или выражение, приводимое к целому числу

Для запроса с 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>]
```

С единственным параметром **m** возвращаются первые **m** строк набора данных.

Необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то будет возвращён весь набор;
- Если **m = 0**, то будет возвращён пустой набор данных;
- Если **m < 0**, то оператор SELECT выдаст ошибку.

С двумя параметрами **m** и **n** выборка ограничивается строками начиная с **m** и до **n** включительно. Номера строк начинаются с 1.

Для двух аргументов необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то будет

возвращён пустой набор данных;

- Если число **m** не превышает общего количества строк в наборе данных, а **n** превышает, то то выборка ограничивается строками начиная с **m** до конца набора данных;
- Если **m < 1** и **n < 1**, то оператор SELECT выдаст ошибку;
- Если **n = m -1**, то будет возвращён пустой набор данных
- Если **n < m -1**, то оператор SELECT выдаст ошибку.

SQL-совместимый синтаксис ROWS устраняет необходимость использования предложений FIRST и SKIP, за исключением одного случая: при использовании предложения SKIP без FIRST, когда возвращаются все строки набора данных после пропуска заданного числа строк. (Хотя Вы можете обойти это и при использовании ROWS — подставив в качестве второго параметра число, превышающее общее количество строк в наборе данных).

Вы не можете использовать ROWS вместе с FIRST и/или SKIP в одном операторе SELECT, но разрешается использовать один синтаксис в запросе, а другой в подзапросе или в двух разных подзапросах.

При использовании UNION, предложение ROWS распространяется на весь UNION в целом и должен быть помещен после последнего оператора SELECT. Если надо ограничить выборку для одного или нескольких операторов SELECT, входящих в UNION, то у Вас есть два варианта: использовать FIRST / SKIP для этих операторов SELECT или конвертировать их в производные таблицы с предложением ROWS.

Предложение ROWS также можно использовать с операторами UPDATE и DELETE .

UNION

Доступно: DSQL, ESQL, PSQL

UNION в подзапросах

Изменено: 2.0

Описание: Разрешено использовать предложение UNION в подзапросах. Это применимо не только к подзапросам, возвращающем столбец для основного оператора SELECT, но также и к подзапросам в предикатах ANY|SOME, ALL и IN предикатах, а также опционально для оператора SELECT, используемого в INSERT.

Пример:

```
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

Добавлено: 2.0

Описание: Вы можете теперь использовать опциональное ключевое слово DISTINCT при определении UNION. С помощью него из выборки убираются дубликаты. Теперь DISTINCT, являясь противоположностью ALL, - режим по умолчанию; в любом случае, он не добавляет никакой новой функциональности.

Синтаксис:

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

Пример:

```
SELECT NAME, PHONE
        FROM TRANSLATORS
UNION DISTINCT
SELECT NAME, PHONE
        FROM PROOFREADERS
```

В данном запросе переводчики, работающие и в качестве корректоров, будут отображаться в выборке только один раз , если их номер телефона совпадает в обеих таблицах. Такой же результат был бы получен и без DISTINCT. При использовании ALL они появлялись бы два раза.

WITH LOCK

Доступно: DSQL, PSQL

Добавлено: 1.5

Описание: WITH LOCK обеспечивает возможность ограниченной явной пессимистической блокировки для осмотрительного использования в условиях:

a) когда затронут чрезвычайно малый набор строк (в идеале одна строка);
b) и для контроля кода приложения.

Только для экспертов!

Потребность в пессимистической блокировке в Firebird действительно очень редка и должна быть хорошо понята прежде, чем её использовать.

Важно понять эффекты уровней изоляции и других атрибутов транзакции прежде чем попытаться реализовать явную блокировку в приложении.

Синтаксис:

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

Если выборка с WITH LOCK прошла успешно, то запрос до завершения транзакции обеспечит блокировку выбранных строк и предотвратит любые другие попытки получения доступа на запись к любой из этих строк, или зависимых от них.

Если в запросе присутствует FOR UPDATE, то блокировка будет применена к каждой строке по отдельности, по мере выборки строки в серверный кэш данных. Это делает возможным то, что блокировка, которая, казалось, успешно выполнялась когда требовалось, однако перестанет работать впоследствии, когда будет предпринята попытка выбрать строку, которая стала заблокированной другой транзакцией.

Предложение WITH LOCK можно использовать только для оператора SELECT верхнего уровня и для выборки данных только из одной таблицы.

Блокировка недоступна:

- в подзапросах;
- для соединения таблиц;
- при использовании оператора DISTINCT, предложения GROUP BY и любых других агрегатных операций;
- для представлений;
- для селективных хранимых процедур;
- для внешних таблиц.

Более подробное обсуждение блокировки "SELECT ... WITH LOCK" Вы можете прочитать в разделе Примечания. Это нужно прочитать всем, кто хочет пользоваться этой функцией.

UPDATE

Доступно: DSQL, ESQL, PSQL

Описание: Изменяет значения записей в таблице (или в одной или нескольких базовых таблицах представления). Обновляемые столбцы и их новое значение перечисляют в предложении SET. Количество обновляемых строк можно ограничить в условии WHERE или предложением ROWS.

Синтаксис:

```
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*> ::= Любые целые числа или выражение, приводимое к целому числу

<*values*> ::= *value_expression* [, *value_expression* ...]

<*variables*> ::= :*varname* [, :*varname* ...]

Ограничения

- Директива TRANSACTION доступна только в ESQL;
- В чистом сеансе DSQL предложение WHERE CURRENT OF не имеет смысла, так как в DSQL не существует операторов для создания курсора;

- Предложения PLAN, ORDER BY, ROWS и RETURNS не доступны в ESQL;
- Начиная с версии Firebird 2.0 никакой столбец не может упоминаться несколько раз в предложении SET оператора UPDATE;
- Использование предложения "INTO *<variables>*" разрешается только в PSQL;
- При возврате значения в контекстную переменную NEW этому имени не должно предшествовать двоеточие (“:”)

Изменение семантики SET

Изменено: 2.5

Описание: В предыдущих версиях Firebird, если было сделано несколько присвоений в предложении SET, новые значения столбцов становились сразу же доступными для последующих присвоений в этом же операторе. Например, в таком присвоении “set a=3, b=a” значение b будет установлено равным 3, а не старому значению a. Это нестандартное поведение в настоящее время исправлено. Начиная с Firebird 2.5 любые присвоения в SET будут использовать старые значения столбцов.

Пример:

Возьмём таблицу TEST:

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

Выполнение следующего оператора:

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

приведёт к следующим изменениям данных в таблице:

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

а в версиях Firebird до 2.5 этот же оператор изменит данные в таблице следующим образом:

```
A       B
----------
5       5
5       5
```

Сохранение старого поведения: В течение ограниченного времени Вы можете сохранить старое, нестандартное поведение, установив параметр OldSetClauseSemantics в firebird.conf равным 1. Если этот параметр установлен в 1, то будет использоваться для всех соединений с базой данных. Этот параметр будет запрещён и удален в будущем.

Использование COLLATE для столбцов с текстовым BLOB

Добавлено: 2.0

Описание: Предложение COLLATE доступно теперь и для столбцов текстовых BLOB.

Пример:

```
UPDATE MYTABLE
        SET NAMEBLOB_SP = 'Juan'
WHERE NAMEBLOB_BR COLLATE PT_BR = 'Joậo'
```

ORDER BY

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Разрешено использование предложения ORDER BY в операторе UPDATE. Это целесообразно только в сочетании с предложением ROWS, но допустимо также и без него.

PLAN

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Оператор UPDATE теперь поддерживает предложение PLAN, т.е. пользователи могут оптимизировать выполнение обновления записей вручную.

Использование алиаса делает недоступным использование полного имени таблицы

Изменено: 2.0

Описание: Если в Firebird 2.0 или выше Вы дадите таблице или представлению псевдоним (алиас), то Вы должны везде использовать псевдоним, а не имя таблицы, если Вы хотите корректно определять имя столбца.

Пример:

Корректное использование:

```
UPDATE FRUIT
    SET SORT = 'Антоновка'
WHERE …

UPDATE FRUIT
    SET FRUIT.SORT = 'Антоновка'
WHERE …

UPDATE FRUIT F
    SET SORT = 'Антоновка'
WHERE …

UPDATE FRUIT F
    SET F.SORT = 'Антоновка'
WHERE ...
```

Теперь запрещено:

```
UPDATE FRUIT F
    SET FRUIT.SORT = 'Антоновка'
WHERE ...
```

RETURNING

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Оператор UPDATE, обновляющий *только* одну строку, может дополнительно включать предложение RETURNING для возврата значений обновляемой строки. Предложение, если присутствует, не должно содержать все обновлённые столбцы, а также может содержать другие столбцы или выражения. Возвращаемые значения учитывают все изменения, сделанные в триггере до вставки (BEFORE INSERT), но не учитывают изменения, сделанные в триггере после вставки (AFTER INSERT). И старое (OLD.fieldname), и новое (NEW.fieldname) значения могут быть использованы в списке возвращаемых столбцов. Если имени столбца не предшествует префикс OLD или NEW, то возвращается новое значение.

Пример:

```
UPDATE SCHOLARS
        SET   FIRSTNAME = 'Олег',
              LASTNAME = 'Петров'
WHERE FIRSTNAME = 'Олежка' AND LASTNAME = 'Перов'
RETURNING ID, OLD.LASTNAME, NEW.LASTNAME
```

Примечания:

- В DSQL оператор с предложением RETURNING *всегда* возвращает одну строку. Если обновления записи не было, то все поля возвращаются со значением NULL. Такое поведение может измениться в более поздней версии Firebird. В PSQL, если ни одна строка не была обновлена, то ничего не возвращается, и значения переменных сохраняют существующие значения.

ROWS

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Ограничивает количество обновляемых строк на указанное число или диапазон.

Синтаксис:

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

<*m*>, <*n*> ::= Любые целые числа или выражения, приводимые к целому числу

С единственным параметром **m** обновление ограничено первыми **m** строками набора данных, определенного таблицей или представлением и дополнительными условиями в предложениях WHERE и ORDER BY.

Необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то будет обновлён весь набор данных;
- Если **m = 0**, то ничего не будет обновлено;
- Если **m < 0**, то оператор обновления выдаст ошибку.

С двумя параметрами **m** и **n** обновление ограничено строками начиная с **m** и до **n** включительно. Номера строк начинаются с 1.

Для двух аргументов необходимо отметить следующее:

- Если **m** больше общего количества строк в наборе данных, то ничего не будет обновлено;
- Если число **m** не превышает общего количества строк в наборе данных, а **n** превышает, то будут обновлены строки начиная с **m** до конца набора данных;
- Если **m < 1** и **n < 1**, то оператор обновления выдаст ошибку;
- Если **n = m -1**, то ничего не будет обновлено;
- Если **n < m -1**, то оператор обновления выдаст ошибку.

Предложение ROWS также можно использовать с операторами SELECT и DELETE .

UPDATE OR INSERT

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Оператор UPDATE OR INSERT проверяет, есть ли существующие

записи, уже содержащие новые значения столбцов, которые перечислены в предложении MATCHING. Если существуют, то записи обновляются. Если же не существуют, то записи вставляются. При отсутствии предложения MATCHING сопоставление выполняется по первичному ключу. Если присутствует предложение RETURNING и найдена более, чем одна соответствующая запись, то оператор UPDATE OR INSERT вернёт ошибку.

Синтаксис:

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

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

Ограничения

- Столбец может присутствовать в списке вставки/обновления только один раз;
- Если в таблице нет первичного ключа, то предложение MATCHING является обязательным;
- Предложение “INTO *<variables>*” доступно только в PSQL;
- Когда значения возвращаются в контекстную переменную NEW, то ей не должно предшествовать двоеточие (":").

Пример:

```
UPDATE OR INSERT INTO COWS (
    NAME, NUMBER, LOCATION)
VALUES (
    'Олег Петров', 3278823, 'Владивосток')
MATCHING (NUMBER)
RETURNING REC_ID INTO :ID;
```

Примечания:

- Соответствия определяются с использованием IS NOT DISTINCT, а не оператора равно (“=”). Это означает, что одно значение NULL соответствует одно другому;

- Опциональное предложение RETURNING означает что оно:
  - ...может содержать любые столбцы обновляемой таблицы, независимо от того, были они упомянуты ранее в операторе, а также и другие выражения;
  - ...может содержать префиксы OLD и NEW для имен полей; по умолчанию возвращается новое значение поля;
- ...возвращает значения полей после срабатывания триггеров до вставки или обновления (BEFORE), но не после вставки или обновления (AFTER).

Глава 8

Управление транзакциями

RELEASE SAVEPOINT

Доступно: DSQL

Добавлено: 1.5

Описание: Удаляет именованную точку сохранения, освобождая все связанные с ней ресурсы.

Синтаксис:

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

Без использования предложения ONLY удаляются также все точки сохранения, создаваемые после указанной.

Более подробное описание точек сохранения приведено в разделе SAVEPOINT .

ROLLBACK

Доступно: DSQL, ESQL

Синтаксис:

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

- Предложение TRANSACTION доступно только в ESQL;
- Предложение RELEASE доступно только в ESQL и не рекомендуется для использования;
- Предложения RETAIN и TO доступны только в DSQL.

ROLLBACK RETAIN

Доступно: DSQL

Добавлено: 2.0

Описание: Отменяет все изменения в базе данных, выполненные в рамках транзакции, при этом не закрывая её. Пользовательские переменные, заданные с помощью функции RDB$SET_CONTEXT () остаются неизменными.

Синтаксис:

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

Примечание

Функциональные возможности, предоставляемые ROLLBACK RETAIN, присутствуют начиная с InterBase 6, но единственным способом получить доступ к ним был через вызов API функции isc_rollback_retaining ().

ROLLBACK TO SAVEPOINT

Доступно: DSQL

Добавлено: 1.5

Описание: Отменяет все изменения, произошедшие в рамках транзакции, начиная с созданной точки сохранения (SAVEPOINT).

Синтаксис:

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

ROLLBACK TO SAVEPOINT выполняет следующие операции:

- Все изменения в базе данных, выполненные в рамках транзакции начиная с созданной точки сохранения, отменяются. Пользовательские переменные, заданные с помощью функции RDB$SET_CONTEXT () остаются неизменными;
- Все точки сохранения, создаваемые после названной, уничтожаются. Все

более ранние точки сохранения, как сама точка сохранения, остаются. Это означает, что можно откатываться к той же точке сохранения несколько раз;

- Все явные и неявные блокированные записи, начиная с точки сохранения, освобождаются. Другие транзакции, запросившие ранее доступ к строкам, заблокированным после точки сохранения, должны продолжать ожидать, пока транзакция не фиксируется или откатывается. Другие транзакции, которые ещё не запрашивали доступ к этим строкам, могут запросить и сразу же получить доступ к разблокированным строкам.

Более подробное описание точек сохранения приведено в разделе SAVEPOINT .

SAVEPOINT

Доступно: DSQL

Добавлено: 1.5

Описание: Создает SQL 99 совместимую точку сохранения, к которой можно позже откатывать работу с базой данных, не отменяя все действия, выполненные с момента старта транзакции. Механизмы точки сохранения также известны под термином "вложенные транзакции" (“nested transactions”).

Синтаксис:

SAVEPOINT <*name*>

<*name*> ::= выбранный пользователем идентификатор для точки сохранения, уникальный для стартовавшей транзакции

Если имя точки сохранения уже существует в рамках транзакции, то существующая точка сохранения будет удалена, и создается новая с тем же именем.

Для отката изменений к точке сохранения используйте оператор:

ROLLBACK [WORK] TO [SAVEPOINT] *name*

ROLLBACK TO SAVEPOINT выполняет следующие операции:

- Все изменения в базе данных, выполненные в рамках транзакции начиная с созданной точки сохранения, отменяются. Пользовательские переменные, заданные с помощью функции RDB$SET_CONTEXT () остаются неизменными;

- Все точки сохранения, создаваемые после названной, уничтожаются. Все более ранние точки сохранения, как сама точка сохранения, остаются. Это означает, что можно откатываться к той же точке сохранения несколько раз;
- Все явные и неявные блокированные записи, начиная с точки сохранения, освобождаются. Другие транзакции, запросившие ранее доступ к строкам, заблокированным после точки сохранения, должны продолжать ожидать, пока транзакция не фиксируется или откатывается. Другие транзакции, которые ещё не запрашивали доступ к этим строкам, могут запросить и сразу же получить доступ к разблокированным строкам.

Внутренний механизм точек сохранения может использовать большие объемы памяти, особенно если Вы обновляете одни и те же записи многократно в одной транзакции. Если точка сохранения уже не нужна, но Вы еще не готовы закончить транзакцию, то можно её удалить, тем самым освобождая ресурсы:

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

Без использования предложения ONLY удаляются также все точки сохранения, создаваемые после указанной.

Пример DSQL сессии с использованием точек сохранения:

```
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; -- возвращает пустую строку
ROLLBACK TO Y;
SELECT * FROM TEST; -- возвращает две строки
ROLLBACK;
SELECT * FROM TEST; -- возвращает одну строку
```

Внутренние точки сохранения

По умолчанию сервер использует автоматическую системную точку сохранения уровня транзакции для выполнения её отката. При выполнении оператора ROLLBACK, все изменения, выполненные в транзакции, откатываются до системной точки сохранения и после этого транзакция подтверждается.

Когда объем изменений, выполняемых под системной точкой сохранения

уровня транзакции, становится большим (затрагивается порядка 10^4-10^6 записей) сервер освобождает системную точку сохранения и, при необходимости отката транзакции, использует механизм TIP.

Совет

Если Вы ожидаете, что объем изменений в транзакции будет большим, то можно задать опцию NO AUTO UNDO в операторе SET TRANSACTION, или – если используется API – установить флаг TPB isc_tpb_no_auto_undo. В обеих вариантах предотвращается создание системной точки сохранения уровня транзакции.

Точки сохранения и PSQL

Использование операторов управления транзакциями в PSQL не разрешается , так как это нарушит атомарность оператора, вызывающего процедуру. Но Firebird поддерживает вызов и обработку исключений в PSQL, так, чтобы действия, выполняемые в хранимых процедурах и триггерах, могли быть выборочно отменены без полного отката всех действий в них. Внутренне автоматические точки сохранения используется для:

- отмены всех действия внутри блока BEGIN ... END, где происходит исключение;
- отмены всех действия, выполняемых в SP/триггере (или, в случае селективной SP, всех действий, выполненных с момента последнего оператора SUSPEND), если они завершаются преждевременно из-за непредусмотренной ошибки или исключения.

Каждый блок обработки исключений PSQL также ограничен автоматическими точками сохранения сервера.

SET TRANSACTION

Доступно: DSQL, ESQL

Изменено: 2.0

Описание: Задаёт параметры транзакции и стартует её.

Синтаксис:

```
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 ...]
```

- Опция NAME доступна только в ESQL. При этом должна быть объявлена и инициализирована переменная базового языка. Без опции NAME оператор SET TRANSACTION применяется к транзакции по умолчанию;
- Опция USING также доступна только в ESQL. Она задаёт базы данных, к которым транзакция может получить доступ;
- Опции IGNORE LIMBO и LOCK TIMEOUT не доступны в ESQL;
- Опции IGNORE LIMBO и NO WAIT являются взаимоисключающими;
- Опции по умолчанию для транзакции: READ WRITE + WAIT + SNAPSHOT.

IGNORE LIMBO

Доступно: DSQL

Добавлено: 2.0

Описание: С этой опцией игнорируются записи, создаваемые “потерянными” (т.е. не завершёнными) транзакциями (limbo transaction). Транзакции считается “потерянной”, если не завершён второй этап двухфазного подтверждения (two-phase commit).

Примечание

Параметр IGNORE LIMBO аналогичен параметру isc_tpb_ignore_limbo TPB, доступного в API со времен InterBase - в основном используются утилитой

командной строки gfix.

LOCK TIMEOUT

Доступно: DSQL

Добавлено: 2.0

Описание: Эта опция доступна только вместе при использовании WAIT. Она принимает неотрицательное целое число в качестве параметра, задавая максимальное количество секунд, в течении которых транзакция должна ожидать при возникновении конфликта блокировки. Если время ожидания прошло, а блокировка ещё не снята, то выдается сообщение об ошибке.

Примечание

Это совершенно новая опция, добавленная в Firebird 2. Её эквивалент в API - новый параметр TPB isc_tpb_lock_timeout.

NO AUTO UNDO

Доступно: DSQL, ESQL

Добавлено: 2.0

Описание: При использовании опции NO AUTO UNDO не ведётся журнал, используемый для отмены изменений в случае отката транзакции. При откате транзакции в конечном счете сборка мусора выполнится в рамках других транзакций. Эта опция может быть полезна при выполнении операций массовых вставок, которые не должны откатываться. Для транзакций, в рамках которых не выполняется никаких изменений, опция NO AUTO UNDO игнорируется

Примечание

Опция NO AUTO UNDO имеет свой аналог в API - параметр TPB isc_tpb_no_auto_undo, доступный начиная с InterBase.

Глава 9

Операторы PSQL

PSQL - процедурный SQL – это язык программирования Firebird, используемый в хранимых процедурах, триггерах и выполнимых блоках.

Блок BEGIN ... END может быть пустым

Доступно: PSQL

Изменено: 1.5

Описание: Начиная с Firebird 1.5 блок BEGIN...END может быть пустым, т.о. создаётся своеобразная «заглушка», позволяющая избежать написания фиктивных операторов.

Пример:

```
CREATE TRIGGER BI_ATABLE FOR ATABLE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
END
```

BREAK

Доступно: PSQL

Добавлено: 1.0

Наилучшая альтернатива: LEAVE

Описание: Оператор BREAK немедленно завершает цикл WHILE или FOR и продолжает выполнение кода с первого оператора после цикла.

Пример:

```
CREATE PROCEDURE SELPHRASE(NUM INTEGER)
RETURNS (NN INTEGER, PHRASE VARCHAR(40))
AS
BEGIN
        NN = 0;
        FOR SELECT PHR
                FROM PHRASES
                ORDER BY 1
        INTO :PHRASE DO
                BEGIN
                        NN = : NN + 1;
                        IF (:NUM < :NN) THEN
                                BREAK;
                        SUSPEND;
                END
        PHRASE = '*** Готово! ***';
        SUSPEND;
END
```

Хранимая процедура возвращает первые *NUM* строк из таблицы PHRASES, пронумерованные от 1 до *NUM* в порядке возрастания. Переменная *NN* задаёт порядковый номер возвращаемой записи; по её же значению происходит проверка условия выхода из цикла выборки записей. Как только значение входного параметра *NUM* станет меньше, чем значение переменной *NN*, цикл выборки прерывается оператором BREAK. Сразу же после прерывания цикла хранимая процедура переходит к выполнению первого после цикла оператора - «PHRASE = '*** Готово! ***';».

Важно

Начиная с Firebird 1.5 предпочтительнее использовать SQL-99 совместимый оператор LEAVE .

CLOSE CURSOR

Доступно: PSQL

Добавлено: 2.0

Описание: Закрывает открытый курсор. Любые все ещё открытые курсоры

будут автоматически закрыты после выполнения кода триггера, хранимой процедуры или выполнимого блока, в пределах кода которого он был открыт.

Синтаксис:

CLOSE *cursorname*;

Пример: См. в разделе DECLARE ... CURSOR

DECLARE

Доступно: PSQL

Описание: Служит для объявления локальной переменной в PSQL.

Синтаксис:

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

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

- Если переменная тестового типа (*sql_datatype*), то при её объявлении можно задать её набор символов;
- Очевидно, что предложение COLLATE допускается только для текстовых типов переменных.

DECLARE ... CURSOR

Добавлено: 2.0

Описание: Объявляет именованный курсор и связывает его с собственным оператором SELECT. В дальнейшем курсор может быть открыт, использоваться для обхода результирующего набора данных, и снова быть закрытым. Также поддерживаются позиционированные обновления и удаления (при использовании WHERE CURRENT OF). Курсоры в PSQL доступны в триггерах, хранимых процедурах и выполнимых блоках (EXECUTE BLOCK).

Пример:

```
EXECUTE BLOCK
RETURNS (RELATION CHAR(31), SYSFLAG INTEGER)
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
```

Примечания:

- Предложение "FOR UPDATE" разрешено использовать в операторе SELECT, но оно не требуется для успешного выполнения позиционированного обновления или удаления;
- Удостоверьтесь, что объявленные имена курсоров не совпадают ни с какими именами, определенными позже в предложениях AS CURSOR;
- Если курсор требуется только для прохода по результирующему набору данных, то практически всегда проще (и менее подвержено ошибкам) использовать оператор FOR SELECT с предложением AS CURSOR. Объявленные курсоры должны быть явно открыты, использованы для выборки данных и закрыты. Кроме того, Вы должны проверить контекстную переменную row_count после каждой выборки и выйти из цикла, если её значение ноль. Предложение AS CURSOR делает эту проверку автоматически. Однако, объявленные курсоры дают большие возможности для контроля за последовательными событиями и позволяют управлять несколькими курсорами параллельно;
- Оператор SELECT может содержать параметры, например: “SELECT NAME || :SFX FROM NAMES WHERE NUMBER = :NUM”. Каждый параметр должен быть заранее объявлен как переменная PSQL (это касается также входных и выходных параметров). При открытии курсора параметру

присваивается текущее значение переменной;

- Внимание! Если значение переменной PSQL, используемой в операторе SELECT, изменяется во время выполнения цикла, то её новое значение может (но не всегда) использоваться при выборке следующих строк. Лучше избегать таких ситуаций. Если Вам действительно требуется такое поведение, то необходимо тщательно протестировать код и убедиться, что Вы точно знаете, как изменения переменной влияют на результаты выборки. Особо отмечу, что поведение может зависеть от плана запроса, в частности, от используемых индексов. В настоящее время нет строгих правил для таких ситуаций, но в новых версиях Firebird это может измениться.

A *См. также*: OPEN CURSOR , FETCH CURSOR и CLOSE CURSOR .

DECLARE [VARIABLE] с инициализацией

Изменено: 1.5

Описание: Начиная с Firebird 1.5 можно инициализировать переменные при их объявлении. Ключевое слово VARIABLE стало опциональным.

Пример:

```
CREATE PROCEDURE PROCCIE (A INTEGER)
RETURNS (B INTEGER)
AS
        DECLARE P INTEGER;
        DECLARE Q INTEGER = 8;
        DECLARE R INTEGER DEFAULT 9;
        DECLARE VARIABLE S INTEGER;
        DECLARE VARIABLE T INTEGER = 10;
        DECLARE VARIABLE U INTEGER DEFAULT 11;
BEGIN
        …
END
```

DECLARE с DOMAIN вместо типа данных

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование доменов вместо типов данных SQL при объявлении входных и выходных параметров и

локальных переменных. Если перед названием домена дополнительно используется предложение "TYPE OF", то используется только тип данных домена — не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию. Если домен текстового типа, то всегда используется его набор символов и порядок сортировки.

Пример:

```
CREATE PROCEDURE MYPROC (A INTEGER, F TERNBOOL)
RETURNS (B INTEGER, X TYPE OF BIGFLOAT)
AS
    DECLARE P INTEGER;
    DECLARE Q INTEGER = 8;
    DECLARE Y STOCKNUM DEFAULT -1;
BEGIN
    <текст кода>
END
```

Этот пример предполагает, что домены TERNBOOL, BIGFLOAT и STOCKNUM уже определены в базе данных.

Предупреждение

Если тип домена позднее изменяется, PSQL код с использованием этого домена может вызывать ошибки. Информация о том, как это обнаружить, находится в Приложении A (раздел Поле RDB$VALID_BLR).

TYPE OF COLUMN в объявлении переменных и параметров

Добавлено: 2.5

Описание: По аналогии с синтаксисом «TYPE OF *domain*» , поддерживаемым начиная с версии 2.1, теперь также можно объявлять переменные и параметры, используя тип данных столбцов существующих таблиц и представлений. Используется только тип данных, а в случае строковых типов ещё и набор символов и параметры сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Пример:

```
CREATE TABLE CARS (
    MAKE VARCHAR(20),
```

```
    MODEL VARCHAR(20),
    WEIGHT NUMERIC(4),
    TOPSPEED NUMERIC(3),
CONSTRAINT UK_MAKE_MODEL UNIQUE (MAKE, MODEL)
)

CREATE PROCEDURE MAX_KINETIC_ENERGY(
    MAKE TYPE OF COLUMN CARS.MAKE,
    MODEL TYPE OF COLUMN CARS.MODEL)
RETURNS (MAX_E_KIN DOUBLE PRECISION)
AS
    DECLARE MASS TYPE OF COLUMN CARS.WEIGHT;
    DECLARE VELOCITY TYPE OF COLUMN CARS.TOPSPEED;
BEGIN
    SELECT WEIGHT, TOPSPEED
        FROM CARS
        WHERE MAKE = :MAKE AND MODEL = :MODEL
    INTO :MASS, :VELOCITY;
    MAX_E_KIN = 0.5 * :MASS * :VELOCITY * :VELOCITY;
END
```

Предупреждения

- Для текстовых типов в TYPE OF COLUMN используются набор символов и порядок сортировки — так же, как и при использовании [TYPE OF] *<domain>*. Однако, из-за ошибки, сортировки не всегда принимаются во внимание при выполнении операции сравнения (например, на равенство). В случаях, когда сортировка имеет важное значение, необходимо тщательно тестировать свой код перед использованием! Эта ошибка исправлена в Firebird 3;
- Если тип столбца позднее изменяется, PSQL код с использованием этого столбца может вызывать ошибки. Информация о том, как это обнаружить, находится в Приложении A (раздел Поле RDB$VALID_BLR).

COLLATE в объявлении переменных

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование предложения COLLATE при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
CREATE PROCEDURE GIMMETEXT
    RETURNS (TXT CHAR(32) CHARACTER SET WIN1251
                COLLATE WIN1251_UA)
AS
    DECLARE SIMOUNAO MYTEXTDOMAIN COLLATE PT_BR
        DEFAULT 'NAO';
BEGIN
    <текст кода>
END
```

NOT NULL при объявлении переменных и параметров

Добавлено: 2.1

Описание: Начиная с Firebird 2.1 поддерживается использование NOT NULL ограничения при объявлении входных и выходных параметров и локальных переменных.

Пример:

```
CREATE PROCEDURE COMPUTE(
    A INTEGER NOT NULL,
    B INTEGER NOT NULL)
RETURNS (
    OUTCOME BIGINT NOT NULL)
AS
    DECLARE TEMP BIGINT NOT NULL;
BEGIN
    <текст кода>
END
```

EXCEPTION

Доступно: PSQL

Изменено: 1.5

Описание: Синтаксис оператора EXCEPTION был для того, чтобы пользователь мог:

- Повторно вызвать пойманное исключение или ошибку;
- Выдать пользовательское сообщение при срабатывании определённого пользователем исключения.

Синтаксис:

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

<*exception-name*> ::= имя заранее созданного исключения
[*custom-message*] ::= текст, выдаваемый при вызове исключения (опционально)

Повторный вызов перехваченного исключения

В блоке обработки исключений (только в нём), Вы можете повторно вызвать пойманное исключение или ошибку, вызывая оператор EXCEPTION без параметров. Вне блока с исключением такой вызов не имеет никакого эффекта.

Пример:

```
WHEN ANY DO
    BEGIN
        INSERT INTO ERROR_LOG (…)
            VALUES (SQLCODE, ...);
        EXCEPTION;
    END
```

В этом примере сначала регистрируется некоторая информация об исключении или ошибке и затем повторно вызывается исключение.

Поддержка пользовательского сообщения об ошибке

Начиная с 1.5 Вы можете переопределить сообщение об ошибке по умолчанию при вызове исключения, заменив его альтернативным при выдаче исключения.

Примеры:

```
EXCEPTION EX_DATA_ERROR
    'Текст сообщения, выдаваемый при вызове исключения';
```

```
EXCEPTION EX_BAD_TYPE
    'Неверный тип данных для записи с ID ' || NEW.ID;
```

Примечание

Начиная с Firebird 2.0 максимальная длина сообщения исключения увеличена с 78 до 1021 байтов.

EXECUTE PROCEDURE

Доступно: DSQL, PSQL

Изменено: 1.5

Описание: Начиная с Firebird 1.5 в качестве входных параметров для процедур, выполняемых с помощью оператора EXECUTE PROCEDURE, разрешено использовать выражения. Полное описание и примеры приведены в разделе EXECUTE PROCEDURE.

EXECUTE STATEMENT

Доступно: PSQL

Добавлено: 1.5

Изменено: 2.5

Описание: Оператор EXECUTE STATEMENT принимает строковый параметр и выполняет его, как будто это оператор DSQL. Если оператор возвращает данные, то с помощью предложения INTO их можно передать в локальные переменные. Если оператор возвращает более одной строки данных, то для создания цикла выборки нужно использовать конструкцию “FOR ... DO”.

Синтаксис (полный):

```
<execute-statement> ::= EXECUTE STATEMENT <argument>
    [<option> ...]
    [INTO <variables>]

<looped-version> ::= FOR <execute-statement> DO <psql-statement>
```

<*argument*> ::= *paramless-stmt*
| (*paramless-stmt*)
| (<*stmt-with-params*>) (<*param-values*>)

<*stmt-with-params*> ::= Оператор, содержащий один или несколько параметров, в одной из этих конструкций:

- именованный: ':' + paramname, т.е. :a, :b, :size
- позиционный: каждый параметр представлен в виде '?'

Не разрешается одновременно использовать именованные и позиционные параметры

<*param-values*> ::= <*named-values*> | <*positional-values*>
<*named-values*> ::= *paramname* := *value-expr* [, *paramname* := *value-expr* ...]
<*positional-values*> ::= *value-expr* [, *value-expr* ...]

option> ::= WITH {AUTONOMOUS|COMMON} TRANSACTION
| WITH CALLER PRIVILEGES
| AS USER *user*
| PASSWORD *password*
| ROLE *role*
| ON EXTERNAL [DATA SOURCE] <*connect-string*>

<*connect-string*> ::= [<*hostspec*>] *путь к БД или алиас*
<*hostspec*> ::= <*tcpip-hostspec*> | <*netbeui-hostspec*>
<*tcpip-hostspec*> ::= *hostname*:
<*netbeui-hostspec*> ::= *hostname*\

<*variables*> ::= [:]*varname* [, [:]*varname* ...]

<*psql-statement*> ::= Простые или сложные операторы PSQL

ВНИМАНИЕ:
paramless-stmt, <stmt-with-params>, user, password, role и <connect-string> - строковые выражения. При явном задании этих аргументов (т.е. в виде символьных строк) их необходимо заключать в одинарные кавычки.

Ниже сначала приведены основы использования оператор EXECUTE STATEMENT (как это было реализовано в Firebird 1.5). После этого представлены новые возможности, добавленные в Firebird 2.5.

Без возврата данных

Эта конструкция используется для выполнения операторов INSERT, UPDATE, DELETE и EXECUTE PROCEDURE, не возвращающих данные.

Синтаксис (частичный):

EXECUTE STATEMENT <*statement*>

<*statement*> ::= Оператор SQL, не возвращающий данных

Пример:

```
CREATE PROCEDURE DYNAMICSAMPLEONE(
        PROCNAME VARCHAR(100))
AS
        DECLARE VARIABLE STMT VARCHAR(1024);
        DECLARE VARIABLE PARAM INTEGER;
BEGIN
        SELECT MIN(SOMEFIELD)
                FROM SOMETABLE
        INTO :PARAM;
        STMT = 'EXECUTE PROCEDURE '
                || PROCNAME
                || '('
                || CAST(PARAM AS VARCHAR(20))
                || ')';
        EXECUTE STATEMENT STMT;
END
```

Предупреждение

Хотя эта оператор EXECUTE STATEMENT также может быть использован со всеми видами операторов DDL (за исключением CREATE/DROP DATABASE), обычно очень неразумно *(примечание переводсика: настоятельно не рекомендуется)* использовать этот прием для обхода запрета использования операторов DDL в PSQL.

Возврат одной строки данных

Используется в одиночных (singleton - т.е. возвращающем одну запись) операторах SELECT.

Синтаксис (частичный):

EXECUTE STATEMENT <*select-statement*> INTO <*var*> [, <*var*> …]

<*select-statement*> ::= Оператор SQL, возвращающий только одну строку данных
<*var*> ::= Переменная PSQL, опционально с двоеточием перед ней

Пример:

```
CREATE PROCEDURE DYNAMICSAMPLETWO(
      TABLENAME VARCHAR(100))
AS
      DECLARE VARIABLE PARAM INTEGER;
BEGIN
      EXECUTE STATEMENT
            ' SELECT MAX(CHECKFIELD) ' ||
            ' FROM ' || TABLENAME
      INTO :PARAM;
      IF (PARAM > 100) THEN
            EXCEPTION EX_OVERFLOW
                  'OVERFLOW IN ' || :TABLENAME;
END
```

Возврат любого количества строк данных

Используется (по аналогии конструкции “FOR SELECT ... DO”) для операторов SELECT, возвращающих более одной строки.

Синтаксис (частичный):

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

<*select-statement*> ::= Любой оператор SELECT
<*var*> ::= Переменная PSQL, опционально с двоеточием перед ней

<*psql-statement*> ::= Простой или сложный оператор PSQL

Пример:

```
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
```

Улучшенная производительность

Изменено: 2.5

Описание: В предыдущих версиях (до Firebird 2.5) при выполнении оператора EXECUTE STATEMENT в цикле SQL-оператор подготавливается, а затем выполняется, для каждой итерации. Начиная с Firebird 2.5 такой оператор подготавливается к выполнению только один раз, что даёт огромный выигрыш в производительности.

WITH {AUTONOMOUS|COMMON} TRANSACTION

Добавлено: 2.5

Описание: Традиционно, выполнение оператора SQL всегда происходит в текущей транзакции, и это до сих пор используется по умолчанию. При использовании предложения WITH AUTONOMOUS TRANSACTION запускается новая транзакция с такими же параметрами, как и текущая. Она будет подтверждена, если оператор выполнился без ошибок и отменена (откачена) в противном случае. С предложением WITH COMMON TRANSACTION по

возможности используется текущая транзакция. Если оператор должен работать в отдельном соединении, то используется уже запущенная в этом соединении транзакция (если таковая транзакция имеется). В противном случае стартует новая транзакция с параметрами текущей транзакции. Любые новые транзакции, запущенные в режиме "COMMON", подтверждаются или откатываются вместе с текущей транзакцией.

Синтаксис (частичный):

```
[FOR]
    EXECUTE STATEMENT sql-statement
    WITH {AUTONOMOUS|COMMON} TRANSACTION
    [...other options...]
    [INTO <variables>]
[DO psql-statement]
```

WITH CALLER PRIVILEGES

Добавлено: 2.5

Описание: По умолчанию операторы SQL выполняются с правами текущего пользователя. Спецификация WITH CALLER PRIVILEGES добавляет к ним привилегии для вызова хранимой процедуры или триггера, так же, как если бы оператор выполнялся непосредственно подпрограммой. WITH CALLER PRIVILEGES не имеет никакого эффекта, если также присутствует предложение ON EXTERNAL.

Синтаксис (частичный):

```
[FOR]
    EXECUTE STATEMENT sql-statement
    WITH CALLER PRIVILEGES
    [...other options...]
    [INTO <variables>]
[DO psql-statement]
```

ON EXTERNAL [DATA SOURCE]

Добавлено: 2.5

Описание: С предложением ON EXTERNAL DATA SOURCE SQL оператор

выполняется в отдельном соединении с той же или другой базой данных, возможно даже на другом сервере. Если строка подключения имеет значение NULL или '' (пустая строка), предложение ON EXTERNAL считается отсутствующим и оператор выполняется для текущей базы данных.

Синтаксис (частичный):

[FOR]
    EXECUTE STATEMENT *sql-statement*
    ON EXTERNAL [DATA SOURCE] <*connect-string*>
    [AS USER *user*]
    [PASSWORD *password*]
    [ROLE *role*]
    [...other options...]
    [INTO <*variables*>]
[DO *psql-statement*]

<*connect-string*> ::= [<*hostspec*>]*path-or-alias*
<*hostspec*> ::= <*tcpip-hostspec*> | <*netbeui-hostspec*>
<*tcpip-hostspec*> ::= *hostname*:
<*netbeui-hostspec*> ::= *hostname*\

ВНИМАНИЕ:
sql-statement, user, password, role и <connect-string> - строковые выражения. При явном задании этих аргументов (т.е. в виде символьных строк) их необходимо заключать в одинарные кавычки.

Пул подключений (Connection pooling):

- Внешние соединения используют по умолчанию предложение WITH COMMON TRANSACTION и остаются открытыми до закрытия текущей транзакции. Они могут быть снова использованы при последующих вызовах оператора EXECUTE STATEMENT, но только если строка подключения точно такая же;
- Внешние соединения , созданные с использованием предложения WITH AUTONOMOUS TRANSACTION, закрываются после выполнения оператора;
- Операторы WITH AUTONOMOUS TRANSACTION могут использовать соединения, которые ранее были открыты операторами WITH COMMON TRANSACTION. В этом случае использованное соединение остаётся открытым и после выполнения оператора, т.к. у этого соединения есть по крайней мере одна не закрытая транзакция.

Пул транзакций (Transaction pooling):

- При использовании предложения WITH COMMON TRANSACTION транзакции будут снова использованы как можно дольше. Они будут подтверждаться или откатываться вместе с текущей транзакцией;
- При использовании предложения WITH AUTONOMOUS TRANSACTION всегда запускается новая транзакция. Она будет подтверждена или отменена сразу же после выполнения оператора;

Обработка исключений: При использовании предложения ON EXTERNAL дополнительное соединение всегда делается через так называемого внешнего провайдера, даже если это соединение к текущей базе данных. Одним из последствий этого является то, что Вы не можете обработать исключение привычными способами. Каждое исключение, вызванное оператором, возвращает eds_connection или eds_statement ошибки. Для обработки исключений в коде PSQL Вы должны использовать WHEN GDSCODE eds_connection, WHEN GDSCODE eds_statement или WHEN ANY. Если предложение ON EXTERNAL не используется, то исключения перехватываются в обычном порядке, даже если это дополнительное соединение с текущей базой данных.

Дополнительные примечания:

- Набор символов, используемый для внешнего соединения, совпадает с используемым для текущего соединения;
- Двухфазные транзакции не поддерживаются;
- Подробности об авторизации пользователя приведены ниже в разделе AS USER, PASSWORD и ROLE .

AS USER, PASSWORD и ROLE

Добавлено: 2.5

Описание: При желании можно указать имя пользователя, пароль и/или роль, от имени которого должен быть выполнен оператор.

Синтаксис (частичный):

```
[FOR]
    EXECUTE STATEMENT sql-statement
    AS USER user
    PASSWORD password
    ROLE role
```

[...other options...]
[INTO <*variables*>]
[DO *psql-statement*]

ВНИМАНИЕ:
sql-statement, *user*, *password и role - строковые выражения. При явном задании этих аргументов (т.е. в виде символьных строк) их необходимо заключать в одинарные кавычки.*

Авторизация: То, как авторизуется пользователь и открыто ли отдельное соединение, зависит от присутствия и значений параметров ON EXTERNAL [DATA SOURCE], AS USER, PASSWORD и ROLE.

- При использовании предложения ON EXTERNAL открывается новое соединение и:

  – Если присутствует по крайней мере один из параметров AS USER, PASSWORD и ROLE, то будет предпринята попытка нативной аутентификации с указанными значениями параметров (в зависимости от строки соединения — локально или удалённо). Для недостающих параметров не используются никаких значений по умолчанию;
  – Если все три параметра отсутствуют и строка подключения не содержит имени сервера (или IP адреса), то новое соединение устанавливается к локальному серверу с пользователем и ролью текущего соединения. Термин 'локальный' означает 'компьютер, где установлен сервер Firebird'. Это совсем не обязательно компьютер клиента;
  – Если все три параметра отсутствуют, но строка подключения содержит имя сервера (или IP адреса), то будет предпринята попытка доверенной (trusted) авторизации к удалённому серверу. Если авторизация прошла, то удаленная операционная система назначит пользователю имя - обычно это учётная запись, под которой работает сервер Firebird.

- Если предложение ON EXTERNAL отсутствует:

  – Если присутствует по крайней мере один из параметров AS USER, PASSWORD и ROLE, то будет открыто соединение к текущей базе данных с указанными значениями параметров. Для недостающих параметров не используются никаких значений по умолчанию;
  – Если все три параметра отсутствуют, то оператор выполняется в текущем соединении.

Внимание: Если значение параметра NULL или '', то весь параметр считается

отсутствующим. Кроме того, если параметр считается отсутствующим, то AS USER принимает значение CURRENT_USER, а ROLE — CURRENT_ROLE. Сравнение при авторизации сделано чувствительным к регистру: в большинстве случаев это означает, что имена пользователя и роли должны быть написаны в верхнем регистре.

Параметризованные операторы

Добавлено: 2.5

Описание: Начиная с Firebird 2.0 разрешено использовать параметры в SQL операторе. При выполнении оператора [FOR] EXECUTE STATEMENT значения должны быть присвоено каждому параметру.

Синтаксис (частичный):

```
[FOR]
    EXECUTE STATEMENT (<parameterized-statement>)
                      (<param-assignments>)
    [...options...]
    [INTO <variables>]
[DO psql-statement]

<parameterized-statement> ::= Оператор SQL, содержащий
    <named-param>s или <positional-param>s

<named-param> ::= :paramname
<positional-param> ::= ?
<param-assignments> ::= <named-assignments> |
<positional-assignments>
<named-assignments> ::= paramname := value [, paramname := value ...]
<positional-assignments> ::= value [, value ...]
```

ВНИМАНИЕ:
<parameterized-statement> - строковые выражения. При явном задании этих аргументов (т.е. в виде символьных строк) их необходимо заключать в одинарные кавычки.

Примеры:

Для именованных параметров:

```
...
        DECLARE LICENSE_NUM VARCHAR(15);
        DECLARE CONNECT_STRING VARCHAR(100);
        DECLARE STMT VARCHAR(100) =
                ' SELECT LICENSE FROM CARS ' ||
                ' WHERE DRIVER = :DRV AND LOCATION = :LOC ';
    BEGIN
        ...
        SELECT CONNSTR
                FROM DATABASES
                WHERE CUST_ID = :ID
        INTO :CONNECT_STRING;
        ...
        FOR SELECT ID
                FROM DRIVERS
        INTO :CURRENT_DRIVER DO
                BEGIN
                        FOR SELECT LOCATION
                                FROM DRIVER_LOCATIONS
                                WHERE DRIVER_ID = :CURRENT_DRIVER
                        INTO :CURRENT_LOCATION DO
                                BEGIN
                                        ...
                                        EXECUTE STATEMENT (STMT)
                                                (DRV := :CURRENT_DRIVER,
                                                 LOC := :CURRENT_LOCATION)
                                                ON EXTERNAL
                                                        :CONNECT_STRING
                                        INTO :LICENSE_NUM;
                                        ...
```

Для позиционных параметров:

```
...
        DECLARE LICENSE_NUM VARCHAR(15);
        DECLARE CONNECT_STRING VARCHAR(100);
        DECLARE STMT VARCHAR(100) =
                ' SELECT LICENSE FROM CARS ' ||
                ' WHERE DRIVER = ? AND LOCATION = ? ';
    BEGIN
        ...
        SELECT CONNSTR
                FROM DATABASES
```

```
        WHERE CUST_ID = :ID
INTO :CONNECT_STRING;
...
FOR SELECT ID
        FROM DRIVERS
INTO :CURRENT_DRIVER DO
        BEGIN
                FOR SELECT LOCATION
                        FROM DRIVER_LOCATIONS
                        WHERE DRIVER_ID = :CURRENT_DRIVER
                INTO :CURRENT_LOCATION DO
                        BEGIN
                                ...
                                EXECUTE STATEMENT (STMT)
                                        (:CURRENT_DRIVER,
                                         :CURRENT_LOCATION)
                                        ON EXTERNAL
                                                :CONNECT_STRING
                                INTO :LICENSE_NUM;
                                ...
```

Примечания: Некоторые вещи, которые надо знать:

- Когда у оператора есть параметры, они должны быть помещены в круглые скобки при вызове EXECUTE STATEMENT, независимо от вида их представления: непосредственно в виде строки, как имя переменной или как выражение;
- Именованным параметрам должно предшествовать двоеточие (:) в самом операторе, но не при присвоении значения параметру;
- Каждый именованный параметр может использоваться в операторе несколько раз, но только один раз при присвоении значения;
- Каждому именованному параметру должно быть присвоено при вызове оператора EXECUTE STATEMENT: присваивать им значения можно в любом порядке;
- Оператор присвоения значения именованному параметру ":=", а не "=" - как в SQL;
- Для позиционных параметров число подставляемых значений должно точно равняться числу параметров (вопросительных знаков) в операторе.

Предостережения для оператора EXECUTE STATEMENT

1. Не существует способа проверить синтаксис выполняемого SQL

оператора;

2. Нет никаких проверок зависимостей для обнаружения удалённых столбцов в таблице или самой таблицы;
3. Несмотря на то, что производительность в циклах была значительно улучшена в Firebird 2.5, их выполнение значительно медленнее, чем при непосредственном выполнении;
4. Возвращаемые значения строго проверяются на тип данных во избежание непредсказуемых исключений преобразования типа. Например, строка '1234' преобразуется в целое число 1234, а строка 'abc' вызовет ошибку преобразования;
5. *От переводчика — настоятельно не рекомендуется использовать оператор EXECUTE STATEMENT для изменения метаданных.*

В целом эта функция должна использоваться очень осторожно, а вышеупомянутые факторы всегда должны приниматься во внимание. Если такого же результата можно достичь с использованием PSQL и/или DSQL, то это всегда предпочтительнее.

EXIT

Доступно: PSQL

Изменено: 1.5

Описание: Начиная с Firebird 1.5 разрешено использовать оператор EXIT во всём PSQL. В более ранних версиях он поддерживался только в хранимых процедурах, но не в триггерах.

Наилучшая альтернатива: Оператор LEAVE

FETCH CURSOR

Доступно: PSQL

Добавлено: 2.0

Описание: Выбирает следующую строку данных из результирующего набора данных курсора и присваивает значения столбцов в переменные PSQL.

Синтаксис:

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

Пример: См. DECLARE ... CURSOR

Примечания:

- Контекстная переменная ROW_COUNT принимает значение 1, если выборка возвратила строку данных и 0 при достижении конца набора данных;
- Вы можете делать позиционный UPDATE или DELETE выбранной строки с использованием предложения WHERE CURRENT OF.

FOR EXECUTE STATEMENT ... DO

Доступно: PSQL

Добавлено: 1.5

Описание: См. раздел Возврат любого количества строк данных

FOR SELECT ... INTO ... DO

Доступно: PSQL

Описание: Выполняет оператор SQL и возвращает результирующий набор данных. В каждой итерации цикла значения полей текущей строки копируются в локальные переменные. Добавление предложения AS CURSOR делает возможным позиционное удаление и обновление данных. Операторы FOR SELECT могут быть вложенными.

Синтаксис:

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

<*select-stmt*> ::= Корректный опеоратор SELECT

<*var*> ::= Имя переменной PSQL, опционально с двоеточием впереди
<*psql-stmt*> ::= Одиночный оператор или блок PSQL кода

- Оператор SELECT может иметь именованные параметры. Например, “SELECT NAME || :SFX FROM NAMES WHERE NUMBER = :NUM”. Каждый параметр должен быть заранее объявленной переменной PSQL или входным/выходным параметром модуля PSQL;
- Внимание! Если значение переменной PSQL, используемой в операторе SELECT, изменяется во время выполнения цикла, то её новое значение может (но не всегда) использоваться при выборке следующих строк. Лучше избегать таких ситуаций. Если Вам действительно требуется такое поведение, то необходимо тщательно протестировать код и убедиться, что Вы точно знаете, как изменения переменной влияют на результаты выборки. Особо отмечу, что поведение может зависеть от плана запроса, в частности, от используемых индексов. В настоящее время нет строгих правил для таких ситуаций, но в новых версиях Firebird это может измениться.

Примеры:

```
CREATE PROCEDURE SHOWNUMS
RETURNS(
        AA INTEGER,
        BB INTEGER,
        SM INTEGER,
        DF INTEGER)
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 INTEGER,
        FIELD CHAR(32))
```

```
AS
BEGIN
        FOR SELECT RDB$RELATION_NAME
                FROM RDB$RELATIONS
                ORDER BY 1
        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 = ' "';
        -- Для исключения повтора имён таблиц и представлений
                                        SUSPEND;
                                END
                END
END
```

Предложение AS CURSOR

Доступно: PSQL

Добавлено: IB

Описание: Дополнительное предложение AS CURSOR создает именованный курсор, на который можно ссылаться (с использованием WHERE CURRENT OF) в цикле FOR SELECT для того, чтобы изменить или удалить текущую строку. Функция была добавлена ещё в InterBase, но не описана в его Language Reference.

Пример:

```
CREATE PROCEDURE DELTOWN (
        TOWNTODELETE VARCHAR(24))
RETURNS (
        TOWN VARCHAR(24),
        POP INTEGER)
```

```
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
```

Примечания:

- Предложение "FOR UPDATE", разрешённое для использования в операторе SELECT, но оно не является обязательным для успешного выполнения позиционированного обновления или удаления;
- Убедитесь в том, что имя курсора, определенное здесь, не совпадает ни с какими именами, созданными ранее оператором DECLARE ... CURSOR ;
- Предложение AS CURSOR не поддерживается в операторе FOR EXECUTE STATEMENT, даже если выполняемый в нём запрос на выборку данных является подходящим.

IN AUTONOMOUS TRANSACTION

Доступно: PSQL

Добавлено: 2.5

Описание: Код, работающий в автономной транзакции, будет подтверждаться сразу же после успешного завершения независимо от состояния родительской транзакции. Это бывает нужно, когда определенные действия не должны быть отменены, даже в случае возникновения ошибки в родительской транзакции.

Синтаксис:

```
IN AUTONOMOUS TRANSACTION DO <psql-statement>
```

Пример:

```
CREATE TRIGGER TR_CONNECT ON CONNECT
AS
BEGIN
-- Проверяем, что все действия сохраняются в журнал:
        IN AUTONOMOUS TRANSACTION DO
                INSERT INTO LOG (
                        MSG)
                VALUES (
                        'USER ' || CURRENT_USER || ' CONNECTS.');
                IF (CURRENT_USER IN (
                        SELECT USERNAME
                                FROM BLOCKED_USERS)) THEN
                        BEGIN
-- Проверяем, что все действия сохраняются в журнал
-- и отправляем сообщение о событии:
                                IN AUTONOMOUS TRANSACTION DO
                                        BEGIN
                                                INSERT INTO LOG (
                                                        MSG)
                                                VALUES (
                                                        'USER ' ||
                                                        CURRENT_USER ||
                                                        ' REFUSED.');
                                                POST_EVENT
                                                        'CONNECTION ATTEMPT' ||
                                                        ' BY BLOCKED USER!';
                                        END
-- теперь вызываем исключение:
                                EXCEPTION EX_BADUSER;
                        END
END
```

Примечания:

- Автономные транзакции имеют тот же уровень изоляции, как и у родительской транзакции;
- Поскольку автономная транзакция абсолютно независима от своего родителя, необходимо соблюдать осторожность, чтобы избежать взаимных блокировок;
- Если исключение произойдет в автономной транзакции, то все действия, сделанные в её рамках, будут отменены.

LEAVE

Доступно: PSQL

Добавлено: 1.5

Изменено: 2.0

Описание: Оператор LEAVE моментально прекращает работу внутреннего цикла операторов WHILE или FOR. С использованием опционального параметра метки (добавлен в Firebird 2.0) LEAVE также может выйти и из внешних циклов, при этом выполнение кода продолжается с первого оператора, следующего после прекращения блока внешнего цикла.

Синтаксис:

```
[label:]
{FOR | WHILE} ... DO
    ...
    (здесь возможны вложенные циклы, с меткой или без неё)
    ...
    LEAVE [label];
```

Пример:

В приведённом ниже примере при ошибке при вставке записи происходит запись об этом в таблицу лога ошибок и прекращается выполнение цикла. Код продолжает выполняться со строки “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;
...
```

Следующий пример использует метку. “LEAVE LOOPA” завершает внешний цикл, а “LEAVE LOOPB” - внутренний. Обратите внимание: простого оператора "LEAVE" также было бы достаточно, чтобы завершить внутренний цикл.

```
...
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

Доступно: PSQL

Добавлено: 2.0

Описание: Открывает ранее объявленный курсор, выполняет объявленный в нём оператор SELECT и получает записи из результирующего набора данных.

Синтаксис:

OPEN *cursorname*;

Пример: См. DECLARE ... CURSOR

Разрешение использования оператора PLAN в триггерах

Изменено: 1.5

Описание: До Firebird 1.5 триггеры, содержащие в своём коде оператор не могли быть откомпилированы. Корректный план теперь может быть использован в коде триггера и будет использоваться при его срабатывании.

Подзапросы в выражениях PSQL

Изменено: 2.5

Описание: Ранее подзапросы не могли использоваться в качестве выражений в PSQL, даже если они возвращали единственное значение. Это заставляло использовать конструкцию SELECT ... INTO в дополнительную переменную, которая часто не была так уж необходима. Начиная с Firebird 2.5 поддерживается прямое использование скалярных подзапросов, как будто бы они были простыми выражениями.

Пример:

Теперь в PSQL допустимы следующие конструкции:

```
VAR = (SELECT ... FROM ...);

IF ((SELECT ... FROM ...) = 1)
      THEN …

IF (1 = ANY (SELECT ... FROM …))
      THEN ...
IF (1 IN (SELECT ... FROM …))
      THEN ...
```

В первых двух примерах Вы, конечно, должны быть уверены, что оператор SELECT возвращает только одну строку!

UDF, вызываемые как пустые функции

Изменено: 2.0

Описание: Начиная с Firebird 2.0 в PSQL можно вызывать UDF, не присваивая значение результата, т.е. как процедура Pascal или пустая функция C. В большинстве случаев это не имеет смысла, т.к. основная цель почти всех UDF состоит в том, чтобы выдать определённый результат. Однако некоторые функции выполняют задачи, не требующие возврата результата, и если Вам не нужен результат выполнения UDF, то нет и нужды присваивать его в фиктивную переменную и тем самым выиграть во времени её выполнения.

Примечание

Функции RDB$GET_CONTEXT и RDB$SET_CONTEXT хотя и классифицированы в этом руководстве как внутренние, фактически являются автоматически объявленными UDF. Поэтому их можно вызывать без присваивания результата их выполнения в переменную. Конечно, это имеет смысл только для функции RDB$SET_CONTEXT.

Разрешение использования WHERE CURRENT OF для курсоров представления

Изменено: 2.0, 2.1

Описание: Из-за возможных проблем надежности в Firebird 2.0 было запрещено использование предложения WHERE CURRENT OF для курсоров в представлениях. В Firebird 2.1 была улучшена логика проверки представлений, что позволило снять это ограничение.

Глава 10

Безопасность и управление доступом к данным

ALTER ROLE

Доступно: DSQL

Добавлено: 2.5

Описание: Единственной целью оператора ALTER ROLE является разрешение или запрещение автоматического (т.е. при условии успешной авторизации) использования роли RDB$ADMIN для администраторов операционной системы Windows. Для Полное описание этого вопроса Вы можете прочитать в разделах Роль RDB$ADMIN и AUTO ADMIN MAPPING .

Синтаксис:

```
ALTER ROLE RDB$ADMIN {SET|DROP} AUTO ADMIN MAPPING
```

GRANT и REVOKE

GRANTED BY

Доступно: DSQL

Добавлено: 2.5

Описание: При предоставлении прав в базе данных в качестве лица, предоставившего эти права, обычно записывается текущий пользователь. Используя предложение GRANTED BY можно предоставлять права от имени другого пользователя. При использовании оператора REVOKE после GRANTED BY права будут удалены только в том случае, если они были зарегистрированы от удаляющего пользователя. Для облегчения миграции из некоторых других реляционных СУБД нестандартное предложение AS поддерживается как синоним оператора GRANTED BY.

Доступ: Предложение GRANTED BY может использовать:

- Владелец базы данных;
- SYSDBA;
- Любой пользователь, имеющий права на роль RDB$ADMIN и указавший её при соединении с базой данных;
- При использовании флага AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication), даже без указания роли.

Даже владелец роли не может использовать GRANTED BY, если он не находится в вышеупомянутом списке.

Синтаксис:

```
GRANT
    {<privileges> ON <object> | role}
    TO <grantees>
    [WITH {GRANT|ADMIN} OPTION]
    [{GRANTED BY | AS} [USER] grantor]

REVOKE
    [{GRANT|ADMIN} OPTION FOR]
    {<privileges> ON <object> | role}
    FROM <grantees>
    [{GRANTED BY | AS} [USER] grantor]
```

Это не полный синтаксис операторов GRANT и REVOKE, но он полностью описывает возможности GRANTED BY.

Пример:

```
-- Соединение с базой данных от имени её владельца BOB

CREATE ROLE DIGGER;
GRANT DIGGER TO FRANCIS;
GRANT DIGGER TO FRED;
GRANT DIGGER TO FRANK
    WITH ADMIN OPTION GRANTED BY FRITZ;
COMMIT;

REVOKE DIGGER FROM FRED;
```

```
        -- OK

        REVOKE ADMIN OPTION FOR DIGGER FROM FRANK;
        -- ERROR: "BOB IS NOT GRANTOR OF ROLE ON DIGGER TO
FRANK."

        REVOKE ADMIN OPTION FOR DIGGER FROM FRANK GRANTED
BY FRITZ;
        -- OK

        REVOKE DIGGER FROM FRANK
        -- ERROR: "BOB IS NOT GRANTOR OF ROLE ON DIGGER TO
FRANK."
        COMMIT;

        -- Отключение пользователя BOB, подключение от имени FRITZ:

        REVOKE DIGGER FROM FRANK;
        -- OK

        REVOKE DIGGER FROM FRANCIS;
        -- ERROR: "FRITZ IS NOT GRANTOR OF ROLE ON DIGGER TO
FRANCIS."

        REVOKE DIGGER FROM FRANCIS GRANTED BY BOB;
        -- ERROR: "ONLY SYSDBA OR DATABASE OWNER CAN USE
GRANTED BY CLAUSE"

        COMMIT;
```

Примечание: Обратите внимание на то, что опции GRANT или ADMIN - просто флаг в записи прав; у них нет отдельного лица, предоставившего это право. Например, эта строка:

```
        GRANT DIGGER TO FRANK
              WITH ADMIN OPTION GRANTED BY FRITZ
```

не обозначает, что FRANK даны права на роль DIGGER, а ADMIN OPTION от имени FRITZ. Она обозначает, что ему даны права на роль DIGGER и ADMIN OPTION от имени FRITZ.

REVOKE ALL ON ALL

Доступно: DSQL

Добавлено: 2.5

Описание: Отменяет все привилегии (включая роли) на всех объектах от одного или более пользователей и/или ролей. Это - быстрый способ "очистить" (отобрать) права, когда пользователю должен быть заблокирован доступ к базе данных.

Синтаксис:

REVOKE ALL ON ALL FROM <*grantee*> [, <*grantee*> …]

<*grantee*> ::= [USER] *username* | [ROLE] *rolename*

Пример:

REVOKE ALL ON ALL FROM BUDDY, PEGGY, SUE

Примечания:

- Когда оператор REVOKE ALL ON ALL вызывается привилегированным пользователем (владельцем базы данных, SYSDBA или любым пользователем, у которого CURRENT_ROLE - RDB$ADMIN), удаляются все права независимо от того, кто их предоставил. В противном случае удаляются только права, предоставленные текущим пользователем;
- Не поддерживается предложение GRANTED BY;
- Этот оператор не удаляет флаг пользователя, давшего права на хранимые процедуры, триггеры или представлений (права на такие объекты конечно удаляются).

REVOKE ADMIN OPTION

Доступно: DSQL

Добавлено: 2.0

Описание: Отменяет ранее предоставленную административную опцию (право на передачу предоставленной пользователю роли другим) из получателей

гранта, не отменяя прав на роль. В одном операторе могут быть обработаны несколько ролей и/или грантополучателей.

Синтаксис:

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

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

Пример:

REVOKE ADMIN OPTION FOR MANAGER FROM JOHN, PAUL, GEORGE, RINGO

Если пользователь получил администраторскую опцию от нескольких грантодателей, то каждый из них должен отменить ее, иначе пользователь все ещё будет в состоянии предоставить рассматриваемую роль (роли) другим.

Роль RDB$ADMIN

Добавлено: 2.5

Описание: В Firebird 2.5 добавлена системная роль RDB$ADMIN, присутствующая в каждой базе данных. Предоставление пользователю роли RDB$ADMIN в базе данных дает ему права SYSDBA, *но только в этой базе данных*. В обычной базе данных это означает полный контроль над всеми объектами. В базе данных пользователей это означает возможность создавать, изменять и удалять учетные записи пользователей. В обоих случаях пользователь с правами RDB$ADMIN роли может всегда передавать эту роль другим. Другими словами, “WITH ADMIN OPTION” уже встроен в эту роль и эту опцию можно не указывать.

В обычной базе данных

Предоставление роли RDB$ADMIN в обычной базе данных

Синтаксис предоставления и удаления роли RDB$ADMIN в обычной базе данных:

GRANT RDB$ADMIN TO *username*
REVOKE RDB$ADMIN FROM *username*

Права на роль RDB$ADMIN могут давать:

- Владелец базы данных;
- SYSDBA;
- Любой пользователь, имеющий права на роль RDB$ADMIN и указавший её при соединении с базой данных;
- При использовании флага AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication), даже без указания роли.

Использование роли RDB$ADMIN в обычной базе данных

Для использования прав роли RDB$ADMIN пользователь просто определяет её при соединении с базой данных.

В базе данных пользователей

Предоставление роли RDB$ADMIN в базе данных пользователей

Так как никто не может соединиться с базой данных пользователей, то операторы GRANT и REVOKE здесь не могут использоваться. Вместо этого роль RDB$ADMIN предоставляют и удаляют новыми командами управления пользователями SQL:

CREATE USER *newuser* PASSWORD '*password*' GRANT ADMIN ROLE
ALTER USER *existinguser* GRANT ADMIN ROLE
ALTER USER *existinguser* REVOKE ADMIN ROLE

Пожалуйста, помните, что GRANT ADMIN ROLE и REVOKE ADMIN ROLE это не операторы GRANT и REVOKE. Это параметры для CREATE USER и ALTER USER.

Кроме того, для утилиты командной строки gsec добавлен новый параметр -admin:

gsec -add *newuser* -pw *password* -admin yes
gsec -mo *existinguser* -admin yes
gsec -mo *existinguser* -admin no

При модификации данных пользователя при этом можно использовать и другие параметры, например, -user and -pass, или -trusted.

Права на роль RDB$ADMIN могут давать:

- SYSDBA;
- Любой пользователь, имеющий права на роль RDB$ADMIN в базе данных пользователей и указавший её при соединении с базой данных (или во время работы с утилитой gsec);
- При использовании флага AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication), даже без указания роли.

Использование роли RDB$ADMIN в базе данных пользователей

Для управления учетными записями пользователей через SQL пользователь, имеющий права на роль RDB $ ADMIN, должен подключиться к базе данных с этой ролью. Так как к базе данных пользователей не имеет права соединяться никто, то пользователь должен подключиться к обычной базе данных, где он также имеет права на роль RDB$ADMIN. Он определяет роль при соединении с обычной базой данных и может в ней выполнить любой SQL запрос. Это не самое элегантное решение, но это единственный способ управлять пользователями через SQL запросы. Если нет обычной базы данных, где у пользователя есть права на роль RDB$ADMIN, то управление учётными записями посредством SQL запросов недоступно.

Для управления учётными записями пользователей с помощью утилиты командной строки gsec при подключении необходимо указать дополнительный параметр -role rdb$admin.

AUTO ADMIN MAPPING

Операционная система: только Windows

Добавлено: 2.5

Описание: В Firebird 2.1 администраторы операционной системы Windows автоматически получают права SYSDBA при подключении к базе данных (если, конечно, разрешена доверенная авторизация; *Примечание переводчика: доверенная авторизация разрешена, если параметр Authentication в конфигурационном файле Firebird имеет значение "trusted" или "mixed"*). В Firebird 2.5 это уже запрещено. Имеют ли администраторы автоматические права SYSDBA теперь зависит от

установки значения флага AUTO ADMIN MAPPING. Это флаг в каждой из баз данных, который по умолчанию выключен. Если флаг AUTO ADMIN MAPPING включен, то он действует, когда администратор Windows: а) подключается с помощью доверенной аутентификации, и б) не определяет никакой роли при подключении. После успешного “auto admin” подключения текущей ролью будет являться RDB$ADMIN.

В обычной базе данных

Включение и выключение флага AUTO ADMIN MAPPING в обычной базе данных осуществляется следующим образом:

```
ALTER ROLE RDB$ADMIN SET AUTO ADMIN MAPPING
ALTER ROLE RDB$ADMIN DROP AUTO ADMIN MAPPING
```

Эти операторы могут быть выполнены пользователями с достаточными правами, а именно:

- SYSDBA;
- Любой пользователь, имеющий права на роль RDB$ADMIN в базе данных пользователей и указавший её при соединении с базой данных (или во время работы с утилитой gsec);
- При использовании флага AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication), даже без указания роли. В обычных базах данных состояние флага AUTO ADMIN MAPPING проверяется только во время подключения. Если подключение производилось с ролью RDB$ADMIN, то даже при удалении прав на неё права сохраняются до момента отключения от базы данных. Аналогично, включение флага AUTO ADMIN MAPPING не изменит текущую роль на RDB$ADMIN для администраторов, которые уже были подключены к базе данных.

В базе данных пользователей

Нет никаких операторов SQL, чтобы включить или выключить флаг AUTO ADMIN MAPPING в базе данных пользователей. Для этого можно использовать только утилиту командной строки gsec:

```
gsec -mapping set
gsec -mapping drop
```

При этом в командной строке можно использовать и другие параметры, например, -user and -pass, или -trusted.

Включение/выключение флага AUTO ADMIN MAPPING разрешено выполнять только:

- SYSDBA;
- При включенном флаге AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication) без указания роли.

В отличие от случая с обычными базами данных пользователи, подключённые к базе данных с ролью RDB$ADMIN, не могут включить или выключить флаг AUTO ADMIN MAPPING в базе данных пользователей. Также заметьте, что администратор операционной системы Windows может только выключить флаг AUTO ADMIN MAPPING. При выключении флага он отключает сам механизм, который предоставлял ему доступ и, таким образом, он не будет в состоянии включить AUTO ADMIN MAPPING. Даже в интерактивном gsec сеансе новая установка флага сразу вступает в силу.

Команды SQL для управления пользователями

Доступно: DSQL

Добавлено: 2.5

Описание: Начиная с Firebird 2.5 добавлена возможность управлять учётными записями пользователей средствами операторов SQL. Но эта возможность предоставлена только следующим пользователям:

- SYSDBA;
- Любому пользователю, имеющему права на роль RDB$ADMIN в базе данных пользователей и права на эту же роль для базы данных в активном подключении (пользователь должен подключаться к базе данных с этой - RDB$ADMIN - ролью);
- При включенном флаге AUTO ADMIN MAPPING - любой администратор операционной системы Windows (при условии использования сервером доверенной авторизации - trusted authentication) без указания роли. При этом не важно, включен или выключен флаг AUTO ADMIN MAPPING в самой базе данных, т.к. Он включен в базе данных пользователей (security2.fdb).

Непривилегированные пользователи могут использовать только оператор ALTER USER для изменения собственной учётной записи.

CREATE USER

Описание: Создаёт учётную запись пользователя Firebird.

Синтаксис:

CREATE USER *username* PASSWORD '*password*'
[FIRSTNAME '*firstname*']
[MIDDLENAME '*middlename*']
[LASTNAME '*lastname*']
[GRANT ADMIN ROLE]

С опцией GRANT ADMIN ROLE создаётся новый пользователь с правами роли RDB$ADMIN в базе данных пользователя. Это позволяет ему управлять учётными записями пользователей, но не дает ему специальных полномочий в обычных базах данных. Более подробная информация изложена в разделе Роль RDB$ADMIN .

Пример:

CREATE USER bigshot PASSWORD 'buckshot'
CREATE USER john password 'fYe_3Ksw' FIRSTNAME 'John' LASTNAME 'Doe'
CREATE USER mary PASSWORD 'lamb_chop' FIRSTNAME 'Mary' GRANT ADMIN ROLE

ALTER USER

Описание: Изменяет данные учётной записи пользователя. Это единственный оператор управления учётными записями, который может также использоваться непривилегированными пользователями для изменения их собственных учетных записей.

Синтаксис:

ALTER USER *username*
[PASSWORD '*password*']

[FIRSTNAME '*firstname*']
[MIDDLENAME '*middlename*']
[LASTNAME '*lastname*']
[{GRANT|REVOKE} ADMIN ROLE]

-- Должен присутствовать по крайней мере один из дополнительных параметров
-- Опции GRANT/REVOKE ADMIN ROLE предназначены для привилегированных пользователей

Пример:

```
ALTER USER bobby PASSWORD '67-UiT_G8' GRANT ADMIN ROLE
ALTER USER dan FIRSTNAME 'No_Jack' lastname 'Kennedy'
ALTER USER dumbbell REVOKE ADMIN ROLE
```

DROP USER

Описание: Удаляет учётную запись пользователя Firebird.

Синтаксис:

DROP USER *username*

Пример:

```
DROP USER timmy
```

Глава 11

Контекстные переменные

CURRENT_CONNECTION

Доступно: DSQL, PSQL

Добавлено: 1.5

Изменено: 2.1

Описание: Контекстная переменная CURRENT_CONNECTION содержит уникальный идентификатор текущего соединения.

Тип: INTEGER

Пример:

```
SELECT CURRENT_CONNECTION FROM RDB$DATABASE

EXECUTE PROCEDURE P_LOGIN(CURRENT_CONNECTION)
```

Значение контекстной переменной CURRENT_CONNECTION хранится в странице заголовка базы данных и сбрасывается в 0 после восстановления. Начиная с версии Firebird 2.1 она увеличивается при каждом новом подключении (В предыдущих версиях это было только если в рамках подключения пользователь хотя бы считывал что либо из базы данных). В результате CURRENT_CONNECTION теперь указывает количество подключений к базе данных с момента её создания (или после восстановления).

CURRENT_ROLE

Доступно: DSQL, PSQL

Добавлено: 1.0

Описание: Контекстная переменная CURRENT_ROLE служит для определения роли, с которой произошло подключение к базе данных. Если подключение произошло без указания роли, то CURRENT_ROLE принимает значение NONE.

Тип: VARCHAR(31)

Пример:

```
IF (CURRENT_ROLE <> 'MANAGER') THEN
    EXCEPTION ONLY_MANAGERS_MAY_DELETE;
ELSE
    DELETE FROM CUSTOMERS
        WHERE CUSTNO = :CUSTNO;
```

CURRENT_ROLE всегда возвращает значение роли подключения или NONE. Если пользователь соединяется с несуществующей ролью, то сервер присваивает ей значение NONE не вызывая при этом ошибку.

CURRENT_TIME

Доступно: DSQL, ESQL, PSQL

Изменено: 2.0

Описание: CURRENT_TIME возвращает текущее серверное время. В версиях до Firebird 2.0, дробная часть всегда была “.0000”, т.е. давая точность до секунды. Начиная с Firebird 2.0 Вы можете определить точность при запросе значения этой переменной. Значение по умолчанию - все ещё 0 после запятой, т.е. точность значения до одной секунды.

Тип: Время (TIME)

Синтаксис:

```
CURRENT_TIME [(precision)]

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

Дополнительный параметр *точности* (*precision*) не поддерживается в ESQL/

Пример:

```
SELECT CURRENT_TIME
    FROM RDB$DATABASE
-- возвращает, например 14:20:19.0000

SELECT CURRENT_TIME(2) FROM RDB$DATABASE
-- возвращает, например 14:20:23.1200
```

Примечания:

- В отличие от CURRENT_TIME, точность по умолчанию CURRENT_TIMESTAMP изменилась до 3-х десятичных знаков. В результате контекстная переменная CURRENT_TIMESTAMP больше не является точной суммой переменных CURRENT_DATE и CURRENT_TIME, если Вы явно не указали точность CURRENT_TIME;
- В модуле PSQL (процедура, триггер или исполнимый блок), значение CURRENT_TIME останется постоянным до момента полного выполнения кода независимо от того, когда Вы его считываете. Если происходит вызов вложенного (вложенных) модуля, то значение CURRENT_TIME всё равно останется таким же, как и в модуле самого верхнего уровня. Если Вам требуется получение реального значения времени в PSQL – например, для измерения временных интервалов – используйте контекстную переменную 'NOW' с полным приведением типа (не краткий синтаксис).

CURRENT_TIMESTAMP

Доступно: DSQL, ESQL, PSQL

Изменено: 2.0

Тип: Дата и время (TIMESTAMP)

Описание: CURRENT_TIME STAMP возвращает текущее серверную дату и время. В версиях до Firebird 2.0, дробная часть всегда была “.0000”, т.е. давая точность после запятой 0. Начиная с Firebird 2.0 Вы можете определить точность при запросе значения этой переменной. Значение по умолчанию — 3 цифры после запятой, т.е. точность вплоть до одной миллисекунды.

Синтаксис:

```
CURRENT_ TIMESTAMP [(precision)]
```

precision ::= 0 | 1 | 2 | 3

Дополнительный параметр *точности* (*precision*) не поддерживается в ESQL/

Пример:

```
SELECT CURRENT_TIMESTAMP
    FROM RDB$DATABASE
-- возвращает, например 2008-08-13 14:20:19.6170

SELECT CURRENT_TIMESTAMP(2)
    FROM RDB$DATABASE
-- возвращает, например 2008-08-13 14:20:23.1200
```

Примечания:

- Точность по умолчанию для CURRENT_TIME — 0 цифр после запятой, т.е. начиная с Firebird 2.0 CURRENT_TIMESTAMP больше не точная сумма CURRENT_DATE и CURRENT_TIME, если явно не определять точность CURRENT_TIME;
- В модуле PSQL (процедура, триггер или исполнимый блок), значение CURRENT_ TIMESTAMP останется постоянным до момента полного выполнения кода независимо от того, когда Вы его считываете. Если происходит вызов вложенного (вложенных) модуля, то значение CURRENT_ TIMESTAMP всё равно останется таким же, как и в модуле самого верхнего уровня. Если Вам требуется получение реального значения времени в PSQL – например, для измерения временных интервалов – используйте контекстную переменную 'NOW' с полным приведением типа (не краткий синтаксис).

CURRENT_TRANSACTION

Доступно: DSQL, PSQL

Добавлено: 1.5

Описание: Контекстная переменная CURRENT_TRANSACTION содержит уникальный номер текущей транзакции.

Тип: INTEGER

Примеры:

```
SELECT CURRENT_TRANSACTION
    FROM RDB$DATABASE

NEW.TXN_ID = CURRENT_TRANSACTION;
```

Значение CURRENT_TRANSACTION хранится в странице заголовка базы данных и сбрасывается в 0 после восстановления. Оно увеличивается при старте новой транзакции.

CURRENT_USER

Доступно: DSQL, PSQL

Добавлено: 1.0

Описание: Контекстная переменная CURRENT_USER содержит имя текущего подключённого пользователя. Это полностью эквивалентно USER.

Тип: VARCHAR(31)

Пример:

```
CREATE TRIGGER BI_CUSTOMERS FOR CUSTOMERS
    BEFORE INSERT AS
BEGIN
    NEW.ADDED_BY = CURRENT_USER;
    NEW.PURCHASES = 0;
END
```

DELETING

Доступно: PSQL

Добавлено: 1.5

Описание: Контекстная переменная DELETING доступна только в триггерах и указывает на то, что триггер сработал в результате выполнения операции DELETE. Может использоваться в триггерах на различные события (Триггеры на несколько типов событий).

Тип: логический (boolean)

Пример:

```
...
IF (DELETING) THEN
    BEGIN
        INSERT INTO REMOVED_CARS (
            ID, MAKE, MODEL,
            REMOVED)
        VALUES (
            OLD.ID, OLD.MAKE, OLD.MODEL,
            CURRENT_TIMESTAMP);
    END
...
```

GDSCODE

Доступно: PSQL

Добавлено: 1.5

Изменено: 2.0

Описание: В блоке обработки ошибок "WHEN ... DO" контекстная переменная GDSCODE содержит числовое представление текущего кода ошибки Firebird. До версии Firebird 2.0 GDSCODE можно было получить только с использованием конструкции WHEN GDSCODE. Теперь эту контекстную переменную можно также использовать в блоках WHEN ANY, WHEN SQLCODE и WHEN EXCEPTION КОГДА ЛЮБОЙ, КОГДА SQLCODE и КОГДА ИСКЛЮЧЕНИЕ при условии, что код ошибки соответствует коду ошибки Firebird. Вне обработчика ошибок GDSCODE всегда равен 0. Вне PSQL GDSCODE не существует вообще.

Тип: INTEGER

Пример:

```
...
WHEN GDSCODE GRANT_OBJ_NOTFOUND,
     GDSCODE GRANT_FLD_NOTFOUND,
```

```
        GDSCODE GRANT_NOPRIV,
        GDSCODE GRANT_NOPRIV_ON_BASE
DO
    BEGIN
        EXECUTE PROCEDURE
            LOG_GRANT_ERROR(GDSCODE);
        EXIT;
    END
...
```

Обратите внимание, пожалуйста: После, WHEN GDSCODE Вы должны использовать символьные имена — такие, как grant_obj_notfound и т.д. Но контекстная переменная GDSCODE - ЦЕЛОЕ ЧИСЛО. Для сравнения его с определенной ошибкой Вы должны использовать числовое значение, например, 335544551 для grant_obj_notfound.

INSERTING

Доступно: PSQL

Добавлено: 1.5

Описание: Контекстная переменная INSERTING доступна только в триггерах и указывает на то, что триггер сработал в результате выполнения операции INSERT. Может использоваться в триггерах на различные типы событий (Триггеры на несколько типов событий).

Тип: логический (boolean)

Пример:

```
…
IF (INSERTING OR UPDATING) THEN
    BEGIN
        IF (NEW.SERIAL_NUM IS NULL) THEN
            NEW.SERIAL_NUM = GEN_ID(GEN_SERIALS, 1);
    END
...
```

NEW

Доступно: PSQL, только в триггерах

Изменено: 1.5, 2.0

Описание: Контекстная переменная NEW содержит новую версию записи базы данных, которая была вставлена или обновлена. Начиная с Firebird 2.0 эта переменная в триггерах после события (AFTER) доступна только для чтения.

Тип: Совпадает с типом данных строки

Примечание

В триггерах на несколько типов событий (добавлены в Firebird 1.5) контекстная переменная NEW доступна всегда. Но при срабатывании триггера по событию DELETE новой версии записи не создаётся, т.е. чтение переменной NEW всегда будет возвращать значение NULL, а запись в неё вызовет исключение времени выполнения.

'NOW'

Доступно: DSQL, ESQL, PSQL

Изменено: 2.0

Описание: 'NOW' является не переменной, а строковой константой. Однако, при её приведении CAST() к типу даты/времени, Вы получите текущую дату и/или время. В версиях до Firebird 2.0 дробная часть всегда была “.0000”, т.е. давая точность до секунды. Начиная с Firebird 2.0 точность увеличена до трёх знаков после запятой, т.е. до миллисекунд. 'NOW' нечувствительна к регистру и при приведении типов игнорируются начальные и конечные пробелы.

Тип: CHAR(3)

Примеры:

```
SELECT 'NOW'
        FROM RDB$DATABASE
-- возвращает 'NOW'

SELECT CAST('NOW' AS DATE)
```

```
        FROM RDB$DATABASE
-- возвращает, например 2008-08-13

SELECT CAST('NOW' AS TIME)
        FROM RDB$DATABASE
-- возвращает, например 14:20:19.6170

SELECT CAST('NOW' AS TIMESTAMP)
        FROM RDB$DATABASE
-- возвращает, например 2008-08-13 14:20:19.6170
```

Короткий синтаксис (Сокращённое приведение типов даты и времени (datetime)) для последних трёх запросов:

```
SELECT DATE 'NOW'
        FROM RDB$DATABASE
SELECT TIME 'NOW'
        FROM RDB$DATABASE
SELECT TIMESTAMP 'NOW'
        FROM RDB$DATABASE
```

Примечания:

- При использовании приведения типов CAST() 'NOW' всегда возвращает фактическую дату/время, даже в модулях PSQL, где CURRENT_DATE, CURRENT_TIME и CURRENT_TIMESTAMP возвращают одно и то же значение вплоть до окончания выполнения кода. Это делает 'NOW' полезным для измерения временных интервалов в триггерах, процедурах и исполнимых блоках;
- При использовании сокращенного синтаксиса 'NOW' оценивается во время синтаксического анализа и её значение замораживается до тех пор, пока запрос остаётся подготовленным - даже при его многократном выполнении! Это, чтобы Вы были в курсе;
- Если Вам не требуется прогрессирующих значений в PSQL или надо получить постоянное значение во время многократных выполнений запроса, то использование CURRENT_DATE, CURRENT_TIME и CURRENT_TIMESTAMP обычно предпочтительней, чем использование 'NOW'. Имейте в виду, что точность по умолчанию у CURRENT_TIME - единицы секунды; чтобы получить точность вплоть до миллисекунд используйте CURRENT_TIME (3).

OLD

Доступно: PSQL, только в триггерах

Изменено: 1.5, 2.0

Описание: Контекстная переменная OLD содержит существующую версию записи базы данных перед удалением или обновлением. Начиная с Firebird 2.0 эта переменная доступна только для чтения.

Тип: Совпадает с типом данных строки

Примечание

В триггерах на несколько типов событий (добавлены в Firebird 1.5) контекстная переменная OLD доступна всегда. Очевидно, что при срабатывании триггера по событию INSERT нет никакой существующей ранее версии записи. В этой ситуации переменная OLD всегда будет возвращать NULL; попытка записи запись в неё вызовет исключение на этапе выполнения.

ROW_COUNT

Доступно: PSQL

Добавлено: 1.5

Изменено: 2.0

Описание: Контекстная переменная ROW_COUNT содержит число строк, затронутых последним оператором DML (INSERT, UPDATE, DELETE, SELECT или FETCH) в текущем триггере, хранимой процедуре или исполняемом блоке.

Тип: INTEGER

Пример:

```
UPDATE FIGURES
    SET NUMBER = 0
    WHERE ID = :ID;
IF (ROW_COUNT = 0) THEN
    INSERT INTO FIGURES (ID, NUMBER)
```

```
VALUES (:ID, 0);
```

Поведение с SELECT и FETCH:

- После выполнения singleton SELECT запроса (запроса, который может вернуть не более одной строки данных), ROW_COUNT равна 1, если была получена строка данных и 0 в противном случае;
- В цикле FOR SELECT переменная ROW_COUNT увеличивается на каждой итерации (начиная с 0 в качестве первого значения);
- После выборки (FETCH) из курсора, ROW_COUNT равна 1, если была получена строка данных и 0 в противном случае. Выборка нескольких записей из одного курсора *не* увеличивает ROW_COUNT после 1;
- В Firebird 1.5 ROW_COUNT всегда равна 0 после выполнения любого оператора SELECT.

Примечание

Контекстная переменная ROW_COUNT не может быть использована для определения количества строк, затронутых при выполнении операторов EXECUTE STATEMENT или EXECUTE PROCEDURE.

SQLCODE

Доступно: PSQL

Добавлено: 1.5

Изменено: 2.0

Устарело: 2.5.1

Описание: В блоках обработки ошибок "WHEN ... DO” контекстная переменная SQLCODE содержит текущий код ошибки SQL. До Firebird 2.0 значение SQLCODE можно было получить только в блоках обработки ошибок WHEN SQLCODE и WHEN ANY. Теперь она может быть отлична от нуля в блоках WHEN GDSCODE и WHEN EXCEPTION при условии, что ошибка, вызвавшее срабатывание блока, соответствует коду ошибки SQL. Вне обработчиков ошибок SQLCODE всегда равен 0, а вне PSQL не существует вообще.

Тип: INTEGER

Пример:

```
WHEN ANY DO
    BEGIN
        IF (SQLCODE <> 0) THEN
            MSG = 'Обнаружена ошибка SQL!';
        ELSE
            MSG = 'Ошибки нет!';
        EXCEPTION EX_CUSTOM MSG;
    END
```

Важное замечание: SQLCODE в настоящее время устарела, т.к. введён SQL-2003-совместимый код состояния SQLSTATE . Поддержка SQLCODE и WHEN SQLCODE будет прекращена в следующих версиях Firebird.

SQLSTATE

Доступно: PSQL

Добавлено: 2.5.1

Описание: В блоках обработки ошибок "WHEN ... DO" контекстная переменная SQLSTATE переменная содержит 5 символов SQL-2003-совместимого кода состояния, переданного оператором, вызвавшим ошибку. Вне обработчиков ошибок SQLSTATE всегда равен '00000', а вне PSQL не существует вообще.

Тип: CHAR(5)

Пример:

```
WHEN ANY DO
    BEGIN
        MSG = CASE SQLSTATE
            WHEN '22003' THEN
                'Число вышло за пределы диапазона!'
            WHEN '22012' THEN
                'Деление на ноль!'
            WHEN '23000' THEN
                'Нарушение ограничения целостности!'
            ELSE 'Ошибок нет! SQLSTATE = ' || SQLSTATE;
            END;
        EXCEPTION EX_CUSTOM MSG;
```

```
END
```

Примечания:

- SQLSTATE предназначен для замены SQLCODE. Последняя в настоящее время устарела и буден удалёна будущих версиях Firebird;
- Firebird не поддерживает (пока) синтаксис "WHEN SQLSTATE ... DO ". Используйте обработчик блока WHEN ANY для проверки значения переменной SQLSTATE;
- Любой код SQLSTATE состоит из двух символов класса и трёх символов подкласса. Класс 00 (успешное выполнение), 01 (предупреждение) и 02 (нет данных) представляют собой *условия завершения*. Каждый код статуса вне этих классов является *исключением*. Поскольку классы 00, 01 и 02 не вызывают ошибку, они никогда не будут обнаруживаться в переменной SQLSTATE;
- Полный список кодов SQLSTATE приведён в *Приложении к Firebird 2.5 Release Notes*.

UPDATING

Доступно: PSQL

Добавлено: 1.5

Описание: Контекстная переменная UPDATING доступна только в триггерах и указывает на то, что триггер сработал в результате выполнения операции UPDATE. Может использоваться в триггерах на различные типы событий (Триггеры на несколько типов событий).

Тип: логический (boolean)

Пример:

```
...
IF (INSERTING OR UPDATING) THEN
        BEGIN
        IF (NEW.SERIAL_NUM IS NULL) THEN
                NEW.SERIAL_NUM = GEN_ID(GEN_SERIALS, 10);
END
...
```

Глава 12

Операторы и предикаты

Разрешено использование NULL как операнда

Изменено: 2.0

Описание: До Firebird 2.0 большинство операторов и предикатов не позволяло использовать NULL литералы в качестве операндов. Сравнения или операции, например, такие как "A <> NULL", "B + NULL" или "NULL <ANY (...)" отклонялись синтаксическим анализатором (парсером). Теперь это разрешено использовать почти всюду, но необходимо знать следующее:

Подавляющее большинство из этих разрешённых выражений возвращает значение NULL независимо от состояния или значения другого операнда и, следовательно, бесполезны вообще для практических задач.

В частности, не пытайтесь определить, что значением поля или переменной является NULL, путем тестирования на равенство "= NULL" или неравенство "<> NULL". Всегда используйте "IS [NOT] NULL.

Предикаты: Также теперь разрешено использование NULL литералов в предикатах IN, ANY / SOME и ALL, хотя раньше в большинстве случаев этого делать было нельзя. Здесь также нет никакого практического преимущества, но ситуация немного более сложная - предикаты с NULL не всегда возвращают результат NULL. Более подробно это изложено в статье *Firebird Null Guide* в разделе “Предикаты”.

|| - конкатенация (сцепление) строк

Доступно: DSQL, ESQL, PSQL

Конкатенация текстовых BLOB

Изменено: 2.1

Описание: Начиная с Firebird 2.1 разрешено использовать оператор конкатенации || для полей типа BLOB любой длины и с любым набором символов. При конкатенации полей типа BLOB и обычных полей результатом будет строкой.Если в операции конкатенации присутствуют текстовые и двоичные поля BLOB, то результатом будет двоичный BLOB.

Результат типа VARCHAR или BLOB

Изменено: 2.0, 2.1

Описание: До Firebird 2.0 в качестве результата конкатенации строк использовался тип CHAR(n). В Firebird 2.0 тип результата конкатенации строк был изменён на VARCHAR(n). В результате максимальная длина результата конкатенации стала 32765 вместо прежних 32767. Начиная с Firebird 2.1 если по крайней мере один из операндов типа BLOB, то и результат также типа BLOB и ограничение максимальной длины конкатенации не действует. При конкатенации строк (BLOB не участвует) тип результата будет VARCHAR(n) с максимумом в 32765 байт.

Проверка переполнения

Изменено: 1.0, 2.0

Описание: В версиях Firebird 1.x вызывалась ошибка, если сумма объявленных строковых длин операндов (по их объявленным в БД значениям) при конкатенации превышала 65535 байтов, даже если фактический результат укладывался в максимально допустимый размер 32767 байта. Начиная с Firebird 2.0 объявленные строковые длины операндов не берутся во внимание и, соответственно, ошибка не возникает. Только при превышении фактическим результатом значения 32765 байтов (новый лимит для результатов конкатенации) будет вызвана ошибка.

ALL

Доступно: DSQL, ESQL, PSQL

Разрешено использование NULL

Изменено: 2.0

Описание: В предикате ALL теперь разрешено использовать NULL в качестве проверочного значения. Заметьте, что это не приносит практической пользы. Даже если весь набор будет заполнен NULL и используется оператор “=”, то предикат будет возвращать значение NULL, а не TRUE.

UNION как подзапрос

Изменено: 2.0

Описание: В подзапросе с предикатом ALL теперь можно использовать UNION.

ANY/SOME

Доступно: DSQL, ESQL, PSQL

Разрешено использование NULL

Изменено: 2.0

Описание: В предикате ANY и SOME теперь разрешено использовать NULL в качестве проверочного значения. Заметьте, что это не приносит практической пользы. В частности тестовое значение NULL не будут считать равным NULL в результирующем наборе подзапроса.

UNION как подзапрос

Изменено: 2.0

Описание: В подзапросе с предикатом ANY или SOME теперь можно использовать UNION.

IN

Доступно: DSQL, ESQL, PSQL

Разрешено использование NULL

Изменено: 2.0

Описание: В предикате IN теперь разрешено использовать NULL литералы - и как тестовое значение и в списке. Заметьте, что это не приносит практической пользы. В частности, "NULL IN (..., NULL, ..., ...)" не вернет значение TRUE, а "NULL NOT IN (..., NULL, ..., ...)" не вернется значение FALSE.

UNION как подзапрос

Изменено: 2.0

Описание: В подзапросе с предикатом IN теперь можно использовать UNION.

IS [NOT] DISTINCT FROM

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Два операнда считают различными (DISTINCT), если они имеют различные значения, или если одно из них - NULL, и другое нет. Они считаются NOT DISTINCT (равными), если имеют одинаковые значения или оба имеют значение NULL.

Тип результата: Boolean (логический)

Синтаксис:

op1 IS [NOT] DISTINCT FROM *op2*

Пример:

```
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 всегда возвращает TRUE или FALSE и никогда NULL (неизвестное значение). Операторы «=» и «<>», наоборот, вернут NULL, если один или оба операнда имеют значение NULL. См. также таблицу 12.1.

Таблица 12.1. **Сравнение IS [NOT] DISTINCT с “=” и “<>”**

Характеристика операнда	**Результаты различных операторов**			
	=	**NOT DISTINCT**	<>	**DISTINCT**
Одинаковые значения	TRUE	TRUE	FALSE	FALSE
Различные значения	FALSE	FALSE	TRUE	TRUE
Оба NULL	NULL		NULL	FALSE
Одно NULL	NULL		NULL	TRUE

NEXT VALUE FOR

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Возвращает следующее значение в последовательности (SEQUENCE). SEQUENCE является SQL-совместимым термином генератора в InterBase и Firebird. NEXT VALUE FOR полностью эквивалентен GEN_ID (..., 1) и начиная с Firebird 2.0 является рекомендуемым синтаксисом.

Синтаксис:

NEXT VALUE FOR *sequence-name*

Пример:

NEW.CUST_ID = NEXT VALUE FOR CUSTSEQ;

NEXT VALUE FOR не поддерживает приращение значения, отличное от 1. Если требуется другое значение шага, то используйте старую функцию GEN_ID.

См. также: CREATE SEQUENCE , GEN_ID()

SIMILAR TO

Доступно: DSQL, PSQL

Добавлено: 2.5

Описание: Оператор SIMILAR TO проверяет соответствие строки с шаблоном регулярного выражения SQL. В отличие от некоторых других языков для успешного выполнения шаблон должен соответствовать всей строке — соответствие подстроки не достаточно. Если один из операндов имеет значение NULL, то и результат будет NULL. В противном случае результат является TRUE или FALSE.

Тип результата: Boolean (логический)

Синтаксис SIMILAR TO:

string-expression [NOT] SIMILAR TO <*pattern*> [ESCAPE <*escape-char*>]

<*pattern*> ::= регулярное выражения SQL
<*escape-char*> ::= одиночный символ

Синтаксис регулярных выражений SQL (SQL regular expressions): Следующий синтаксис определяет формат регулярного выражения SQL. Это - полное и корректное его определение. Кроме того, он является весьма формальным и довольно длинным и, вероятно, озадачивает тех, кто не имеет некоторого опыта работы с регулярными выражениями (или с очень формальными и довольно длинными нисходящими определениями). Не стесняйтесь пропустить его и начать

читать следующий раздел, Создание регулярных выражений , использующий подход от простого к сложному.

<*regular expression*> ::= <*regular term*> ['|' <*regular term*> …]

<*regular term*> ::= <*regular factor*> …

<*regular factor*> ::= <*regular primary*> [<*quantifier*>]

<*quantifier*> ::= ?
| *
| +
| '{' <*m*> [,[<*n*>]] '}'

<*m*>, <*n*> ::= целое положительное число с <*m*> <= <*n*> если оба присутствуют

<*regular primary*> ::= <*character*>
| <*character class*>
| %
| (<*regular expression*>)

<*character*> ::= <*escaped character*>
| <*non-escaped character*>

<*escaped character*> ::= <*escape-char*> <*special character*>
| <*escape-char*> <*escape-char*>

<*special character*> ::= любой из символов []()|^-+*%_?{

<*non-escaped character*> ::= любой символ за исключением <*special character*> и не эквивалентный <*escape-char*> (если задан)

<*character class*> ::= '_'
| '[' <*member*> ... ']'
| '[^' <*non-member*> ... ']'
| '[' <*member*> ... '^' <*non-member*> ... ']'

<*member*>, <*non-member*> ::= <*character*>
| <*range*>
| <*predefined class*>

<*range*> ::= <*character*>-<*character*>

<*predefined class*> ::= '[:' <*predefined class name*> ':]'

<*predefined class name*> ::= ALPHA | UPPER | LOWER | DIGIT
| ALNUM | SPACE | WHITESPACE

Создание регулярных выражений

Символы

В регулярных выражениях большинство символов представляет сами себя. Единственное исключение - специальные символы (*special character*):

[ ] ( ) | ^ - + * % _ ? {

и управляющие символы, если они заданы.

Регулярному выражению, не содержащему специальных или управляющих символов, соответствует только полностью идентичные строки (в зависимости от используемой сортировки). То есть это функционирует точно так же, как оператор “=”:

```
'Apple' SIMILAR TO 'Apple'      -- TRUE
'Apples' SIMILAR TO 'Apple'     -- FALSE
'Apple' SIMILAR TO 'Apples'     -- FALSE
'APPLE' SIMILAR TO 'Apple'      — в зависимости от сортировки
```

Шаблоны

Известным SQL шаблонам _ и % соответствует любой единственный символ и строка любой длины, соответственно:

```
'Birne' SIMILAR TO 'B_rne'      -- TRUE
'Birne' SIMILAR TO 'B_ne'       -- FALSE
'Birne' SIMILAR TO 'B%ne'       -- TRUE
'Birne' SIMILAR TO 'Bir%ne%'    -- TRUE
'Birne' SIMILAR TO 'Birr%ne'    -- FALSE
```

Обратите внимание, что шаблон % также соответствует пустой строке.

Классы символов

Набор символов, заключённый в квадратные скобки определяют класс символов. Символ в строке соответствует классу в шаблоне, если символ является элементом класса.:

```
'Citroen' SIMILAR TO 'Cit[arju]oen'        -- TRUE
'Citroen' SIMILAR TO 'Ci[tr]oen'           -- FALSE
'Citroen' SIMILAR TO 'Ci[tr][tr]oen'       -- TRUE
```

Как видно из второй строки классу только соответствует единственный символ, а не их последовательность.

Два символа, соединенные дефисом, в определении класса определяют диапазон. Диапазон для активного сопоставления включает в себя эти два конечных символа и все символы, находящиеся между ними. Диапазоны могут быть помещены в любом месте в определении класса без специальных разделителей, чтобы сохранить в классе и другие символы.

```
'Datte' SIMILAR TO 'Dat[q-u]e'             -- TRUE
'Datte' SIMILAR TO 'Dat[abq-uy]e'          -- TRUE
'Datte' SIMILAR TO 'Dat[bcg-km-pwz]e'      -- FALSE
```

В определении класса также могут использоваться следующие предопределенные классы символов:

[:ALPHA:]
Латинские буквы a...x и A...Z. С регистро-нечувствительными сортировками этот класс также включает подчёркнутые символы.

[:DIGIT:]
Десятичные цифры 0...9.

[:ALNUM:]
Объединение [:ALPHA:] и [:DIGIT:].

[:UPPER:]
Прописные (в верхнем регистре) латинские буквы A...Z. Также включает в себя нижний регистр при регистро-нечувствительных сортировках и подчёркнутые символы при регистро-нечувствительных сортировках.

[:LOWER:]
Строчные (в нижнем регистре) латинские буквы a...z. Также включает в себя нижний регистр при регистро-нечувствительных сортировках и подчёркнутые

символы при регистро-нечувствительных сортировках.

[:SPACE:]
Символ пробела (ASCII 32).

[:WHITESPACE:]
Вертикальная табуляции (ASCII 9), перевод строки (ASCII 10), горизонтальная табуляция (ASCII 11), разрыв страницы (ASCII 12), возврат каретки (ASCII 13) и пробел (ASCII 32).

Включение в оператор SIMILAR TO предопределенного класс имеет тот же эффект, как и включение всех его элементов. Использование предопределенных классов допускается *только в пределах* определения класса. Если Вам нужно сопоставление только с предопределенным классом и ничего больше, то поместите дополнительную пару скобок вокруг него.

```
'Erdbeere' SIMILAR TO 'Erd[[:ALNUM:]]eere'          -- TRUE
'Erdbeere' SIMILAR TO 'Erd[[:DIGIT:]]eere'          -- FALSE
'Erdbeere' SIMILAR TO 'Erd[a[:SPACE:]b]eere'        -- TRUE
'Erdbeere' SIMILAR TO [[:ALPHA:]]                   -- FALSE
'E' SIMILAR TO [[:ALPHA:]] -- TRUE
```

Если определение класса запускается со знаком вставки (^), то всё, что следует за ним, исключается из класса. Все остальные символы проверяются.

```
'Framboise' SIMILAR TO 'Fra[^ck-p]boise'            -- FALSE
'Framboise' SIMILAR TO 'Fr[^a][^a]boise'            -- FALSE
'Framboise' SIMILAR TO 'Fra[^[:DIGIT:]]boise'       -- TRUE
```

Если знак вставки (^) находится не в начале последовательности, то класс включает в себя все символы *до* него и исключает символы *после* него.

```
'Grapefruit' SIMILAR TO 'Grap[a-m^f-i]fruit'        -- TRUE
'Grapefruit' SIMILAR TO 'Grap[abc^xyz]fruit'        -- FALSE
'Grapefruit' SIMILAR TO 'Grap[abc^de]fruit'         -- FALSE
'Grapefruit' SIMILAR TO 'Grap[abe^de]fruit'         -- FALSE

'3' SIMILAR TO '[[:DIGIT:]^4-8]'                    -- TRUE
'6' SIMILAR TO '[[:DIGIT:]^4-8]'                    -- FALSE
```

Наконец, уже упомянутый подстановочный знак “_” является собственным классом символов, соответствуя любому единственному символу.

Кванторы

Вопросительный знак сразу после символа или класса указывает на то, что для соответствия предыдущий элемент может произойти 0 или 1 раз:

```
'Hallon' SIMILAR TO 'Hal?on'                -- FALSE
'Hallon' SIMILAR TO 'Hal?lon'               -- TRUE
'Hallon' SIMILAR TO 'Halll?on'              -- TRUE
'Hallon' SIMILAR TO 'Hallll?on'             -- FALSE
'Hallon' SIMILAR TO 'Halx?lon'              -- TRUE
'Hallon' SIMILAR TO 'H[a-c]?llon[x-z]?'     -- TRUE
```

Звёздочка (*) сразу после символа или класса указывает на то, что для соответствия предыдущий элемент может произойти 0 или более раз:

```
'Icaque' SIMILAR TO 'Ica*que'               -- TRUE
'Icaque' SIMILAR TO 'Icar*que'              -- TRUE
'Icaque' SIMILAR TO 'I[a-c]*que'            -- TRUE
'Icaque' SIMILAR TO '_*'                    -- TRUE
'Icaque' SIMILAR TO '[[:ALPHA:]]*'          -- TRUE
'Icaque' SIMILAR TO 'Ica[xyz]*e'            -- FALSE
```

Знак плюс (+) сразу после символа или класса указывает на то, что для соответствия предыдущий элемент может произойти 1 или более раз:

```
'Jujube' SIMILAR TO 'Ju_+'                  -- TRUE
'Jujube' SIMILAR TO 'Ju+jube'               -- TRUE
'Jujube' SIMILAR TO 'Jujuber+'              -- FALSE
'Jujube' SIMILAR TO 'J[jux]+be'             -- TRUE
'Jujube' SIMILAR TO 'J[[:DIGIT:]]+ujube'    -- FALSE
```

Если символ или класс сопровождаются числом, заключённым в фигурные скобки, то для соответствия нужно повторение элемента *точно в это число раз*:

```
'Kiwi' SIMILAR TO 'Ki{2}wi'                 -- FALSE
'Kiwi' SIMILAR TO 'K[ipw]{2}i'              -- TRUE
'Kiwi' SIMILAR TO 'K[ipw]{2}'               -- FALSE
'Kiwi' SIMILAR TO 'K[ipw]{3}'               -- TRUE
```

Если число сопровождается запятой, то для соответствия нужно повторение элемента *как минимум в это число раз*:

```
'Limone' SIMILAR TO 'Li{2,}mone'            -- FALSE
```

```
'Limone' SIMILAR TO 'Li{1,}mone'          -- TRUE
'Limone' SIMILAR TOto 'Li[nezom]{2,}'     -- TRUE
```

Если фигурные скобки содержат два числа (m и n), разделённые запятой, и второе число больше первого, то для соответствия элемент должен быть повторен, как минимум, m раз и не больше m+1 раз:

```
'Mandarijn' SIMILAR TO 'M[a-p]{2,5}rijn'     -- TRUE
'Mandarijn' SIMILAR TO 'M[a-p]{2,3}rijn'     --  FALSE
'Mandarijn' SIMILAR TO 'M[a-p]{2,3}arijn'    -- TRUE
```

Кванторы ?, * и + сокращение для {0,1}, {0,} и {1.}, соответственно.

Термин ИЛИ

В условиях регулярных выражений можно использовать оператор ИЛИ (|). Соответствие произошло, если строка параметра соответствует по крайней мере одному из условий:

```
'Nektarin' similar to 'Nek|tarin'          -- FALSE
'Nektarin' similar to 'Nektarin|Persika'   -- TRUE
'Nektarin' similar to 'M_+|N_+|P_+'        -- TRUE
```

Подвыражения

Одна или более частей регулярного выражения могут быть сгруппированы в подвыражения (также называемые подмасками). Для этого их нужно заключить в круглые скобки:

```
'Orange' SIMILAR TO 'O(ra|ri|ro)nge'          --  TRUE
'Orange' SIMILAR TO 'O(r[a-e])+nge'           --  TRUE
'Orange' SIMILAR TO 'O(ra){2,4}nge'           -- FALSE
'Orange' SIMILAR TO 'O(r(an|in)g|rong)?e'     --  TRUE
```

Экранирование специальных символов

Для исключения из процесса сопоставления специальных символов (которые часто встречаются в регулярных выражениях) их надо экранировать. Специальных символов экранирования по умолчанию нет — их при необходимости определяет пользователь:

```
'Peer (Poire)' SIMILAR TO 'P[^ ]+ \(P[^ ]+\)' ESCAPE '\'        --  TRUE
'Pera [Pear]' SIMILAR TO 'P[^ ]+ #[P[^ ]+#]' ESCAPE '#'        --  TRUE
'Paron-Appledryck' SIMILAR TO 'P%$-A%' ESCAPE '$'        --  TRUE
'Parondryck' SIMILAR TO 'P%--A%' ESCAPE '-'                     -- FALSE
```

SOME

Смотрите раздел ANY/SOME .

Глава 13

Агрегатные функции

Агрегатные функции воздействуют на группы записей, а не на индивидуальные записи или переменные. Они часто используются в сочетании с предложением GROUP BY.

LIST()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Функция LIST возвращает строку, состоящую из NOT NULL значений аргументов в группе, разделенной запятой (по умолчанию) или заданным пользователями разделителем. Если нет никаких NOT NULL значений (это включает случай, когда группа пуста), возвращается значение NULL.

Тип результата: BLOB

Синтаксис:

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

- С использованием ALL (значение по умолчанию) будут перечислены все NOT NULL значения аргумента. С DISTINCT – из результата удаляются дубликаты, кроме случая с аргументом типа BLOB;
- Начиная с Firebird 2.5 опциональный аргумент *разделителя* (*separator*) может быть любым строковым выражением. Это позволяет указать в качестве разделителя, например, ASCII_CHAR(13). Это улучшение также внесено в версию Firebird 2.1.4;
- Аргументы *expression* и *separator* поддерживают блобы (BLOB) любого размера и с любым набором символов;
- Дата/время и численные аргументы неявно преобразовываются в строки до объединения;

- Результатом функции является текстовый BLOB, за исключением использования в качестве аргумента BLOB других подтипов;
- Упорядочение списка значений не определяется.

MAX()

Доступно: DSQL, ESQL, PSQL

Добавлено: IB

Изменено: 2.1

Описание: Функция MAX возвращает максимальное значение аргумента в группе. Если аргумент является строкой, то это значение, которое приходит последним при использовании активной сортировки.

Тип результата: Зависит от типа аргумента.

Синтаксис:

MAX (*expression*)

- Если группа пустая или содержит только значения NULL, то результатом будет NULL;
- Начиная с Firebird 2.1 функция полностью поддерживает текстовые блобы (BLOB) любой размерности и с любым набором символов.

MIN()

Доступно: DSQL, ESQL, PSQL

Добавлено: IB

Изменено: 2.1

Описание: Функция MIN возвращает минимальное значение аргумента в группе. Если аргумент является строкой, то это значение, которое приходит первым при использовании активной сортировки.

Тип результата: Зависит от типа аргумента.

Синтаксис:

MIN (*expression*)

- Если группа пустая или содержит только значения NULL, то результатом будет NULL;
- Начиная с Firebird 2.1 функция полностью поддерживает текстовые блобы (BLOB) любой размерности и с любым набором символов.

Глава 14

Встроенные функции

Примечание переводчика

Важно

Если в Вашей базе данных имя декларированной внешней функции совпадает с именем встроенной, то будет вызываться внешняя функция. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ABS()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает абсолютное значение аргумента.

Тип результата: Числовой.

Синтаксис:

ABS (*number*)

Важно

Если в Вашей базе данных декларирована внешняя функция ABS(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ACOS()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает арккосинус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

ACOS (*number*)

- Результат - угол в диапазоне [0, π] радиан;
- Если аргумент выходит за диапазон [-1, 1], то возвращается NaN.

Важно

Если в Вашей базе данных декларирована внешняя функция ACOS(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ASCII_CHAR()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает ASCII символ, соответствующий числу, переданному в качестве аргумента.

Тип результата: [VAR]CHAR(1) CHARACTER SET NONE

Синтаксис:

ASCII_CHAR (<*code*>)

<*code*> ::= целое число в диапазоне [0..255]

Важно

- Если в Вашей базе данных декларирована внешняя функция ASCII_CHAR, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION ;
- Если Вы привыкли к поведению внешней функции ASCII_CHAR, которая

возвращает пустую строку, если аргумент равен 0, то, пожалуйста, обратите внимание на то, что внутренняя функция корректно возвращает символ с ASCII кодом 0.

ASCII_VAL()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает ASCII код, соответствующий символу, переданному в качестве аргумента.

Тип результата: SMALLINT

Синтаксис:

ASCII_VAL (*ch*)

ch ::= a [VAR]CHAR или текстовый BLOB с максимальным размером 32767 байта

- Если аргументов является строка длиной более одного символа, то возвратится ASCII код первого символа;
- Если аргумент — пустая строка, то возвратится 0;
- Если аргумент — NULL, то возвратится NULL;
- Если первый символ аргумента — многобайтный, то вызовется ошибка. В версиях Firebird 2.1-2.13 и 2.5 ошибка вызывается, если любой символ в строке многобайтный. Этот баг исправлен в версиях Firebird 2.1.4 и 2.5.1;

Важно

Если в Вашей базе данных декларирована внешняя функция ASCII_VAL(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ASIN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает арксинус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

ASIN (*number*)

- Результат - угол в диапазоне [-π/2, π/2] радиан;
- Если аргумент выходит за диапазон [-1, 1], то возвращается NaN.

Важно

Если в Вашей базе данных декларирована внешняя функция ASIN(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ATAN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает арктангенс аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

ATAN (*number*)

- Результат - угол в диапазоне [-π/2, π/2] радиан;
- Если аргумент выходит за диапазон [-1, 1], то возвращается NaN.

Важно

Если в Вашей базе данных декларирована внешняя функция ATAN(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

ATAN2()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает угол как *отношение* синуса к косинусу, аргументы у которых задаются этими двумя параметрами, а *знаки* синуса и косинуса соответствуют знакам параметров. Это позволяет получать результаты по всей окружности, включая углы -π/2 и π/2.

Синтаксис:

ATAN2 (*y*, *x*)

- Результат - угол в диапазоне [-π, π] радиан;
- Если x отрицательный, то при нулевом значении y результат равен π, а при значении -0 равен -π;
- Если и y и x равны 0, то результат бессмыслен. Начиная с Firebird 3.0 в этом случае будет вызываться ошибка.

Важно

Если в Вашей базе данных декларирована внешняя функция ATAN2(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

Примечания:

- Полностью эквивалентное описание этой функции следующее: ATAN2 (y, x) является углом между положительной осью X и линией от начала координат до точки (x, y). Это также делает очевидным, что значение ATAN2 (0, 0) не определено;
- Если x больше, чем 0, ATAN2 (y, x) совпадает с ATAN (y/x);
- Если известны и синус, и косинус угла , то ATAN2 (SIN, COS) возвращает угол.

BIN_AND()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает результат побитовой операции AND (И) над аргументом (аргументами).

Тип результата: INTEGER или BIGINT

Синтаксис:

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

Важно

Если в Вашей базе данных декларирована внешняя функция BIN_AND(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

BIN_OR()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает результат побитовой операции OR (ИЛИ) над аргументом (аргументами).

Тип результата: INTEGER или BIGINT

Синтаксис:

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

Важно

Если в Вашей базе данных декларирована внешняя функция BIN_OR(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

BIN_SHL()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает первый параметр, побитово смещённый влево на значение второго параметра, т.е. a << b или 2^b.

Тип результата: BIGINT

Синтаксис:

BIN_SHL (*number*, *shift*)

BIN_SHR()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает первый параметр, побитово смещённый влево на значение второго параметра, т.е. a >> b или a/2^b.

Тип результата: BIGINT

Синтаксис:

BIN_SHR (*number*, *shift*)

- Выполняемая операция является арифметическим сдвигом вправо (SAR), а это означает, что знак первого операнда всегда сохраняется.

BIN_XOR()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает результат побитовой операции XOR над аргументом (аргументами).

Тип результата: INTEGER или BIGINT

Синтаксис:

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

Важно

Если в Вашей базе данных декларирована внешняя функция BIN_XOR, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

BIT_LENGTH()

Доступно: DSQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Возвращает длину входной строки в битах. Для многобайтных наборов символов результат может быть в 8 раз меньше, чем количество символов в "формальном" числе байт на символ, записанном в RDB$CHARACTER_SETS.

Примечание

С параметрами типа CHAR эта функция берет во внимание всю формальную строковую длину (например, объявленная длина поля или переменной). Если Вы хотите получить "логическую" длину в битах, не считая пробелов, то перед передачей аргумента в BIT_LENGTH надо выполнить над ним операцию RIGHT TRIM.

Тип результата: INTEGER

Синтаксис:

BIT_LENGTH (*str*)

Поддержка блобов (BLOB): Начиная с Firebird 2.1 функция полностью поддерживает текстовые блобы (BLOB) любой размерности и с любым набором символов.

Примеры:

```
SELECT BIT_LENGTH('Hello!')
    FROM RDB$DATABASE
-- возвращает 48

SELECT BIT_LENGTH(_ISO8859_1 'Grüß Di!')
    FROM RDB$DATABASE
-- возвращает 64: каждый, и ü, и ß занимают один байт в ISO8859_1

SELECT BIT_LENGTH(
    CAST (_ISO8859_1 'Grüß di!' AS VARCHAR(24)
    CHARACTER SET UTF8))
FROM RDB$DATABASE
-- возвращает 80: каждый, и ü, и ß занимают по два байта в UTF8

SELECT BIT_LENGTH (
    CAST (_ISO8859_1 'Grüß di!' AS CHAR(24)
    CHARACTER SET UTF8))
FROM RDB$DATABASE
-- возвращает 208: размер всех 24 позиций CHAR и два из них
16-битные
```

См. также: OCTET_LENGTH() , CHAR_LENGTH(), CHARACTER_LENGTH()

CAST()

Доступно: DSQL, ESQL, PSQL

Добавлено:

Изменено: 2.0, 2.1, 2.5

Описание: Функция CAST преобразует выражение в требуемый тип данных или домен. Если преобразование невозможно, то возникает ошибка.

Тип результата: Определяется пользователем

Синтаксис:

```
CAST (expression AS <target_type>)

<target_type> ::= sql_datatype
                | [TYPE OF] domain
                | TYPE OF COLUMN relname.colname
```

Короткий синтаксис:

Альтернативный синтаксис, поддерживаемый только при приведении к типу строковым типам данных выражений типов DATE, TIME или TIMESTAMP:

```
datatype 'date/time string'
```

Этот синтаксис был доступен уже в InterBase, но должным образом никогда не документировался. *Пожалуйста, обратите внимание*: Сокращенный синтаксис оценивается сразу во время синтаксического анализа, в результате чего значения остаются теми же до тех пор, пока результаты выполнения оператора не будут подтверждены или отменены. Для даты, например, '12-AUG-2012' это не имеет никакого значения (*Примечание переводчика: за исключением случая участия операции приведения в выходных данных и подготовки запроса до полуночи, а его подтверждения после полуночи*). Но для псевдо-переменных 'NOW', 'YESTERDAY', 'TODAY' и 'TOMORROW' это не верно (Вы не всегда получите ожидаемое значение). Если вам нужно точное текущее значение, то используйте полный синтаксис CAST ().

Примеры:

Полный синтаксис:

```
SELECT CAST ('12' || '-JUNE-' || '1991' AS DATE)
    FROM RDB$DATABASE
```

Сокращённый синтаксис приведения строки к дате:

```
UPDATE PEOPLE
    SET AGECAT = 'Old'
WHERE BIRTHDATE < DATE '1-JAN-1943'
```

Заметьте, что Вы можете отбросить даже короткий синтаксис из примера выше — сервер сам поймет из контекста, как интерпретировать строку (определив,

сравнение происходит с полем типа DATE):

```
UPDATE PEOPLE
    SET AGECAT = 'Old'
WHERE BIRTHDATE < '1-JAN-1943'
```

Но это не всегда возможно. В примере ниже короткий синтаксис приведения не может быть отброшен - в противном случае сервер посчитал бы, что из строки вычитается целое число:

```
SELECT DATE 'TODAY' — 7
    FROM RDB$DATABASE
```

В таблице 14.1 показано, какие преобразования типов можно делать при помощи функции CAST.

Таблица 14.1 Допустимые преобразования типов для функции CAST

Из типа	В тип
Числовые типы	Числовые типы [VAR]CHAR BLOB
[VAR]CHAR BLOB	[VAR]CHAR BLOB Числовые типы DATE TIME TIMESTAMP
DATE TIME	[VAR]CHAR BLOB TIMESTAMP
TIMESTAMP	[VAR]CHAR BLOB DATE TIME

Имейте в виду, что иногда информация не будет потеряна, например, когда вы приводите тип TIMESTAMP к типу DATE. Кроме того, тот факт, что типы CAST-совместимы сам по себе не гарантирует, что преобразование будет успешным. Например, "CAST (123456789 как SMALLINT)", безусловно, приведет к ошибке, так же будет и с "CAST ("Добрый день!" AS DATE)".

Преобразование входных полей: Начиная с Firebird 2.0 Вы можете

преобразовывать входные параметры к заданному типу данных:

```
CAST (? AS INTEGER)
```

Это дает контроль над типом входного поля, создаваемого сервером. Заметьте, что для преобразования параметров всегда требуется полного синтаксис – краткий не поддерживаются.

Преобразование к домена или его типу: Начиная с Firebird 2.1 поддерживается преобразование к домену или его базовому типу. При приведении к домену, учитываются любые объявленные для него ограничения (NOT NULL и/или CHECK), иначе преобразование не удастся. Имейте в виду, что проверка проходит, если оно имеет значение TRIE или NULL! Таким образом, учитывая вышеприведённые утверждения:

```
CREATE DOMAIN QUINT AS INTEGER 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)
```

только результат первого преобразования приведёт к ошибке.

Когда используется TYPE OF, выражение преобразуется к базовому типу домена, игнорируя любые ограничения. С доменом QUINT, определенным в примере выше, следующие два преобразования эквивалентны и оба успешно выполнятся:

```
SELECT CAST (2000 AS TYPE OF QUINT) FROM RDB$DATABASE
SELECT CAST (2000 AS INTEGER) FROM RDB$DATABASE
```

При использовании TYPE OF с типом (VAR)CHAR набор символов и сортировка сохраняются:

```
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;
-- возвратит Deze -> Die -> deze -> die
```

```
SELECT CAST(ZIN AS TYPE OF DUNL20) FROM ZINNEN ORDER BY 1;
-- возвратит deze -> Deze -> die -> Die
```

Предупреждение

При изменении определения домена существующие преобразования в этот домен или его типы может стать недействительным. Если эти преобразования происходят в модулях PSQL, то эти ошибки можно обнаружить (см. Приложении A, раздел Поле RDB$VALID_BLR).

Преобразование к типу столбца: Начиная с Firebird 2.5 поддерживается преобразование к типу столбца таблицы или представления. Используется только сам тип столбца; в случае строковых типов это включает набор символов и сортировку. Ограничения и значения по умолчанию исходного столбца не применяются.

```
CREATE TABLE TTT (
        S VARCHAR(40)
                CHARACTER SET UTF8 COLLATE UNICODE_CI_AI);
COMMIT;

SELECT CAST ('Jag har många vänner' AS TYPE OF COLUMN TTT.S)
        FROM RDB$DATABASE;
```

Предупреждение

- Для текстовых типов, также как и при преобразовании к домену, сохраняются набор символов и сортировка. Однако, из-за ошибки при выполнении сравнения (например, проверки на равенство) сортировки учитываются не всегда. Если сортировка имеет значение, то Вы должны тщательно протестировать свой код перед внедрением! Эта ошибка исправлена в Firebird 3.0;
- При изменении определения типа столбца существующие преобразования в его тип может стать недействительным. Если эти преобразования происходят в модулях PSQL, то эти ошибки можно обнаружить (см. Приложении A, раздел Поле RDB$VALID_BLR).

В случаях, где сопоставление имеет значение, полностью протестируйте код перед развертыванием!

Преобразование к типу BLOB: Успешное преобразование в и из BLOB доступно начиная с Firebird 2.1.

CEIL(), CEILING()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает наименьшее целое число, большее или равное аргументу.

Тип результата: BIGINT или DOUBLE PRECISION

Синтаксис:

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

Важно

Если в Вашей базе данных декларирована внешняя функция CEILING(), то она перекрывает внутреннюю функцию (не затрагивая CEIL()). Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

См. также: FLOOR()

CHAR_LENGTH(), CHARACTER_LENGTH()

Доступно: DSQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Возвращает длину (в символах) входной строки.

Примечание

С параметрами типа CHAR эта функция берет во внимание всю формальную строковую длину (например, объявленная длина поля или переменной). Если Вы хотите получить "логическую" длину без учёта пробелов, то перед передачей аргумента в CHAR[ACTER]_LENGTH надо выполнить над ним операцию RIGHT TRIM.

Тип результата: INTEGER

Синтаксис:

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

Поддержка BLOB: Начиная с Firebird 2.1 эти функции полностью поддерживают обработку полей типа BLOB любой длины и с любым набором символов.

Пример:

```
SELECT CHAR_LENGTH('Hello!')
FROM RDB$DATABASE
-- Результат 6

SELECT CHAR_LENGTH(_ISO8859_1 'Grüß di!')
FROM RDB$DATABASE
-- Результат 8

SELECT CHAR_LENGTH (CAST (_ISO8859_1 Grüß di I!' AS VARCHAR(24) CHARACTER SET UTF8))
FROM RDB$DATABASE
-- Результат 8; То, сто символы ü и ß занимают по два байта не имеет значения.

SELECT CHAR_LENGTH (CAST (_ISO8859_1 Grüß di!' AS CHAR(24) CHARACTER SET UTF8))
FROM RDB$DATABASE
-- Результат 24: Все позиции CHAR(24)
```

См. также: BIT_LENGTH() , OCTET_LENGTH()

CHAR_TO_UUID()

Доступно: DSQL, PSQL

Добавлено: 2.5

Описание: Преобразует читабельную 36-символьную строку UUID к соответствующему 16-байтовому UUID.

Тип результата: CHAR(16) CHARACTER SET OCTETS

Синтаксис:

CHAR_TO_UUID (ascii_uuid)

Пример:

```
SELECT CHAR_TO_UUID('A0bF4E45-3029-2a44-D493-4998c9b439A3')
FROM RDB$DATABASE
-- Результат A0BF4E4530292A44D4934998C9B439A3 (16-байтовая строка)

SELECT
CHAR_TO_UUID('A0bF4E45-3029-2A44-X493-4998c9b439A3')
FROM RDB$DATABASE
-- ошибка: -читабельный аргумент UUID для CHAR_TO_UUID должен
-- должен иметь шестнадцатеричную цифру в позиции 20 вместо "X (ASCII 88)"
```

См. также: UUID_TO_CHAR(), GEN_UUID()

COALESCE()

Доступно: DSQL, PSQL

Добавлено: 1.5

Описание: Функция COALESCE принимает два или более аргумента возвращает значение первого NOT NULL аргумента. Если все аргументы имеют значение NULL, то и результат будет NULL.

Тип результата: в зависимости от входных аргументов

Синтаксис:

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

Пример:

```
SELECT COALESCE(
```

```
        PE.NICKNAME, PE.FIRSTNAME, 'Mr./Mrs.') ||
        ' ' || PE.LASTNAME  FULLNAME
    FROM PERSONS PE
```

В данном примере предпринимается попытка использовать все имеющиеся данные для составления полного имени. Выбирается поле NICKNAME из таблицы PERSONS. Если оно имеет значение NULL, то берётся значение из поля FIRSTNAME. Если и оно имеет значение NULL, то используется строка 'Mr./Mrs.'. Затем к значению функции COALESCE фамилия (поле LASTNAME). Обратите внимание, что эта схема нормально работает, только если выбираемые поля имеют значение NULL или не пустое значение: если одно из них является пустой строкой, то именно оно и возвратится в качестве значения функции COALESCE.

Примечание

В Firebird 1.0.x, где функция COALESCE не доступна, Вы можете в этих целях использовать внешнюю функцию *nvl.

COS()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает косинус угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

COS (*angle*)

Любой NOT-NULL результат находится в диапазоне [-1, 1].

Важно

Если в Вашей базе данных декларирована внешняя функция cos, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

COSH()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает гиперболический косинус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

COSH (*angle*)

Любой NOT-NULL результат находится в диапазоне [1, + ∞].

Важно

Если в Вашей базе данных декларирована внешняя функция cosh, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

COT()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает котангенс угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

COT (*angle*)

Важно

Если в Вашей базе данных декларирована внешняя функция cot, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна

внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

DATEADD()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Добавляет заданное число лет, месяцев, недель, дней, часов, минут, секунд или миллисекунд к значению даты/времени (параметр WEEK — неделя — добавлен в версии 2.5).

Тип результата: DATE, TIME или TIMESTAMP

Синтаксис:

```
DATEADD (<args>)

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

<amount> ::= <datetime> ::= целое выражение (отрицательное вычитается)

<unit> ::= YEAR | MONTH | WEEK | DAY
        | HOUR | MINUTE | SECOND | MILLISECOND

<datetime> ::= DATE, TIME или TIMESTAMP выражение
```

- Тип результата определяется третьим аргументом;
- С аргументом типа TIMESTAMP и DATE можно использовать любую составляющую даты/времени (<unit>) (До Firebird 2.5 для типа данных DATE было запрещено использовать приращения меньше дня);
- Для типа данных TIME разрешается использовать только HOUR, MINUTE, SECOND и MILLISECOND .

Примеры:

```
DATEADD (28 DAY TO CURRENT_DATE)
DATEADD (-6 HOUR TO CURRENT_TIME)
DATEADD (MONTH, 9, DATEOFCONCEPTION)
DATEADD (-38 WEEK TO DATEOFBIRTH)
DATEADD (MINUTE, 90, TIME 'NOW')
DATEADD (? YEAR TO DATE '11-SEP-1973')
```

DATEDIFF()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Возвращает количество лет, месяцев, недель, дней, часов, минут, секунд или миллисекунд между двумя значениями даты/времени.

Тип результата: BIGINT

Синтаксис:

```
DATEDIFF (<args>)

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

<unit> ::= <momentN> ::= YEAR | MONTH | WEEK | DAY
      | HOUR | MINUTE | SECOND | MILLISECOND

<datetime> ::= DATE, TIME или TIMESTAMP выражение
```

- Параметры DATE и TIMESTAMP могут использоваться совместно. Совместное использование типа TIME с типами DATE и TIMESTAMP не разрешается;
- С аргументом типа TIMESTAMP и DATE можно использовать любую составляющую даты/времени (<unit>) (До Firebird 2.5 для типа данных DATE было запрещено использовать приращения меньше дня);
- Для типа данных TIME разрешается использовать только HOUR, MINUTE, SECOND и MILLISECOND.

Вычисление:

- Функция DATEDIFF не проверяет разницу в более мелких составляющих даты/времени, чем задана в первом аргументе (<unit>). В результате получаем;
  - DATEDIFF (YEAR, DATE '1-JAN-2009', DATE '31-DEC-2009') = 0, но
  - DATEDIFF (YEAR, DATE '31-DEC-2009', DATE '1-JAN-2010') = 1
- Однако для *более мелких* составляющих даты/времени имеем:
  - DATEDIFF (DAY, DATE '26-JUN-1908', DATE '11-SEP-1973') = 23818
  - DATEDIFF (DAY, DATE '30-NOV-1971', DATE '8-JAN-1972') = 39
- Отрицательное значение функции говорит о том, что дата/время в <moment2> меньше, чем в <moment1>.

Примеры:

```
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()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Функция DECODE эквивалентна конструкции «*Простой CASE* », в которой заданное выражение сравнивается с другими выражениями до нахождения совпадения. Результатом является значение, указанное после выражения, с которым найдено совпадение. Если совпадений не найдено, то возвращается значение по умолчанию (если оно, конечно, задано — в противном случае возвращается NULL).

Тип результата: различный

Синтаксис:

```
DECODE(<test-expr>,
    <expr>, result
    [,<expr>, result …]
```

[, *defaultresult*])

Эквивалентная конструкция CASE

```
CASE <test-expr>
    WHEN <expr> THEN result
    [WHEN <expr> THEN result …]
[ELSE defaultresult]
END
```

Внимание

Совпадение эквивалентно оператору «=», т.е. если <*test-expr*> имеет значение NULL, то он не соответствует ни одному из <expr>, даже тем, которые имеют значение NULL.

Пример:

```
SELECT
    NAME, AGE,
    DECODE(UPPER(SEX),
        'M', 'Мужской',
        'F', 'Женский',
        'Х3'),
    RELIGION
FROM PEOPLE
```

См. Также: Конструкция CASE , Простой CASE

EXP()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Функция EXP возвращает натуральную экспоненту (e^{number})

Тип результата: DOUBLE PRECISION

Синтаксис:

```
EXP(number)
```

Важно

Если в Вашей базе данных декларирована внешняя функция EXP(), то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

См. также: LN()

EXTRACT()

Доступно: DSQL, ESQL, PSQL

Добавлено: IB 6

Изменено: 2.1

Описание: Извлекает и возвращает составляющую даты/времени из выражнеий с типом DATE, TIME и TIMESTAMP. Введено в InterBase 6, но не описано в его Language Reference.

Тип результата: SMALLINT или NUMERIC

Синтаксис:

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

<part> ::= YEAR | MONTH | WEEK
         | DAY | WEEKDAY | YEARDAY
         | HOUR | MINUTE | SECOND | MILLISECOND

<datetime> ::= выражение с типом данных DATE, TIME или
               TIMESTAMP
```

Типы и диапазоны результата приведены в таблице 14.2. Если составляющая даты/времени не присутствует в аргументе дата/время (например SECOND в аргументе с типом DATE или YEAR в TIME), то функция вызовет ошибку.

Таблица 14.2. Типы и диапазоны результатов оператора EXTRACT

Составляющая даты/времени	Тип	Диапазон	Комментарий
YEAR	SMALLINT	1–9999	
MONTH	SMALLINT	1–12	
WEEK	SMALLINT	1–53	
DAY	SMALLINT	1–31	
WEEKDAY	SMALLINT	0–6	0 = Воскресенье
YEARDAY	SMALLINT	0–365	0 = 1 января
HOUR	SMALLINT	0–23	
MINUTE	SMALLINT	0–59	
SECOND	NUMERIC(9,4)	0.0000–59.9999	Включает в себя миллисекунды
MILLISECOND	NUMERIC(9,1)	0.0–999.9	Возвращает неправильный результат в версиях Firebird 2.1 и 2.1.1

Пример:

```
SELECT
    SUBSTRING( EXTRACT(DAY FROM CURRENT_DATE) +
        100 FROM 2 FOR 2) || '.' ||
    SUBSTRING( EXTRACT(MONTH FROM CURRENT_DATE) +
        100 FROM 2 FOR 2) || '.' ||
    SUBSTRING( EXTRACT(YEAR FROM CURRENT_DATE)
        FROM 1 FOR 4) || ' г.'
FROM RDB$DATABASE
```

MILLISECOND

Добавлено: 2.1 (с ошибкой)

Исправлено: 2.1.2

Описание: Начиная с версии Firebird 2.1 можно извлекать миллесекунды из аргументов с типом TIME и NIMESTAMP. Возвращаемый ти данных — NUMERIC(9, 1).

Примечание

При извлечении миллесекунд из контекстной переменной CURRENT_TIME, которая по умолчанию имеет точность до секунды, результатом будет 0. С параметрами типа CHAR эта функция берет во внимание всю формальную строковую длину (например, объявленная длина поля или переменной). Для извлечения миллесекунд используйте CURRENT_TIME(3) или CURRENT_TIMESTAMP.

WEEK

Добавлено: 2.1

Описание: Начиная с версии Firebird 2.1 можно извлекать номер недели из аргумента с типом данных DATE или TIMESTAMP. В соответствии со стандартом ISO-8601 неделя начинается с понедельника и всегда включает в себя 7 дней. Первой неделей года является первая неделя, у которой в ней больше дней в новом году (по крайней мере 4): дни 1-3 могут принадлежать предыдущей неделе (52 или 53) прошлого года. По аналогии дни 1-3 текущего года могут принадлежать 1 неделе следующего года.

Внимание

Будьте внимательны при объединении результатов извлечения WEEK и YEAR. Например, 30 декабря 2008 года принадлежит первой неделе 2009 года. Т. е. EXTRACT (WEEK FROM DATE '30 DEC 2008') вернёт 1. а YEAR всегда вернёт календарный год (в данном примере 2008). В этом случае результаты YEAR и WEEK противоречат друг другу. То же самое происходит, если первые дни января принадлежат последней неделе предыдущего года.

Обратите внимание также на несоответствие WEEKDAY стандарту ISO-8601: для воскресенья результатом будет 0, тогда как по стандарту ISO-8601 результат определён как 7.

FLOOR()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает наибольшее целое число, меньшее или равное аргументу.

Тип результата: BIGINT или DOUBLE PRECISION

Синтаксис:

FLOOR(number)

Важно

Если в Вашей базе данных декларирована внешняя функция floor, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

См. также: CEIL(), CEILING()

GEN_ID()

Доступно: DSQL, ESQL, PSQL

Добавлено: IB

Описание: Увеличивает значение генератора или последовательности и возвращает новое значение. Начиная с версии Firebird 2.0 доступно использование SQL-совместимого оператора NEXT VALUE FOR (за исключением случая увеличения значения на число, отличное от 1).

Тип результата: BIGINT

Синтаксис:

GEN_ID (*generator-name*, <*step*>)
<*step*> ::= целое число

Пример:

```
NEW.MY_TABLE_ID = GEN_ID(GEN_ MY_TABLE_ID, 1);
```

Предупреждение

Использование GEN_ID с отрицательным приращением может поставить

под угрозу целостность ваших данных. Используйте отрицательное приращение только в том случае, если Вы точно уверены, что это не приведёт к потере целостности данных.

GEN_UUID()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5.2

Описание: Возвращает универсальный уникальный идентификатор ID в виде 16-байтной строки символов. До версии Firebird 2.5.2 это была полностью случайная строка, что не отвечало требованиям стандарта RFC-4122. Начиная с версии 2.5.2 функция возвращает строку UUID 4-ой версии, где несколько битов зарезервированы, а остальные являются случайными.

Тип результата: CHAR(16) CHARACTER SET OCTETS

Синтаксис:

```
GEN_UUID()
```

Пример:

```
SELECT GEN_UUID() FROM RDB$DATABASE
```

– Результат (до версии Firebird 2.5.2):
017347BFE212B2479C00FA4323B36320 (16-байтная строка)

– Результат (начиная с версии Firebird 2.5.2):
XXXXXXXX-XXXX-4XXX-YXXX-XXXXXXXXXXXX
где 4 это номер версии, а Y может принимать значение 8, 9, A или B .

См. Также: CHAR_TO_UUID() , UUID_TO_CHAR()

HASH()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает хэш-значение входной строки. Эта функция полностью поддерживает текстовые BLOB любой длины и с любым набором символов.

Тип результата: BIGINT

Синтаксис:

HASH (*string*)

IIF()

Доступно: DSQL, PSQL

Добавлено: 2.0

Описание: Функция IIF имеет три аргумента. Если первый аргумент является истиной, то результатом будет второй параметр, в противном случае результатом будет третий параметр.

Тип результата: Зависит от входных параметров

Синтаксис:

IIF (<*condition*>, *ResultT*, *ResultF*)

<*condition*> ::= булевское выражение

Пример:

```
SELECT IIF( SEX = 'M', 'Sir', 'Madam' ) FROM CUSTOMERS
```

По сути IIF(*Cond*, *Result1*, *Result2*) это короткая запись оператора “CASE WHEN *Cond* THEN *Result1* ELSE *Result2* END“. Оператор IIF также можно сравнить в тройным оператором "?:" в C-подобных языках.

LEFT()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает левую часть строкового параметра. Количество символов задается вторым параметром.

Тип результата: VARCHAR или BLOB

Синтаксис:

LEFT (*string*, *length*)

- Функция поддерживает текстовые BLOB любой длины, в том числе и с многобайтными наборами символов;
- Если строковый параметр типа BLOB, то и результат будет типа BLOB. В противном случае результат будет типа VARCHAR(n), где n равно длине строкового параметра;
- Если числовой параметр больше длины строкового параметра, то результатом будет строковый параметр;
- Если числовой параметр не является целым числом, то используется банковское округление: 0.5 становится 0, 1.5 становится 2, 2.5 становится 2, 3.5 становится 4 и т.д.

См. также: RIGHT()

LN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает натуральный логарифм аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

LN(number)

- При отрицательном или нулевом аргументе функция вернёт ошибку.

Важно

Если в Вашей базе данных декларирована внешняя функция ln, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

См. также: EXP()

LOG()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Возвращает логарифм *y* (второй аргумент) по основанию *x* (первый аргумент).

Тип результата: DOUBLE PRECISION

Синтаксис:

```
LOG(x, y)
```

- Если один из аргументов меньше или равен 0, то возникает ошибка. До версии Firebird 2,5 это приведет к выдаче NaN (Not-a-Number — не число), ± INF (infinity - бесконечность) или 0 в зависимости от точного значения аргументов;
- Если оба аргумента равны 1, то результатом функции будет NaN;
- Если $x = 1$ и $y < 1$, то результатом функции будет -INF (-∞);
- Если $x = 1$ и $y > 1$, то результатом функции будет +INF (+∞).

Важно

Если в Вашей базе данных декларирована внешняя функция log, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

LOG10()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5

Описание: Возвращает десятичный логарифм входного аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
LOG10(number)
```

- Если входной аргумент отрицательный или равен 0, возникает ошибка. До версии Firebird 2,5 это приведет к выдаче NaN и -INF (-∞), соответственно.

Важно

Если в Вашей базе данных декларирована внешняя функция log10, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

LOWER()

Доступно: DSQL, ESQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Возвращает входную строку в нижнем регистре. Точный результат зависит от набора символов. Например, для наборов символов ASCII и NONE только ASCII символы приводятся к нижнему регистру; для набора символов OCTETS вся строка возвращается без изменений. Начиная с версии Firebird 2.1 полностью поддерживаются тестовые BLOB с любым набором символов и любой длины.

Тип результата: VAR(CHAR) или BLOB

Синтаксис:

LOWER (*string*)

Примечание

Так как LOWER является зарезервированным словом, то встроенная функция имеет приоритет перед внешней функцией с таким же именем (если таковая объявлена в базе данных). Для вызова внешней функции (даже если она объявлена в нижнем регистре) используйте двойные кавычки и написание имени внешней функции в верхнем регистре: “LOWER(*string*)”.

Пример:

```
SELECT SHERIFF
        FROM TOWNS
WHERE LOWER(NAME) = 'cooper''s valley'
```

См. также: UPPER()

LPAD()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5 (портировано также в версию 2.1.4)

Описание: Дополняет входную строку слева пробелами или определённой пользователем строкой до заданной длины.

Тип результата: VARCHAR или BLOB

Синтаксис:

LPAD(*string, endlength [, padstring]*)

- Функция полностью поддерживает тестовые BLOB с любым набором символов и любой длины;
- Если входная строка имеет тип BLOB, то и результат будет иметь тип BLOB.

В противном случае тип результата будет VARCHAR(*endlength*);

- Если аргумент *padstring* задан и равен '' (пустой строке), то дополнения строки не происходит;
- Если аргумент *endlength* меньше длины входной строки, то происходит её усечение до длины *endlength*, даже если аргумент *padstring* является пустой строкой.

Важно

Если в Вашей базе данных декларирована внешняя функция lpad, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

Примечание

В версиях Firebird 2.1-2.1.3 все результаты не BLOB типа имели тип VARCHAR(32765), что делало желательным приведение его к более меньшему размеру. Это поведение было исправлено в боле старших версиях.

Примеры:

```
LPAD ('Hello', 12)                -- Результат: ' Hello'
LPAD ('Hello', 12, '-')           -- Результат: '-------Hello'
LPAD ('Hello', 12, '')            -- Результат: 'Hello'
LPAD ('Hello', 12, 'abc')         -- Результат: 'abcabcaHello'
LPAD ('Hello', 12, 'abcdefghij')  -- Результат: 'abcdefgHello'
LPAD ('Hello', 2)                 -- Результат: 'He'
LPAD ('Hello', 2, '')             -- Результат: 'He'
LPAD ('Hello', 2, '-')            -- Результат: 'He'

SELECT
    LPAD(EXTRACT(DAY FROM CURRENT_DATE), 2, 0) || '.' ||
    LPAD(EXTRACT(MONTH FROM CURRENT_DATE), 2, 0) || '.' ||
    EXTRACT(YEAR FROM CURRENT_DATE) || ' г.'
FROM RDB$DATABASE
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

См. также: RPAD()

MAXVALUE()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает максимальное значение из входного списка чисел, строк или параметров с типом DATE/TIME/TIMESTAMP. Функция полностью поддерживает текстовые BLOB с любым набором символов и любой длины.

Тип результата: варьируется

Синтаксис:

```
MAXVALUE(expr [, expr ...])
```

- Если один или более входных параметров имеют значение NULL, то результатом функции MAXVALUE тоже будет NULL в отличие от агрегатной функции MAX.

См. также: MINVALUE()

MINVALUE()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает минимальное значение из входного списка чисел, строк или параметров с типом DATE/TIME/TIMESTAMP. Функция полностью поддерживает текстовые BLOB с любым набором символов и любой длины.

Тип результата: варьируется

Синтаксис:

```
MINVALUE(expr [, expr ...])
```

- Если один или более входных параметров имеют значение NULL, то результатом функции MINVALUE тоже будет NULL в отличие от агрегатной

функции MIN.

См. также: MAXVALUE()

MOD()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает остаток от целочисленного деления.

Тип результата: INTEGER или BIGINT

Синтаксис:

```
MOD(a, b)
```

- Вещественные числа округляются до выполнения деления. Так, результатом “7.5 mod 2.5” будет 2 (8 mod 3), а не 0.

Важно

Если в Вашей базе данных декларирована внешняя функция mod, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

NULLIF()

Доступно: DSQL, PSQL

Добавлено: 1.5

Описание: Функция NULLIF возвращает значение первого аргумента, если он не равен второму. При равенстве аргументов возвращается NULL.

Тип результата: Зависит от входных параметров.

Синтаксис:

```
NULLIF(<expr1>, <expr2>)
```

Пример:

```
SELECT AVG( NULLIF(CA.WEIGHT, -1) )
    FROM CARGO CA
```

Этот запрос вернет средний вес груза из таблицы CARGO, за исключением имеющих вес -1, так как агрегатная функция AVG пропускает данные NULL. Предполагается, что значение -1 по смыслу означает "вес груза неизвестен". При использовании простого оператора AVG(CA.WEIGHT) учитываются все значения, в том числе и -1, то есть результат будет неверным.

Примечание

В версиях Firebird 1.0.x функция NULLIF отсутствует, но Вы можете использовать внешнюю функцию *nullif.

См. также: COALESCE()

OCTET_LENGTH()

Доступно: DSQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Возвращает длину входной строки в байтах (октетах). Для многобайтных наборов символов результат может быть меньше, чем количество символов в "формальном" числе байт на символ, записанном в RDB$CHARACTER_SETS.

Примечание

С параметрами типа CHAR эта функция берет во внимание всю формальную строковую длину (например, объявленная длина поля или переменной). Если Вы хотите получить "логическую" длину в байтах без учёта пробелов, то перед передачей аргумента в OCTET_LENGTH надо выполнить над ним операцию RIGHT TRIM.

Тип результата: INTEGER

Синтаксис:

```
OCTET_LENGTH (str)
```

Поддержка BLOB: Начиная с версии Firebird 2.1 функция полностью поддерживает текстовые BLOB с любым набором символов и любой длины.

Примеры:

```
SELECT OCTET_LENGTH('Hello!')
    FROM RDB$DATABASE
-- Результат 6

SELECT OCTET_LENGTH(_ISO8859_1 'Grüß di!')
    FROM RDB$DATABASE
-- Результат 8: ü и ß имеют размер 1 байт в наборе символов ISO8859_1

SELECT OCTET_LENGTH( CAST(_ISO8859_1 'Grüß di!' AS
        VARCHAR(24) CHARACTER SET UTF8))
    FROM RDB$DATABASE
-- Результат 10: ü и ß имеют размер 2 байта в наборе символов UTF8

SELECT OCTET_LENGTH(CAST (_ISO8859_1 'Grüß di!' AS CHAR(24)
        CHARACTER SET UTF8))
    FROM RDB$DATABASE
-- Результат 26: все 24 позиции CHAR плюс 2 дополнительных байта на два 2-байтовых символа
```

См. также: BIT_LENGTH() , CHAR_LENGTH(), CHARACTER_LENGTH()

OVERLAY()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Заменяет часть строки другой строкой. По умолчанию число удаляемых из строки символов равняется длине заменяющей строки. Дополнительный четвертый параметр позволяет пользователю задать своё число символов, которые будут удалены.

Тип результата: VARCHAR или BLOB

Синтаксис:

OVERLAY (*string* PLACING *replacement* FROM *pos* [FOR *length*])

- Функция полностью поддерживает тестовые BLOB с любым набором символов и любой длины;
- Если входная строка имеет тип BLOB, то и результат будет иметь тип BLOB. В противном случае тип результата будет VARCHAR(*n*), где *n* является суммой длин параметров *string* и *replacement*;
- Как и во всех строковых функциях SQL параметр *pos* является определяющим;
- Если *pos* больше длины *string*, то *replacement* помещается сразу после окончания строки;
- Если число символов от *pos* до конца *string* меньше, чем длина *replacement* (или, чем параметр *length*, если он задан), то строка усекается до значения *pos* и *replacement* помещается после него;
- При нулевом параметре *length* («FOR 0") *replacement* просто вставляется в *string* начиная с позиции *pos*;
- Если любой из параметров имеет значение NULL, то и результат будет NULL;
- Если параметры *pos* и *length* не являются целым числом, то используется банковское округление: 0.5 становится 0, 1.5 становится 2, 2.5 становится 2, 3.5 становится 4 и т.д.

Примеры:

```
overlay ('Goodbye' placing 'Hello' from 2)   -- Результат: 'Ghelloe'
overlay ('Goodbye' placing 'Hello' from 5)   -- Результат: 'GoodHello'
overlay ('Goodbye' placing 'Hello' from 8)   -- Результат: 'GoodbyeHello'
overlay ('Goodbye' placing 'Hello' from 20) -- Результат: 'GoodbyeHello'

overlay ('Goodbye' placing 'Hello' from 2 for 0)   – Результат: 'GHellooodbye'
overlay ('Goodbye' placing 'Hello' from 2 for 3)   -- Результат: 'GHellobye'
overlay ('Goodbye' placing 'Hello' from 2 for 6)   -- Результат: 'GHello'
overlay ('Goodbye' placing 'Hello' from 2 for 9)   -- Результат: 'Ghello'

overlay ('Goodbye' placing '' from 4)              -- Результат: 'Goodbye'
overlay ('Goodbye' placing '' from 4 for 3)        -- Результат: 'Gooe'
overlay ('Goodbye' placing '' from 4 for 20)       -- Результат: 'Goo'
```

```
overlay (" placing 'Hello' from 4)          -- Результат: 'Hello'
overlay (" placing 'Hello' from 4 for 0)    -- Результат: 'Hello'
overlay (" placing 'Hello' from 4 for 20)   -- Результат: 'Hello'
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

См. также: REPLACE()

PI()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает приближение значение числа π.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
PI ()
```

Важно

Если в Вашей базе данных декларирована внешняя функция pi, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

POSITION()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает позицию первого вхождения подстроки в строку.

Дополнительный (опционально) третий аргумент задаёт позицию в строке, с которой начинается поиск подстроки, тем самым игнорируются любые вхождения подстроки в строку до этой позиции. Если совпадение не найдено, то результат равен 0.

Тип результата: INTEGER

Синтаксис:

POSITION (<*args*>)

<*args*> ::= *substring* IN *string*
| *substring*, *string* [, *startpos*]

- Опциональный третий параметр поддерживается только вторым вариантом синтаксиса (синтаксис с запятой);
- Пустую строку считают подстрокой любой строки. Поэтому при входном параметре *substring*, равном '' (пустая строка), и при параметре *string*, отличном от NULL, результатом будет:
  - 1, если параметр *startpos* не задан;
  - *startpos*, если *startpos* не превышает длину параметра *string*;
  - 0, если *startpos* превышает длину параметра *string*.

  Замечание: В версиях Firebird 2.1-2.1.3 и 2.5 функция *всегда* возвращала 1, если параметр *substring* являлся пустой строкой. Исправлена начиная с версий 2.1.4 и 2.5.1.
- Функция полностью поддерживает тестовые BLOB с любым набором символов и любой длины.

Примеры:

```
POSITION('be' in 'To be or not to be')      -- Результат: 4
POSITION('be', 'To be or not to be')        -- Результат: 4
POSITION('be', 'To be or not to be', 4)     -- Результат: 4
POSITION('be', 'To be or not to be', 8)     -- Результат: 17
POSITION('be', 'To be or not to be', 18)    -- Результат: 0
POSITION('be' in 'Alas, poor Yorick!')      -- Результат: 0
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

POWER()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает *x* в степени *y*.

Тип результата: DOUBLE PRECISION

Синтаксис:

POWER(*x*, *y*)

Важно

Если в Вашей базе данных декларирована внешняя функция dpower, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

RAND()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает случайное число в диапазоне от 0 до 1.

Тип результата: DOUBLE PRECISION

Синтаксис:

RAND()

Важно

Если в Вашей базе данных декларирована внешняя функция rand, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

RDB$GET_CONTEXT()

Примечание

RDB$GET_CONTEXT и RDB$SET_CONTEXT на самом деле являются заранее объявленными UDF. Они приведены здесь как внутренние функции, потому что они всегда присутствуют – пользователю не надо ничего делать, чтобы сделать их доступными.

Доступно: DSQL, ESQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Возвращает значение контекстной переменной контекста от одного из пространства имен - SYSTEM, USER_SESSION или USER_TRANSACTION.

Тип результата: VARCHAR(255)

Синтаксис:

RDB$GET_CONTEXT ('<*namespace*>', '<*varname*>')

<*namespace*> ::= SYSTEM | USER_SESSION | USER_TRANSACTION
<*varname*> ::= регистрочувтвительная строка с максимальной длиной 80 символов

Пространство имён:
USER_SESSION и USER_TRANSACTION изначально пусты. Пользователь может создать и установить значение переменных в них функцией RDB$SET_CONTEXT () и получить их значения из функции RDB$GET_CONTEXT (). Пространство имен SYSTEM - только для чтения. Оно содержит много предопределенных переменных, показанных в таблице 14.3.

Таблица 14.3. **Контекстные переменные в пространстве имён SYSTEM**

DB_NAME	Полный путь к базе данных или — если подключение через путь запрещено — псевдоним

	(алиас).
NETWORK_PROTOCOL	Протокол, используемый в соединении: 'TCPv4', 'WNET', 'XNET' или NULL
CLIENT_ADDRESS	Для TCPv4 — IP адрес, для XNET — локальный ID процесса. Для всех остальных протоколов переменная имеет значение NULL
CURRENT_USER	Глобальная переменная CURRENT_USER
CURRENT_ROLE	Глобальная переменная CURRENT_ROLE
SESSION_ID	Глобальная переменная CURRENT_CONNECTION
TRANSACTION_ID	Глобальная переменная CURRENT_TRANSACTION
ISOLATION_LEVEL	Уровень изоляции текущей транзакции - CURRENT_TRANSACTION: 'READ COMMITTED', 'SNAPSHOT' или 'CONSISTENCY'
ENGINE_VERSION	Версия сервера Firebird. Добавлена в версии 2.1
CLIENT_PID	PID процесса на клиентском компьютере. . Добавлена в версии 2.5.3
CLIENT_PROCESS	Полный путь к клиентскому приложению, подключившемуся к базе данных. Позволяет не использовать системную таблицу MON$ATTACHMENTS (поле MON$REMOTE_PROCESS). Добавлена в версии 2.5.3

Возвращаемые значения и ошибки: Если запрошенная переменная существует в данном пространстве имен, то будет возвращено её значение в виде строки с максимальной длиной в 255 символов. Обращение к несуществующему пространству имён или несуществующей переменной в пространстве имен SYSTEM приведёт к ошибке. Если Вы опрашиваете несуществующую переменную в одном из других пространств имен, функция вернёт NULL. Пространство имен и имя переменной регистрочувствительны, должны быть непустыми строками и заключены в кавычки.

Примеры:

```
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'))
```

См. также: RDB$SET_CONTEXT()

RDB$SET_CONTEXT()

Примечание

RDB$GET_CONTEXT и RDB$SET_CONTEXT на самом деле являются заранее объявленными UDF. Они приведены здесь как внутренние функции, потому что они всегда присутствуют – пользователю не надо ничего делать, чтобы сделать их доступными.

Доступно: DSQL, ESQL, PSQL

Добавлено: 2.0

Изменено: 2.5.3

Описание: Создает, устанавливает значение или обнуляет переменную в одном из используемых пользователями для записи пространстве имён: USER_SESSION и USER_TRANSACTION.

Тип результата: INTEGER

Синтаксис:

```
RDB$SET_CONTEXT ('<namespace>', '<varname>', <value> | NULL)
```

<*namespace*> ::= USER_SESSION | USER_TRANSACTION
<*varname*> ::= Регистрочувствительная строка с максимальной длиной 80 символов
<*value*> ::= Значение любого типа, при условии что его можно привести к типу VARCHAR (255)

Пространство имён:
Пространства имён USER_SESSION и USER_TRANSACTION изначально пусты. Пользователь может создать переменную и установить её значение с

помощью RDB$SET_CONTEXT() - и получать её значение с помощью RDB$GET_CONTEXT(). Контекст USER_SESSION связан с текущим соединением. Переменные в USER_TRANSACTION существуют только в рамках транзакции, в которой они были созданы. При завершении транзакции (неважно, при её подтверждении или отмене контекст и все переменные, созданные в ней, уничтожаются.

Возвращаемые значения и ошибки:

Функция возвращает 1, если переменная уже существовала до вызова и 0, если не существовала. Для удаления переменной надо установить её значение в NULL. Если данное пространство имен не существует, то функция вернёт ошибку. Пространство имен и имя переменной регистрочувствительны, должны быть непустыми строками и заключены в кавычки.

Примеры:

```
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
```

Примечания:

- Максимальное число переменных в рамках одного соединения равно 1000;
- Все переменные в пространстве имён USER_TRANSACTION сохраняются при ROLLBACK RETAIN или ROLLBACK TO SAVEPOINT, независимо от того, в какой точке во время выполнения транзакции они были установлены;
- Так как RDB$SET_CONTEXT является UDF, то можно – но только в PSQL – вызвать её, не присваивая результат, как в приведённом выше втором примере. Обычные встроенные функции не допускают такого использования.

В версии Firedird 2.5.3 в пространство имён были добавлены ещё две переменные для текущей транзакции: LOCK_TIMEOUT (время ожидания подтверждения транзакции при блокировке в секундах) и READ_ONLY (делает текущую транзакцию только читающей).

См. также: RDB$GET_CONTEXT()

REPLACE()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Заменяет все вхождения подстроки в строке.

Тип результата: VARCHAR или BLOB

Синтаксис:

REPLACE (*str*, *find*, *repl*)

- Функция полностью поддерживает тестовые BLOB с любым набором символов и любой длины;
- Если входная строка имеет тип BLOB, то и результат будет иметь тип BLOB. В противном случае тип результата будет VARCHAR(*n*), где *n* является суммой длин параметров *str*, *find* и *repl*, вычисленной таким образом, что даже максимально возможное число замен не будет приводить к переполнению поля;
- Если *find* является пустой строкой, то результатом будет неизменённая входная строка *str*;
- Если *repl* является пустой строкой, то из входной строки *str* будут удалены все вхождения *find*;
- Если один из аргументов является NULL, то и результатом будет NULL, даже если ничего не было заменено.

Примеры:

```
REPLACE ('Billy Wilder', 'il', 'oog')      -- Результат: 'Boogly Woogder'
REPLACE ('Billy Wilder', 'il', '')         -- Результат: 'Bly Wder'
REPLACE ('Billy Wilder', NULL, 'oog')      -- Результат: NULL
REPLACE ('Billy Wilder', 'il', NULL)       -- Результат: NULL
REPLACE ('Billy Wilder', 'xyz', NULL)      -- Результат: NULL (!)
REPLACE ('Billy Wilder', 'xyz', 'abc')     -- Результат: 'Billy Wilder'
REPLACE ('Billy Wilder', '', 'abc')        -- Результат: 'Billy Wilder'
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может

повлиять на производительность.

См. также: OVERLAY()

REVERSE()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает обращённую («задом наперёд») строку.

Тип результата: VARCHAR

Синтаксис:

```
REVERSE(str)
```

Примеры:

```
REVERSE ('spoonful')                -- Результат: 'lufnoops'
REVERSE ('Was it a cat I saw?')     -- Результат: '?was I tac a ti saW'
```

Совет

Эта функция очень удобна, если Вы хотите группировать, искать или сортировать окончания строк, например, при работе с доменными именами или адресами электронной почты:

```
CREATE INDEX IX_PEOPLE_EMAIL ON PEOPLE
    COMPUTED BY (REVERSE(EMAIL));


SELECT * FROM PEOPLE
WHERE REVERSE(EMAIL) STARTING WITH REVERSE('.br');
```

RIGHT()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает правую часть входной строки. Количество символов задается вторым параметром.

Тип результата: VARCHAR или BLOB

Синтаксис:

RIGHT(*string*, *length*)

- Функция поддерживает текстовые BLOB любой длины, в том числе и с многобайтными наборами символов. В версиях Firebird 2.1-2.1.3 и 2.5 имелся баг, приводящий к ошибке для BLOB с размером больше 1024 байт при мультибайтном наборе символов. Это поведение было исправлено в боле старших версиях — 2.1.4 и 2.5.1.;
- Если строковый параметр типа BLOB, то и результат будет типа BLOB. В противном случае результат будет типа VARCHAR(n), где n равно длине строкового параметра;
- Если числовой параметр больше длины строкового параметра, то результатом будет строковый параметр;
- Если числовой параметр не является целым числом, то используется банковское округление: 0.5 становится 0, 1.5 становится 2, 2.5 становится 2, 3.5 становится 4 и т.д.

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

Важно

Если в Вашей базе данных декларирована внешняя функция right, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

См. также: LEFT()

ROUND()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Округляет число до ближайшего целого числа. Если дробная часть равна 0.5, то округление до ближайшего большего целого числа для положительных чисел и до ближайшего меньшего для отрицательных чисел. С дополнительным опциональным параметром *scale* число может быть округлено до одной из степеней числа 10 (десятки, сотни, десятые части, сотые части и т.д.) вместо просто целого числа.

Тип результата: INTEGER, масштабированные BIGINT или DOUBLE

Синтаксис:

ROUND (<*number*> [, <*scale*>])

<*number*> :: = численное выражение

<*scale*> :: = целое число, определяющее число десятичных разрядов, к которым должен быть проведено округление, например:
2 для округления к самому близкому кратному 0.01 числу
1 для округления к самому близкому кратному 0.1 числу
0 для округления к самому близкому целому числу
-1 для округления к самому близкому кратному 10 числу
-2 для округления к самому близкому кратному 100 числу

Примечания:

- Если используется параметр *scale*, то результат имеет такой же масштаб, как и первый параметр *number*, например:

  – ROUND(123.654, 1) Результат: 123.700 (а не 123.7)
  – ROUND(8341.7, -3) Результат: 8000.0 (а не 8000)
  – ROUND(45.1212, 0) Результат: 45.0000 (а не 45)

В противном случае (параметр *scale* не задан) результат будет:

– ROUND(45.1212) Результат: 45

Важно

- Если в Вашей базе данных декларирована внешняя функция round, i64round, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .
- Если Вы привыкли к поведению внешней функции round, i64round, то, пожалуйста, обратите внимание - внутренняя функция всегда округляет половины (0.5) вверх для положительных и вниз для отрицательных чисел.

RPAD()

Доступно: DSQL, PSQL

Добавлено: 2.1

Изменено: 2.5 (портировано также в версию 2.1.4)

Описание: Дополняет входную строку справа пробелами или определённой пользователем строкой до заданной длины.

Тип результата: VARCHAR или BLOB

Синтаксис:

RPAD(*string*, *endlength [*, *padstring]*)

- Функция полностью поддерживает тестовые BLOB с любым набором символов и любой длины;
- Если входная строка имеет тип BLOB, то и результат будет иметь тип BLOB. В противном случае тип результата будет VARCHAR(*endlength*);
- Если аргумент *padstring* задан и равен '' (пустой строке), то дополнения строки не происходит;
- Если аргумент *endlength* меньше длины входной строки, то происходит её усечение до длины *endlength*, даже если аргумент *padstring* является пустой строкой.

Важно

Если в Вашей базе данных декларирована внешняя функция rpad, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию

(UDF) — см. раздел EXTERNAL FUNCTION .

Примечание

В версиях Firebird 2.1-2.1.3 все результаты не BLOB типа имели тип VARCHAR(32765), что делало желательным приведение его к более меньшему размеру. Это поведение было исправлено в боле старших версиях.

Примеры:

```
RPAD ('Hello', 12)              -- Результат: 'Hello '
RPAD ('Hello', 12, '-')         -- Результат: 'Hello-------'
RPAD ('Hello', 12, '')          -- Результат: 'Hello'
RPAD ('Hello', 12, 'abc')       -- Результат: 'Helloabcabca'
RPAD ('Hello', 12, 'abcdefghij') -- Результат: 'Helloabcdefg'
RPAD ('Hello', 2)               -- Результат: 'He'
RPAD ('Hello', 2, '-')          -- Результат: 'He'
RPAD ('Hello', 2, '')           -- Результат: 'He'
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

См. также: LPAD()

SIGN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает знак входного параметра: -1, 0 или 1.

Тип результата: SMALLINT

Синтаксис:

```
SIGN (number)
```

Важно

Если в Вашей базе данных декларирована внешняя функция sign, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

SIN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает синус угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

SIN (*angle*)

Любой NOT-NULL результат находится в диапазоне [-1, 1].

Важно

Если в Вашей базе данных декларирована внешняя функция sin, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

SINH()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает гиперболический синус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

SINH (*number*)

Важно

Если в Вашей базе данных декларирована внешняя функция sinh, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

SQRT()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает квадратный корень входного параметра.

Тип результата: DOUBLE PRECISION

Синтаксис:

SQRT (*number*)

Важно

Если в Вашей базе данных декларирована внешняя функция sqrt, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

SUBSTRING()

Доступно: DSQL, PSQL

Добавлено: 1.0

Изменено: 2.0, 2.1, 2.1.5, 2.5.1

Описание: Возвращает подстроку строки *str*, начиная с заданной позиции

startpos до конца строки или указанной длины *length*.

Тип результата: VARCHAR(*n*) или BLOB

Синтаксис:

SUBSTRING (*str* FROM *startpos* [FOR *length*])

Эта функция возвращает подстроку, начиная с позиции, заданной параметром *startpos* (первый символ в возвращаемой строке). Без использования опционального параметра FOR функция возвращает все оставшиеся символы в строке. С использованием параметра FOR - возвращает заданное параметром *length* количество символов или оставшуюся часть строки - в зависимости от того, что короче.

В версии Firebird 1.x параметры и должны быть заданы целыми числами. Начиная с версии Firebird 2.0 они могут быть любым допустимым целочисленным выражением.

Начиная с версии Firebird 2.1 функция полностью поддерживает двоичные и текстовые блобы (BLOB) любой длины и с любым набором символов. Если параметр *str* имеет тип BLOB, то и результат будет иметь тип. Для любых других типов результатом будет тип VARCHAR(*n*). Ранее тип результата был CHAR(*n*), если *str* имел тип CHAR(*n*) или являлся строковым литералом.

Для входного параметра *str*, не являющегося блобом, длина результата функции всегда будет равна длине *str*, независимо от значений параметров *startpos* и *length*. То есть результат SUBSTRING('pinhead' FROM 4 FOR 2) будет иметь тип VARCHAR(7) и содержать строку 'he'.

Если любой из входных параметров имеет значение NULL, то и результат тоже будет иметь значение NULL.

Ошибки

- Если входной параметр *str* имеет тип BLOB и не задан параметр *length*, то результат ограничивается 32767 символами. Обходной путь: для больших BLOB всегда указывайте в качестве третьего аргумента CHAR_LENGTH (*str*) или достаточно большое целое число, если Вы не уверены, что запрашиваемая подстрока поместится в 32767 символа.
  Эта ошибка исправлена в версии Firebird 2.5.1 и портирована в Firebird 2.1.5.
- Ошибка в версии Firebird 2.0, когда параметры *startpos* и/или *length* имели значение NULL и функция возвращала значение “false emptystrings”, также исправлена.

Пример:

```
INSERT INTO ABBRNAMES(ABBRNAME)
    SELECT SUBSTRING(LONGNAME FROM 1 FOR 3)
        FROM LONGNAMES
```

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

TAN()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает тангенс угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

TAN (*angle*)

Важно

Если в Вашей базе данных декларирована внешняя функция tan, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

TANH()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает гиперболический тангенс.

Тип результата: DOUBLE PRECISION

Синтаксис:

TANH (*number*)

Любой NOT-NULL результат находится в диапазоне [-1, 1].

Важно

Если в Вашей базе данных декларирована внешняя функция tanh, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

TRIM()

Доступно: DSQL, PSQL

Добавлено: 2.0

Изменено: 2.1

Описание: Удаляет начальные и/или концевые пробелы (или опционально другие строки) из входной строки. Начиная с версии Firebird 2.1 полностью поддерживаются текстовые блобы (BLOB) любой длины и с любым набором символов.

Тип результата: VARCHAR(*n*) или BLOB

Синтаксис:

TRIM ([<*adjust*>] *str*)

<*adjust*> ::= {[*where*] [*what*]} FROM

where ::= BOTH | LEADING | TRAILING -- по умолчанию BOTH

what ::= подстрока, которую надо удалить (неоднократно, если таких вхождений несколько) из входной строки *str* в её начале и/или конце. По умолчанию является пробелом (' ').

Примеры:

```
SELECT TRIM (' Waste no space ')
        FROM RDB$DATABASE       -- Результат: 'Waste no space'

SELECT TRIM (LEADING FROM ' Waste no space ')
        FROM RDB$DATABASE       -- Результат: 'Waste no space '

SELECT TRIM (LEADING '.' from ' Waste no space ')
        FROM RDB$DATABASE       -- Результат: ' Waste no space '

SELECT TRIM (TRAILING '!' from 'Help!!!!')
        FROM RDB$DATABASE       -- Результат: 'Help'

SELECT TRIM ('la' FROM 'lalala I love you Ella')
        FROM RDB$DATABASE       -- Результат: ' I love you El'

SELECT TRIM ('la' FROM 'Lalala I love you Ella')
        FROM RDB$DATABASE       -- Результат: 'Lalala I love you El'
```

Примечания:

- Если входной параметр *str* имеет тип BLOB, то и результат будет иметь тип BLOB. В противном случае результат будет иметь тип VARCHAR(*n*), где *n* является длиной параметра *str*;
- Подстрока для удаления, если она конечно задана, не должна иметь длину больше, чем 32767 байта. Однако при повторениях подстроки в начале и/или конце входного параметра *str* общее число удаляемых байтов может быть гораздо больше. Ограничение на размер удаляемой подстроки будет снято в Firebird 3.

Важно

При использовании BLOB функции может потребоваться загрузить весь объект в память. Несмотря на то, что сервер действительно пытается ограничить использование памяти, при больших размерах BLOB это может повлиять на производительность.

TRUNC()

Доступно: DSQL, PSQL

Добавлено: 2.1

Описание: Возвращает целую часть числа. С дополнительным опциональным параметром *scale* число может быть округлено до одной из степеней числа 10 (десятки, сотни, десятые части, сотые части и т.д.) вместо просто целого числа.

Тип результата: INTEGER, масштабированные BIGINT или DOUBLE

Синтаксис:

TRUNC (<*number*> [, <*scale*>])

<*number*> :: = численное выражение

<*scale*> :: = целое число, определяющее число десятичных разрядов, к которым должен быть проведено округление, например:
2 для приведения к кратному 0.01 числу
1 для приведения к кратному 0.1 числу
0 для приведения к целому числу
-1 для приведения к кратному 10 числу
-2 для приведения к кратному 100 числу

Примечания:

- Если используется параметр *scale*, то результат имеет такой же масштаб, как и первый параметр *number*, например:

  – TRUNC(789.2225, 2) Результат: 789.2200 (а не 789.22)
  – TRUNC(345.4, -2) Результат: 300.0 (а не 300)
  – TRUNC(-163.41, 0) Результат: -163.00 (а не -163)

  В противном случае (параметр *scale* не задан) результат будет:

  – TRUNC(-163.41) Результат: -163

Важно

- Если в Вашей базе данных декларирована внешняя функция trunc, то она перекрывает внутреннюю функцию. Для того, чтобы была доступна

внутренняя функция, необходимо удалить или изменить внешнюю функцию (UDF) — см. раздел EXTERNAL FUNCTION .

- Если Вы привыкли к поведению внешней функции trunc, то, пожалуйста, обратите внимание - внутренняя функция всегда обрезает дробную часть, т.е. увеличивает отрицательныы числа.

UPPER()

Доступно: DSQL, ESQL, PSQL

Добавлено: IB

Изменено: 2.0, 2.1

Описание: Возвращает входную строку в верхнем регистре. Точный результат зависит от набора символов входной строки. Например, для наборов символов NONE и ASCII только ASCII символы переводятся в верхний регистр; для OCTETS — вся входная строка возвращается без изменений. Начиная с версии Firebird 2.1 полностью поддерживаются текстовые блобы (BLOB) любой длины и с любым набором символов.

Тип результата: (VAR)CHAR или BLOB

Синтаксис:

```
UPPER( str )
```

Примеры:

```
SELECT UPPER(_ISO8859_1 'Débâcle')
    FROM RDB$DATABASE
    -- Результат: 'DÉBÂCLE' (до версии Firebird 2.0: 'DéBâCLE')

SELECT UPPER(_ISO8859_1 'Débâcle' COLLATE FR_FR)
    FROM RDB$DATABASE
    -- Результат: 'DEBACLE', в соответствии с французскими правилами приведения в верхний регистр
```

См. также: LOWER()

UUID_TO_CHAR()

Доступно: DSQL, PSQL

Добавлено: 2.5

Описание: Конвертирует 16-ти байтный UUID в 36 символьную легко читаемую ASCII строку.

Тип результата: CHAR(36)

Синтаксис:

UUID_TO_CHAR (*uuid*)

uuid ::= строка, состоящая из 16 однобайтовых символов

Примеры:

```
SELECT UUID_TO_CHAR(x'876C45F4569B320DBCB4735AC3509E5F')
    FROM RDB$DATABASE
    -- Результат: '876C45F4-569B-320D-BCB4-735AC3509E5F'

SELECT UUID_TO_CHAR(GEN_UUID())
    FROM RDB$DATABASE
    -- Результат: e.g. '680D946B-45FF-DB4E-B103-BB5711529B86'

SELECT UUID_TO_CHAR('Firebird swings!')
    FROM RDB$DATABASE
    -- Результат: '46697265-6269-7264-2073-77696E677321'
```

См. Также: CHAR_TO_UUID() , GEN_UUID()

Глава 15

Внешние функции (UDF)

Внешние функции должны быть "объявлены" (т.е. известны) в базе данных прежде, чем они могут быть использованы. Firebird поставляется с двумя внешними библиотеками функций:

- ib_udf – унаследована от InterBase;
- fbudf – новая библиотека, использующая дескрипторы (Передача параметров BY DESCRIPTOR);
- Поставляется начиная с версии Firebird 1.0 для Windows и 1.5 для Linux.

Вы также можете создать свои собственные библиотеки UDF или использовать UDF сторонних разработчиков.

abs

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ABS()

Описание: Возвращает абсолютное значение входного параметра.

Тип результата: DOUBLE PRECISION

Синтаксис:

abs (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION abs
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_abs' MODULE_NAME 'ib_udf';
```

acos

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ACOS()

Описание: Возвращает арккосинус числа

Тип результата: DOUBLE PRECISION

Синтаксис:

acos (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION acos
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_acos' MODULE_NAME 'ib_udf';
```

addDay

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество дней. Для вычитания дней используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addday (*atimestamp*, *number*)

Объявление:

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

addHour

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество часов. Для вычитания часов используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addhour (*atimestamp, number*)

Объявление:

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

addMilliSecond

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество миллисекунд. Для вычитания миллисекунд используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addmillisecond (*atimestamp*, *number*)

Объявление:

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

addMinute

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество минут. Для вычитания минут используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addMinute (*atimestamp*, *number*)

Объявление:

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

addMonth

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество месяцев. Для вычитания месяцев используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addMonth (*atimestamp, number*)

Объявление:

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

addSecond

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество секунд. Для вычитания секунд используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addSecond (*atimestamp*, *number*)

Объявление:

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

addWeek

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество недель. Для вычитания недель используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addWeek (*atimestamp*, *number*)

Объявление:

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

addYear

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция DATEADD()

Описание: Добавляет к первому входному параметру заданное вторым параметром (*number*) количество лет. Для вычитания лет используйте отрицательное значение параметра *number*.

Тип результата: TIMESTAMP

Синтаксис:

addyear (*atimestamp, number*)

Объявление:

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

ascii_char

Библиотека: ib_udf

Добавлено: IB

Изменено: 1.0, 2.0

Наилучшая альтернатива: Встроенная функция ASCII_CHAR()

Описание: Возвращает ASCII символ, соответствующий числу, переданному в качестве аргумента.

Тип результата: VARCHAR(1)

Синтаксис (без изменений):

ascii_char (*intval*)

Объявление:

```
DECLARE EXTERNAL FUNCTION ascii_char
```

```
INTEGER NULL
RETURNS CSTRING(1) FREE_IT
ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

Декларация отражает тот факт, что UDF как таковая возвращается 1-символьную C строку, а не тип SQL CHAR (1), как указано в заявлении, InterBase. Хотя сервер приводит её к типу VARCHAR (1).

Опциональное указание NULL после INTEGER в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как 0 и её результатом будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- Во всех версиях сервера ascii_char(0) возвращает пустую строку, а не ASCII символ с кодом 0;
- До версии Firebird 2.0 тип результата был CHAR(1).

ascii_val

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ASCII_VAL()

Описание: Возвращает ASCII код, соответствующий символу, переданному в качестве аргумента.

Тип результата: INTEGER

Синтаксис:

ascii_val (*ch*)

Объявление:

```
DECLARE EXTERNAL FUNCTION ascii_val
    CHAR(1)
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'IB_UDF_ascii_val' MODULE_NAME 'ib_udf';
```

Внимание

Поскольку поля типа CHAR дополняются пробелами, то пустой строковый параметр будет замечен на пробел (' '), что приведёт к результату 32. Внутренняя функция ASCII_VAL() в этом случае вернёт 0.

asin

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ASIN()

Описание: Возвращает арксинус числа

Тип результата: DOUBLE PRECISION

Синтаксис:

asin (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION asin
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_asin' MODULE_NAME 'ib_udf';
```

atan

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ATAN()

Описание: Возвращает арктангенс числа

Тип результата: DOUBLE PRECISION

Синтаксис:

atan (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION atan
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_atan' MODULE_NAME 'ib_udf';
```

atan2

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция ATAN2()

Описание: Возвращает угол как *отношение* синуса к косинусу, аргументы у которых задаются этими двумя параметрами, а *знаки* синуса и косинуса соответствуют знакам параметров. Это позволяет получать результаты по всей окружности, включая углы -π/2 и π/2.

Тип результата: DOUBLE PRECISION

Синтаксис:

atan2 (*num1, num2*)

Объявление:

```
DECLARE EXTERNAL FUNCTION atan2
        DOUBLE PRECISION, DOUBLE PRECISION
```

```
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_atan2' MODULE_NAME 'ib_udf';
```

bin_and

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция BIN_AND()

Описание: Возвращает результат побитовой операции AND (И) над аргументами.

Тип результата: INTEGER

Синтаксис:

bin_and (*num1*, *num2*)

Объявление:

```
DECLARE EXTERNAL FUNCTION bin_and
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'IB_UDF_bin_and' MODULE_NAME 'ib_udf';
```

bin_or

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция BIN_OR()

Описание: Возвращает результат побитовой операции OR (ИЛИ) над аргументами.

Тип результата: INTEGER

Синтаксис:

bin_or (*num1*, *num2*)

Объявление:

```
DECLARE EXTERNAL FUNCTION bin_or
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'IB_UDF_bin_or' MODULE_NAME 'ib_udf';
```

bin_xor

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция BIN_XOR()

Описание: Возвращает результат побитовой операции XOR над аргументами.

Тип результата: INTEGER

Синтаксис:

bin_xor (*num1*, *num2*)

Объявление:

```
DECLARE EXTERNAL FUNCTION bin_xor
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'IB_UDF_bin_xor' MODULE_NAME 'ib_udf';
```

ceiling

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенные функции CEIL(), CEILING()

Описание: Возвращает наименьшее целое число, большее или равное аргументу.

Тип результата: DOUBLE PRECISION

Синтаксис:

ceiling (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION ceiling
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_ceiling' MODULE_NAME 'ib_udf';
```

cos

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция COS()

Описание: Возвращает косинус угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

cos (*angle*)

Объявление:

```
DECLARE EXTERNAL FUNCTION cos
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_cos' MODULE_NAME 'ib_udf';
```

cosh

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция COSH()

Описание: Возвращает гиперболический косинус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

cosh (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION cosh
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_cosh' MODULE_NAME 'ib_udf';
```

cot

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция COT()

Описание: Возвращает котангенс угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

cot (*angle*)

Объявление:

```
DECLARE EXTERNAL FUNCTION cot
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_cot' MODULE_NAME 'ib_udf';
```

dow

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Описание: Возвращает название дня недели из входного параметра типа TIMESTAMP. Результат может быть локализован.

Тип результата: VARCHAR(15)

Синтаксис:

dow (*atimestamp*)

Объявление:

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

Примечание переводчика: Получить название дня недели можно и без использования UDF - операторами языка SQL:

```
SELECT
    CASE EXTRACT(WEEKDAY FROM
                        CURRENT_TIMESTAMP - 1) + 1
        WHEN 1 THEN 'Понедельник'
        WHEN 2 THEN 'Вторник'
        WHEN 3 THEN 'Среда'
        WHEN 4 THEN 'Четверг'
        WHEN 5 THEN 'Пятница'
        WHEN 6 THEN 'Суббота'
        WHEN 7 THEN 'Воскресенье'
```

```
        END DAY_WEEK
    FROM RDB$DATABASE
```

Подробное описание функций для работы с датой и временем от Ivan Prenosil см. по ссылке здесь.

См. также: sdow

dpower

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция POWER()

Описание: Возвращает *x* в степени *y*.

Тип результата: DOUBLE PRECISION

Синтаксис:

dpower (*x*, *y*)

Объявление:

```
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

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция FLOOR()

Описание: Возвращает наибольшее целое число, меньшее или равное аргументу.

Тип результата: DOUBLE PRECISION

Синтаксис:

floor (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION floor
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_floor' MODULE_NAME 'ib_udf';
```

getExactTimestamp

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: CURRENT_TIMESTAMP или 'NOW'

Описание: Возвращает системное время с точностью до миллисекунд. Эта функция была добавлена, т.к. в версиях до Firebird 2.0, дробная часть всегда была ".0000", давая точность после запятой 0. Начиная с Firebird 2.0 лучше использовать контекстную переменную CURRENT_TIMESTAMP , которая по умолчанию тоже имеет точность до миллисекунд. Для измерения временных интервалов в PSQL используйте контекстную переменную 'NOW' .

Тип результата: TIMESTAMP

Синтаксис:

getExactTimestamp()

Объявление:

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

i64round

См. round, i64round

i64truncate

См. truncate, i64truncate

ln

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция LN()

Описание: Возвращает натуральный логарифм аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

ln (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION ln
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_ln' MODULE_NAME 'ib_udf';
```

log

Библиотека: ib_udf

Добавлено: IB

Изменено: 1.5

Наилучшая альтернатива: Встроенная функция LOG()

Описание: Начиная с Firebird 1.5 функция возвращает логарифм *y* (второй аргумент) по основанию *x* (первый аргумент). В Firebird 1.0.x и InterBase она ошибочно возвращает логарифм *x* по основанию *y*.

Тип результата: DOUBLE PRECISION

Синтаксис:

log (*x*, *y*)

Объявление:

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

Предупреждение

Если версия используемого сервера Firebird меньше 1.5 и Вы используете UDF log, то Вам надо проверить коды приложений и PSQL на предмет получения корректного значения функции — а не ошибочного (см. выше описание функции).

log10

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция LOG10()

Описание: Возвращает десятичный логарифм аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

log10 (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION log10
    DOUBLE PRECISION, DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_log10' MODULE_NAME 'ib_udf';
```

lower

Библиотека: ib_udf

Добавлено: IB

Изменено: 2.0

Наилучшая альтернатива: Встроенная функция LOWER()

Описание: Возвращает входную строку в нижнем регистре. Обратите, пожалуйста, внимание: только ASCII символы корректно обрабатываются функцией. Если возможно, то вместо внешней используйте внутреннюю функцию LOWER().

Тип результата: VARCHAR(*n*)

Синтаксис:

“LOWER” (*str*)

Объявление:

```
DECLARE EXTERNAL FUNCTION "LOWER"
    CSTRING(255) NULL
    RETURNS CSTRING(255) FREE_IT
    ENTRY_POINT 'IB_UDF_lower' MODULE_NAME 'ib_udf';
```

Вышеприведённое объявление взято из файла ib_udf2.sql. Имя функции взято в двойные кавычки и приведено к верхнему регистру, так как LOWER является зарезервированным словом и не может использоваться в качестве идентификатора,

если только не заключёно в двойные кавычки. При вызове внешней функции *lower* Вы также заключать имя функции в кавычки и использовать точное написание имени в верхнем регистре, иначе приоритет будет иметь внутренняя функция. Большинство других имен внутренних функций не являются зарезервированными словами - в этих случаях преобладает внешняя функция, если она, конечно, объявлена.

Опциональное указание NULL после CSTRING(255) в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;
- До версии Firebird 2.0 тип результата был CHAR(*n*);
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255).

lpad

Библиотека: ib_udf

Добавлено: 1.5

Изменено: 1.5.2, 2.0

Наилучшая альтернатива: Встроенная функция LPAD()

Описание: Дополняет входную строку слева символом *padchar* до заданной позиции *endlength* (включительно).

Тип результата: VARCHAR(*n*)

Синтаксис:

lpad (*str, endlength, padchar*)

Объявление:

```
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';
```

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом будет строка с символами *padchar* и длиной *endlengh* (если *str* является NULL), копией *str* (если *padchar* является NULL, а *endlengh* меньше длины *str*) или усечённой до длины *endlengh* строки *str*.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;
- При вызове этой функции убедитесь, что параметр *endlength* не превышает заявленной при объявлении функции длины результата *n* (размерности первого параметра CSTRING(*n*));
- Если параметр *endlength* меньше длины строки *str*, то *str* усекается до длины *endlength*. При отрицательном значении параметра *endlength* результатом будет NULL;
- Значение NULL параметра *endlength* передаётся как 0;
- Если параметр *padchar* является пустой строкой или имеет значение NULL при объявлении функции без ключевого слова NULL после третьего аргумента, то результатом будет равен входной строке *str* (или усечённой до длины *endlength* строки *str*);
- До версии Firebird 2.0 тип результата был CHAR(*n*);
- Ошибка, приводящая к бесконечному циклу, если параметр *padchar* являлся

пустой строкой или имел значение NULL, была исправлена в версии Firebird 2.0;

- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255).

ltrim

Библиотека: ib_udf

Изменено: 1.5, 1.5.2, 2.0

Наилучшая альтернатива: Встроенная функция TRIM()

Описание: Удаляет из входной строки лидирующие (с начала строки) пробелы. Настоятельно рекомендуем Вам использовать вместо внешней встроенную функцию TRIM(), которая обладает большими возможностями и является универсальной.

Тип результата: VARCHAR(*n*)

Синтаксис (без изменений):

ltrim (*str*)

Объявление:

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

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. Замечания к параметрам типа CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;
- До версии Firebird 2.0 тип результата был CHAR(*n*);
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255);
- В версии Firebird 1.0.x функция возвращает NULL, если входная строка пустая или NULL.

mod

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция MOD()

Описание: Возвращает остаток от целочисленного деления.

Тип результата: DOUBLE PRECISION

Синтаксис:

mod (*a*, *b*)

Объявление:

```
DECLARE EXTERNAL FUNCTION mod
        INTEGER, INTEGER
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_mod' MODULE_NAME 'ib_udf';
```

*nullif

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция NULLIF()

Описание: Четыре функции *nullif — для типов INTEGER, BIGINT, вещественных и строковых, соответственно — возвращает значение первого аргумента, если он не равен второму. При равенстве аргументов возвращается NULL.

Тип результата: Зависит от входных параметров.

Синтаксис:

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

Начиная с версии Firebird 1.5 предпочтительней использование внутренней функции NULLIF().

Предупреждения

- Эти функции возвращают NULL, если второй аргумент имеет значение NULL, даже если первый аргумент не NULL. Это является неправильным результатом. Внутренняя функция NULLIF не имеет этой ошибки;
- Функции i64nullif и dnullif возвращают неправильные и/или странные результаты, если каждый аргумент имеет не предполагаемый тип данных (NUMERIC (18,0) или DOUBLE PRECISION). Если у Вас есть сомнения, то явно приведите тип входных параметров к объявленным в функциях типам (объявления функций приведены ниже).

Объявления:

```
DECLARE EXTERNAL FUNCTION inullif
        INTEGER BY DESCRIPTOR, INTEGER BY DESCRIPTOR
        RETURNS INTEGER 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

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция COALESCE()

Описание: Четыре функции *nvl — для типов INTEGER, BIGINT, вещественных и строковых, соответственно — заменяют значение NULL в первом параметре (если он, конечно, имеет значение NULL) на значение, заданное вторым параметром. То есть они возвращают: - значение первого аргумента, если он не имеет значение NULL; - значение второго аргумента, если первый имеет значение NULL.

Тип результата: Зависит от входных параметров.

Начиная с версии Firebird 1.5 предпочтительней использование внутренней функции COALESCE() .

Предупреждение

- Функции i64nvl и dnvl возвращают неправильные и/или странные результаты, если каждый аргумент имеет не предполагаемый тип данных (NUMERIC (18,0) или DOUBLE PRECISION). Если у Вас есть сомнения, то явно приведите тип входных параметров к объявленным в функциях типам (объявления функций приведены ниже).

Объявления:

```
DECLARE EXTERNAL FUNCTION invl
    INTEGER BY DESCRIPTOR, INTEGER BY DESCRIPTOR
    RETURNS INTEGER 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

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция PI()

Описание: Возвращает приближение значение числа π.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
pi ()
```

Объявление:

```
DECLARE EXTERNAL FUNCTION pi
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_pi' MODULE_NAME 'ib_udf';
```

rand

Библиотека: ib_udf

Добавлено: IB

Изменено: 2.0

Наилучшая альтернатива: Встроенная функция RAND()

Описание: Возвращает псевдослучайное число. До версии Firebird 2.0 эта функция сначала инициализировала генератор случайных чисел значением текущего времени в секундах. Поэтому несколько вызовов функции *rand*() в пределах одной и той же секунды возвращают одно и то же значение. Если Вам нужно старое поведение в версиях Firebird 2 и старше, то используйте UDF srand.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
rand ()
```

Объявление:

```
DECLARE EXTERNAL FUNCTION rand
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_rand' MODULE_NAME 'ib_udf';
```

right

См. sright

round, i64round

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Изменено: 1.5, 2.1.3

Наилучшая альтернатива: Встроенная функция ROUND()

Описание: Эти функции возвращают ближайшее к значению входного параметра целое число (масштабируемое числовое/десятичное). Функции не работают с числами двойной точности или числами с плавающей запятой.

Тип результата: INTEGER / NUMERIC(18,4)

Синтаксис:

round (*number*)
i64round (*bignumber*)

Внимание

Половинки всегда округляется в сторону увеличения входного параметра. Например, 3.5 округлится до 4, а -3.5 до -3. Внутренняя функция ROUND(), доступная начиная с версии Firebird 2.1 округляет половинки до ближайшего большего целого числа для положительных чисел и до ближайшего меньшего для отрицательных чисел

Объявление:

В Firebird 1.0.x, точкой входа для обоих функций является round:

```
DECLARE EXTERNAL FUNCTION Round
        INTEGER BY DESCRIPTOR, INTEGER 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';
```

Начиная с версии Firebird 1.5, точкой входа для обоих функций является fbround:

```
DECLARE EXTERNAL FUNCTION Round
    INTEGER BY DESCRIPTOR, INTEGER 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';
```

Если Ваши базы данных мигрируют с версии Firebird 1.0.x на версию Firebird 1.5 или выше, то Вам надо удалить все старые объявления функций *round и *truncate и объявить их заново, используя обновленные имена точек входа. Начиная с версии Firebird 2.0 можно также выполнить обновление объявления функций с помощью ALTER EXTERNAL FUNCTION .

rpad

Библиотека: ib_udf

Добавлено: 1.5

Изменено: 1.5.2, 2.0

Наилучшая альтернатива: Встроенная функция RPAD()

Описание: Возвращает входную строку, дополненную справа символом *padchar* вплоть до достижения заданной параметром *endlength* длины строки.

Тип результата: VARCHAR(*n*)

Синтаксис:

rpad (*str, endlength, padchar*)

Объявление:

```
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';
```

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом будет строка с символами *padchar* и длиной *endlengh* (если *str* является NULL), копией *str* (если *padchar* является NULL).

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;
- При вызове этой функции убедитесь, что параметр *endlength* не превышает заявленной при объявлении функции длины результата *n* (размерности первого параметра CSTRING(*n*));
- Если параметр *endlength* меньше длины строки *str*, то *str* усекается до длины *endlength*. При отрицательном значении параметра *endlength* результатом будет NULL;
- Значение NULL параметра *endlength* передаётся как 0;
- Если параметр *padchar* является пустой строкой или имеет значение NULL при объявлении функции без ключевого слова NULL после третьего аргумента, то результатом будет равен входной строке *str* (или усечённой до длины *endlength* строки *str*);
- До версии Firebird 2.0 тип результата был CHAR(*n*);
- Ошибка, приводящая к бесконечному циклу, если параметр *padchar* являлся пустой строкой или имел значение NULL, была исправлена в версии Firebird 2.0;
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255).

rtrim

Библиотека: ib_udf

Изменено: 1.5, 1.5.2, 2.0

Наилучшая альтернатива: Встроенная функция TRIM()

Описание: Удаляет из входной строки концевые (до конца строки) пробелы. Настоятельно рекомендуем Вам использовать вместо внешней встроенную функцию TRIM(), которая обладает большими возможностями и является универсальной.

Тип результата: VARCHAR(*n*)

Синтаксис (без изменений):

ltrim (*str*)

Объявление:

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

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;

- До версии Firebird 2.0 тип результата был CHAR(*n*);
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255);
- В версии Firebird 1.0.x функция возвращает NULL, если входная строка пустая или NULL.

sdow

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Описание: Возвращает сокращённое название дня недели из входного параметра типа TIMESTAMP. Результат может быть локализован.

Тип результата: VARCHAR(5)

Синтаксис:

dow (*atimestamp*)

Объявление:

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

Примечание переводчика: Получить день недели можно и операторами языка SQL, т. е. не используя UDF:

```
SELECT
    CASE EXTRACT(WEEKDAY FROM
                            CURRENT_TIMESTAMP - 1) + 1
        WHEN 1 THEN 'Пн'
        WHEN 2 THEN 'Вт'
        WHEN 3 THEN 'Ср'
        WHEN 4 THEN 'Чт'
        WHEN 5 THEN 'Пт'
        WHEN 6 THEN 'Сб'
        WHEN 7 THEN 'Вс'
    END SHORT_DAY_WEEK
```

```
FROM RDB$DATABASE
```

Подробное описание SQL функций для работы с датой и временем от Ivan Prenosil см. по ссылке здесь.

См. также: dow

sign

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция SIGN()

Описание: Возвращает знак входного параметра: -1, 0 или 1.

Тип результата: INTEGER

Синтаксис:

```
sign (number)
```

Объявление:

```
DECLARE EXTERNAL FUNCTION sign
        DOUBLE PRECISION
        RETURNS INTEGER BY VALUE
        ENTRY_POINT 'IB_UDF_sign' MODULE_NAME 'ib_udf';
```

sin

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция SIN()

Описание: Возвращает синус угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

sin (*angle*)

Объявление:

```
DECLARE EXTERNAL FUNCTION sin
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_sin' MODULE_NAME 'ib_udf';
```

sinh

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция SINH()

Описание: Возвращает гиперболический синус аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

sin (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION sinh
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_sinh' MODULE_NAME 'ib_udf';
```

sqrt

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция SQRT()

Описание: Возвращает квадратный корень входного параметра.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
sqrt (number)
```

Объявление:

```
DECLARE EXTERNAL FUNCTION sqrt
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf';
```

srand

Библиотека: ib_udf

Добавлено: 2.0

Описание: Инициализирует генератор случайных чисел значением текущего времени в секундах и затем возвращает первое сгенерированное число. Многократный вызов *srand*() в течение одной и той же секунды возвратит одно и то же значение. Это поведение точно такое же, как и у внешней функции rand до версии Firebird 2.0.

Тип результата: DOUBLE PRECISION

Синтаксис:

```
srand ()
```

Объявление:

```
DECLARE EXTERNAL FUNCTION srand
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_srand' MODULE_NAME 'ib_udf';
```

sright

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Описание: Возвращает *numchars* символов входной строки *str*, начиная с её окончания. Если *numchars* больше длины входной строки *str*, то функция возвратит не изменённую входную строку. Функция корректно работает только с однобайтными наборами символов.

Тип результата: VARCHAR(100)

Синтаксис:

sright (*str, numchars*)

Объявление:

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

Примечание переводчика: Эту задачу можно решить с использованием встроенной функции SUBSTRING():

```
SELECT
    SUBSTRING(str FROM CHAR_LENGTH(str) - numchars + 1)
FROM RDB$DATABASE
```

string2blob

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Наилучшая альтернатива: Встроенная функция CAST()

Описание: Возвращает входную строку в виде BLOB.

Тип результата: BLOB

Синтаксис:

string2blob (*str*)

Объявление:

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

BIT_LENGTH()

strlen

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенные функции BIT_LENGTH() , CHAR_LENGTH(), CHARACTER_LENGTH() , и OCTET_LENGTH()

Описание: Возвращает длину входной строки.

Тип результата: INTEGER

Синтаксис:

strlen (*str*)

Объявление:

```
DECLARE EXTERNAL FUNCTION strlen
    CSTRING(32767)
    RETURNS INTEGER BY VALUE
    ENTRY_POINT 'IB_UDF_strlen' MODULE_NAME 'ib_udf';
```

substr

Библиотека: ib_udf

Добавлено: IB

Изменено: 1.0, 1.5.2, 2.0

Наилучшая альтернатива: Встроенная функция SUBSTRING()

Описание: Возвращает подстроку входной строки *str*, начиная с заданной позиции *startpos* до позиции *endpos* включительно. Как и у всех строковых функций считается, что строка начинается с 1-го символа. Если *endpos* больше длины входной строки *str*, то функция возвращает все символы начиная с позиции *startpos* до конца строки. Функция корректно работает только с однобайтными наборами символов.

Тип результата: VARCHAR(*n*)

Синтаксис:

substr (*str, startpos, endpos*)

Объявление:

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

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;

- До версии Firebird 2.0 тип результата был CHAR(*n*);
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255);
- В версии InterBase функция возвращает NULL, если параметр *endpos* больше длины входной строки *str*.

Совет

Используйте встроенную функцию SUBSTRING() вместо внешней *substr*. Она немного отличается по синтаксису, но зато корректно работает с любыми наборами символов.

substrlen

Библиотека: ib_udf

Добавлено: 1.0

Изменено: 1.5, 2.0

Наилучшая альтернатива: Встроенная функция SUBSTRING()

Описание: Возвращает подстроку входной строки *str*, начиная с заданной позиции *startpos* длиной *length* (или меньшей, если символов от *startpos* до конца строки *str* меньше, чем *length*). Как и у всех строковых функций считается, что строка начинается с 1-го символа. Если *startpos* или *length* менбше 1, то результатом будет пустая строка. Функция корректно работает только с однобайтными наборами символов.

Тип результата: VARCHAR(*n*)

Синтаксис:

substrlen (*str, startpos*, *length*)

Объявление:

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

Вышеприведённое объявление взято из файла ib_udf2.sql. Опциональное указание NULL после CSTRING в объявлении функции появилось начиная с версии Firebird 2.0. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка.

Более подробная информация о передаче параметров со значением NULL приведена в примечании .

Примечания:

- В зависимости от того, как Вы объявили функцию (см. примечание CSTRING), эта функция может принять и возвратить строки с размером до 32767 символов;
- До версии Firebird 2.0 тип результата был CHAR(*n*);
- В версии Firebird 1.5.1 и ниже функция по умолчанию имеет объявление с параметром CSTRING(80), а не CSTRING(255).

Совет

В версии Firebird 1.0 также была добавлена встроенная функция SUBSTRING(), имеющая такой же функционал, как и внешняя функция *substrlen* (тем самым делая её сразу же устаревшей). Кроме того, SUBSTRING() корректно работает с любыми наборами символов.

tan

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция TAN()

Описание: Возвращает тангенс угла. Аргумент должен быть задан в радианах.

Тип результата: DOUBLE PRECISION

Синтаксис:

tan (*angle*)

Объявление:

```
DECLARE EXTERNAL FUNCTION tan
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_tan' MODULE_NAME 'ib_udf';
```

tanh

Библиотека: ib_udf

Добавлено: IB

Наилучшая альтернатива: Встроенная функция TANH()

Описание: Возвращает гиперболический тангенс аргумента.

Тип результата: DOUBLE PRECISION

Синтаксис:

tanh (*number*)

Объявление:

```
DECLARE EXTERNAL FUNCTION tanh
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY VALUE
        ENTRY_POINT 'IB_UDF_tanh' MODULE_NAME 'ib_udf';
```

truncate, i64truncate

Библиотека: fbudf

Добавлено: Firebird 1.0 (Windows), Firebird 1.5 (Linux)

Изменено: 1.5, 2.1.3

Наилучшая альтернатива: TRUNC()

Описание: Эти функции возвращают целую часть входного параметра (масштабируемое числовое/десятичное). Функции не работают с числами двойной точности или вещественными числами с плавающей запятой.

Тип результата: INTEGER / NUMERIC(18)

Синтаксис:

truncate (*number*)
i64truncate *(bignumber)*

Внимание

Половинки всегда округляется в сторону увеличения входного параметра. Например, 3.5 округлится до 4, а -3.5 до -3. Внутренняя функция ROUND(), доступная начиная с версии Firebird 2.1 округляет половинки до ближайшего большего целого числа для положительных чисел и до ближайшего меньшего для отрицательных чисел

Объявления:

В Firebird 1.0.x, точкой входа для обоих функций является truncate:

```
DECLARE EXTERNAL FUNCTION Truncate
        INTEGER BY DESCRIPTOR, INTEGER 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';
```

В Firebird 1.5 точкой входа для обоих функций является fbtruncate:

```
        DECLARE EXTERNAL FUNCTION Truncate
                INTEGER BY DESCRIPTOR, INTEGER 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';
```

Если Ваши базы данных мигрируют с версии Firebird 1.0.x на версию Firebird 1.5 или выше, то Вам надо удалить все старые объявления функций *round и *truncate и объявить их заново, используя обновленные имена точек входа. Начиная с версии Firebird 2.0 можно также выполнить обновление объявления функций с помощью ALTER EXTERNAL FUNCTION .

Приложение А:

Примечания

Набор символов NONE воспринимается «как есть» (“as is”)

Начиная с Firebird 1.5:

В Firebird 1.5.1 улучшили работу с набором символов NONE при присваивании значений из/в полей или переменных с другим набором символов, что приводит к меньшему количеству ошибок транслитерации.

Firebird 1.5.0 при подсоединении к базе данных с кодировкой (набором символов) NONE позволяет Вам прочитать данные из двух несовместимых с NONE наборов символов – таких, как SJIS (Япония) и WIN1251 (Россия) (несмотря на то, что в других случаях. отличных от приведённого выше. Вы не можете читать данные из наборов символов, отличных от набора символов подключения до тех пор, пока подключены с ним к базе данных - это вызовет ошибку транслитерации символов). Данные будут приняты «как есть» (“as is”) и сохранены без выдачи ошибки.

Тем не менее, при подключении с набором символов NONE попытка обновить любой русский или японский столбец данных с использованием параметризованных запросов или текстовых строк без явного указания набора символов и сортировки (introducer syntax) не удастся и вызовет ошибку транслитерации, а все последующие запросы данных из этого подключения (с кодировкой NONE) также перестанут работать.

В Firebird 1.5.1, обе эти проблемы были решены. Данные, полученные от клиента с набором символов NONE до сих пор хранятся "как есть", но при записи сохраняется точная бинарная копия полученной строки. Считывание данных из столбцов с разными наборами символов не приводит к ошибке транслитерации. При подключении к базе данных с набором символов NONE не предпринимается никаких действий для приведения строк к набору символов, в котором она хранится — и, таким образом, ни запись, ни чтение не приводят к ошибкам транслитерации.

Это позволяет работать с данными базы данных в различных наборах при подключении к ней с набором символов NONE. Клиент полностью отвечает за правильное представление и преобразование принятой от сервера строки в надлежащем наборе символов.

Уровни абстракции, управляющие этим, считывают младший байт *sqlsubtype*

поля в структуре XSQLVAR, содержащий идентификатор набора символов.

Строки с набором символов NONE принимаются и неявно сохраняются в наборе символов их контекста, а использование продвинутого (introducer) синтаксиса позволяет приложению обрабатывать строки с различными наборами символов. Это позволяет избежать искажений строк при работе приложения с различными наборами символов.

Примечание

Работа с различными наборами символов с использованием продвинутого (introducer) синтаксис или приведения (CAST) строк всё ещё требуется при работе приложений с кодировкой подключения, отличной от NONE. Оба метода показаны в примере, приведённом ниже, для набора символов ISO8859_1. Обратите внимание на префикс подчёркивания (“_”) в продвинутом (introducer) синтаксисе.

Продвинутый (introducer) синтаксис

_ISO8859_1 *mystring*

Приведение типа

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

Понимание предложения WITH LOCK

Это примечание описывает явную блокировку и её последствия. Опция WITH LOCK, добавленная в Firebird 1.5, обеспечивает возможность ограниченной явной пессимистической блокировки для осторожного использования в затронутых наборах строк:

a) крайне малой выборки (в идеале из одной строки) и
b) при контроле из приложения.

Пессимистическая блокировка редко требуется при работе с Firebird. Эту функцию можно использовать только хорошо понимая её последствия. Требуется хорошее знание различных уровней изоляции транзакции. Предложение WITH LOCK доступно для использования в DSQL и PSQL и только для выборки данных (SELECT) из одной таблицы. Предложение WITH LOCK нельзя использовать:

- в подзапросах;
- в запросах с объединением нескольких таблиц (JOIN);
- с оператором DISTINCT, предложением GROUP BY и при использовании любых агрегатных функций;
- при работе с представлениями;

- при выборке данных из селективных хранимых процедур;
- при работе с внешними таблицами.

Синтаксис и поведение

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

При успешном выполнении предложения WITH LOCK будут заблокированы выбранные строки данных и таким образом запрещён доступ на их изменение в рамках других транзакций до момента завершения Вашей транзакции.

При выборке с использованием предложения FOR UPDATE блокировка применяется к каждой строке, одна за другой, по мере попадания выборки в кэш сервера. Это делает возможным ситуацию, в которой успешная блокировка данных *перестаёт работать* при достижении в выборке строки, заблокированной другой транзакцией.

Сервер, в свою очередь, для каждой записи, подпадающей под явную блокировку, возвращает версию записи, которая является в настоящее время подтверждённой (актуальной), независимо от состояния базы данных, когда был выполнен оператор выборки данных, или исключение при попытке обновления заблокированной записи.

Ожидаемое поведение и сообщения о конфликте зависят от параметров транзакции, определенных в блоке TPB:

Таблица A.1. Влияние параметров TPB на явную блокировку

Режим TPB	Поведение
isc_tpb_consistency	Явные блокировки переопределяются неявными или явными блокировками табличного уровня и игнорируются
isc_tpb_concurrency + isc_tpb_nowait	При подтверждении изменения записи в транзакции, стартовавшей после транзакции, запустившей явную блокировку, немедленно возникает исключение конфликта обновления
isc_tpb_concurrency + isc_tpb_wait	При подтверждении изменения записи в транзакции, стартовавшей после транзакции, запустившей явную блокировку, немедленно

	возникает исключение конфликта обновления. Если в активной транзакции идёт редактирование записи (с использованием явной блокировки или нормальной оптимистической блокировкой записи), то транзакция, делающая попытку явной блокировки, ожидает окончания транзакции блокирования и, после её завершения, снова пытается получить блокировку записи. Это означает что при изменении версии записи и подтверждении транзакции с блокировкой возникает исключение конфликта обновления.
isc_tpb_read_committed + isc_tpb_nowait	Если есть активная транзакция, редактирующая запись (с использованием явной блокировки или нормальной оптимистической блокировкой записи), то сразу же возникает исключение конфликта обновления.
isc_tpb_read_committed + isc_tpb_wait	Если в активной транзакции идёт редактирование записи (с использованием явной блокировки или нормальной оптимистической блокировкой записи), то транзакция, делающая попытку явной блокировки, ожидает окончания транзакции блокирования и, после её завершения, снова пытается получить блокировку записи. Для этого режима TPB никогда не возникает конфликта обновления.

Как сервер работает с WITH LOCK

Попытка редактирования записи с помощью оператора UPDATE, заблокированной другой транзакцией, приводит к вызову исключения конфликта обновления или ожиданию завершения блокирующей транзакции - в зависимости от режима TPB. Поведение сервера здесь такое же, как если бы эта запись уже была изменена блокирующей транзакцией.

Нет никаких специальных кодов gdscode, возвращаемых для конфликтов обновления, связанных с пессимистической блокировкой.

Сервер гарантирует, что все записи, возвращенные явным оператором блокировки, фактически заблокированы и действительно соответствуют условиям поиска, заданным в операторе WHERE, если эти условия не зависят ни от каких других таблиц, не имеется операторов соединения, подзапросов и т.п. Это также гарантирует то, что строки, не попадающие под условия поиска, не будут заблокированы. Это не даёт гарантии, что нет строк, которые попадают под

условия поиска, и не заблокированы.

Примечание

Такая ситуация может возникнуть, если в другой, параллельной транзакции подтверждаются изменения в процессе выполнения текущего оператора блокировки.

Сервер блокирует строки по мере их выборки. Это имеет важные последствия, если Вы блокируете сразу несколько строк. Многие методы доступа к базам данных Firebird по умолчанию используют для выборки данных пакеты из нескольких сотен строк (так называемый "буфер выборки"). Большинство компонентов доступа к данным не выделить строки, содержащиеся в последнем принятом пакете, и для которых произошёл конфликт обновления.

Опциональное предложение “OF <column-names>”

Предложение FOR UPDATE обеспечивает метод, позволяющий предотвратить использование буферизованных выборок, с помощью подпункта “OF <column-names>”, что включает позиционированное обновление.

Совет

Кроме того, некоторые компоненты доступа позволяют установить размер буфера выборки и уменьшить его до 1 записи. Это позволяет Вам заблокировать и редактировать строку до выборки и блокировки следующей или обрабатывать ошибки, не отменяя действий Вашей транзакции.

Предостережения при использовании WITH LOCK

- Откат неявной или явной точки сохранения отменяет блокировку записей, которые изменялись в рамках её действий, но ожидающие окончания блокировки транзакции при этом не уведомляются. Приложения не должны зависеть от такого поведения, поскольку в будущем оно может быть изменено;
- Хотя явные блокировки могут использоваться для предотвращения и/или обработки необычных ошибок конфликтов обновления, объем ошибок обновления (deadlock) вырастет, если Вы тщательно не разработаете свою стратегию блокировки и не будете ей строго управлять;
- Большинство приложений не требуют явной блокировки записей. Основными целями явной блокировки являются: 1) предотвращение дорогостоящей обработки ошибок конфликта обновления в сильно загруженных приложениях и 2) для поддержания целостности объектов,

отображаемых из реляционной базы данных в кластеризируемой среде. Если использование Вами явной блокировки не подпадает под одну из этих двух категорий, то это является неправильным способом решения задач в Firebird;

- Явная блокировка — это расширенная функция; не злоупотребляйте её использованием! В то время как явная блокировка может быть очень важной для веб-сайтов, обрабатывающих тысячи параллельных пишущих транзакций или для систем типа ERP/CRM, работающих в крупных корпорациях, большинство прикладных программ не требуют её использования.

Примеры использования явной блокировки

1. Одиночная:

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

2. Несколько строк с последовательной их обработкой с курсором DSQL:

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

Замечания к параметрам типа CSTRING

Внешние функции, работающие со строками, часто используют тип CSTRINGS(*n*) в своих объявлениях. Этот тип представляет собой завершаемую нулем строку максимальной длины *n*. Большинство функций, обрабатывающих тип CSTRING, позволяют принять и возвратить завершенные нулём строки любой длины. Так зачем же тогда задают длину *n*? Потому что сервер Firebird должен выделить память для обработки ввод/вывода параметров и преобразования их в и из типов данных SQL. Большинство строк, используемых в базах данных имеют размер лишь от нескольких десятков до нескольких сотен байт и было бы пустой тратой памяти каждый раз резервировать для каждой обрабатываемой строки 32 Кб памяти. Поэтому стандартные объявления большинства функций, использующих CSTRING – как Вы можете увидеть в файле ib_udf.sql – имеют длину 255 байтов. (В Firebird 1.5.1 и ниже имеют длину по умолчанию в 80 байт). В качестве примера приведём декларацию внешней функции LPAD:

```
DECLARE EXTERNAL FUNCTION lpad
```

```
        CSTRING(255), INTEGER, CSTRING(1)
        RETURNS CSTRING(255) FREE_IT
        ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf';
```

После объявления параметр CSTRING с определенной длиной Вы не можете вызвать функцию с более длинной входной строкой или получать от неё строку с большей, чем объявленной в объявлении длиной. Но длина строк в стандартных объявлениях - просто разумное значение по умолчанию; они не являются константой и при желании Вы можете изменить их (в описаниях функций, например, в файле ib_udf.sql приводится и максимально допустимая длина строковых параметров — как правило, равная 32767 символам). Если у вас есть данные с длиной строки, например, 512 байт, то Вы совершенно спокойно можете изменить объявление нужной Вам функции, указав новое значение строкового параметра вместо старого.

Особый случай - когда Вы обычно работаете с короткими строки (с размером, например, менее 100 байт), но иногда должны вызывать функцию с большой длиной (VAR)CHAR параметра. Объявление функции с CSTRING (32000) гарантирует, что все её вызовы будут успешными, но это также приводит к необходимости выделять 32000 байта памяти для параметре - даже в том случае, что большинство вызовов будет использовать строки с размером до 100 байт. В такой ситуации Вы можете объявить функцию дважды с различными именами и различными длинами строк:

```
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';
```

Теперь Вы можете вызывать функцию lpad для всех маленьких строк и lpadbig для больших. Обратите внимание, что объявленные в первой строке имена отличаются (они определяют, как Вы будете вызывать функции), но точка входа (имя функции в библиотеке) одинакова для обоих функций.

Передача NULL в UDF в Firebird 2

До версии Firebird 2.0 при передаче во внешнюю функцию параметров SQL со значением NULL они всегда преобразовывались его в эквивалент нуля,

например, число 0 или пустуя строку. Исключение составляют внешние функции, использующие механизм “BY DESCRIPTOR”, введённый в Firebird 1. Библиотека fbudf использует дескрипторы, но подавляющее большинство пользовательских функций, в том числе стандартные библиотеки ib_udf, поставляемые с Firebird, по-прежнему используют старый стиль передачи параметров, унаследованный от InterBase.

По этой причине большинство пользовательских функций не могут различить нулевые значения и NULL.

В версии Firebird 2 был несколько улучшен механизм вызова для внешних функций старого стиля. Сервер теперь передаст входной NULL в виде нулевого (null) указателя в функцию, если функция была объявлена в базе данных с ключевым словом NULL после рассматриваемого параметра, например, так:

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

Такое объявление функции гарантирует, что существующие базы данных и приложения будут продолжать функционировать, как и прежде. При объявлении функции с ключевым словом NULL и при входном параметре со значением NULL сервер будет возвращать значение NULL - это является корректным результатом. Без ключевого слова NULL в объявлении функции, как это было до версии Firebird 2.0, NULL передаётся в функцию как пустая строка и её результатом также будет пустая строка. При объявлении функции без ключевого слова NULL она будет работать также, как и в версии Firebird 1.5 или более ранних версиях.

Обратите внимание на то, что объявление функции с ключевым словом NULL не гарантирует Вам, что эта функция правильно обработает входной параметр со значением NULL. Любая функция должна быть написана или переписана таким образом, чтобы правильно обрабатывать значения NULL. Всегда смотрите и используйте объявления функции, предоставленные её разработчиком. Для функций в библиотеки ib_udf они находятся в файле ib_udf2.sql каталога UDF в директории установки сервера Firebird. Обратите внимание на двойку в имени файла — объявления функций со старым стилем приведены в файле ib_udf.sql.

Приведём список обновлённых функций из библиотеки ib_udf, способных принимать и правильно обрабатывать параметры со значением NULL:

- ascii_char;
- lower;
- lpad и rpad;
- ltrim и rtrim;
- substr и substrlen.

Большинство функций из библиотеки ib_udf остаются такими же, как и были. В любом случае передача значения NULL в UDF старого стиля невозможно, если параметр не ссылочного типа.

Замечание: не используйте в новом коде внешние функции trim и substr* - вместо них используйте встроенные функции LOWER, TRIM и SUBSTRING.

«Обновление» функции ib_udf в существующей базе данных

Если Вы используете в существующей базе данных одну или больше из этих функций и хотите извлечь выгоду из улучшенной обработки значений NULL, то выполните для Вашей базы данных скрипт ib_udf_upgrade.sql из каталога \misc\upgrade\ib_udf директории установки Firebird.

Максимальное количество индексов в различных версиях Firebird

С версии Firebird 1.0 до 2.0 было внесено довольно много изменений, связанных с определением максимального количества индексов на таблицу. Сводные данные о максимальном количестве индексов на таблицу приведены в Таблице А.2.

Таблица А.2. Максимальное количество индексов на таблицу в версиях Firebird 1.0 — 2.0

Размер страницы	Версия Firebird											
	1.0, 1.0.2			1.0.3			1.5.x			2.0.x		
	Количество столбцов в индексе											
	1	2	3	1	2	3	1	2	3	1	2	3
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

Поле RDB$VALID_BLR

В версии Firebird 2.1 в системные таблицы RDB$PROCEDURES и RDB$TRIGGERS было добавлено поле RDB$VALID_BLR. Целью его введения была установка флага о возможной не валидности модуля PSQL (процедуры или триггера) при изменении доменов или столбцов таблиц, от которых он зависит. При возникновении описанной выше ситуации флаг поле RDB$VALID_BLR устанавливается в 0 для процедур и/или триггеров, код которых возможно является не валидным.

Нижеприведённый запрос находит процедуры и триггеры, зависящие от определенного домена (в примере это домен 'MYDOMAIN'), и выводит информацию о состоянии поля 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')
/*
    Замените MYDOMAIN фактическим именем проверяемого
    домена. Используйте заглавные буквы, если домен создавался
    нечувствительным к регистру — в противном случае
    используйте точное написание имени домена с учётом регистра
*/
```

Следующий запрос находит процедуры и триггеры, зависящие от определенного столбца таблицы (в примере это столбец 'MYCOLUMN' таблицы 'MYTABLE'), и выводит информацию о состоянии поля 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 = 'MYTABLE'
                AND RDB$FIELD_NAME = 'MYCOLUMN')
/*
        Замените MYTABLE и MYCOLUMN фактическими именами
        проверяемой таблицы и её столбца. Используйте заглавные
        буквы, если таблица и её столбец создавались
        нечувствительными к регистру — в противном случае
        используйте точное написание имени таблицы и её столбца
        с учётом регистра
*/
```

К сожалению, не все случаи не валидности кода PSQL будут отражены в поле RDB$VALID_BLR. Поэтому после изменения домена или столбца таблицы желательно тщательно проанализировать все процедуры и триггеры, о которых сообщают вышеупомянутые запросы - даже те, которые имеют 1 в столбце "RDB$VALID_BLR".

Обратите внимание на то, что для модулей PSQL, наследованных от более ранних версий Firebird (включая многие системные триггеры, даже если база данных создавалась под версией Firebird 2.1 или выше), поле RDB$VALID_BLR имеет значение NULL. Это *не означает*, что их BLR является недействительным.

Команды утилиты командной строки isql SHOW PROCEDURES и SHOW TRIGGERS при выводе информации отмечают звездочкой модули, у которых поле RDB$VALID_BLR равно 0. Команды SHOW PROCEDURE *PROCNAME* и SHOW TRIGGER *TRIGNAME*, выводящие на экран код PSQL модуля, не сигнализируют пользователя о недопустимом BLR.

Приложение B:Зарезервированные и ключевые слова — полный список

Зарезервированные слова

Полный список зарезервированных слов в версии Firebird 2.5:

```
ADD
ADMIN
```

ALL
ALTER
AND
ANY
AS
AT
AVG
BEGIN
BETWEEN
BIGINT
BIT_LENGTH
BLOB
BOTH
BY
CASE
CAST
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
COLLATE
COLUMN
COMMIT
CONNECT
CONSTRAINT
COUNT
CREATE
CROSS
CURRENT
CURRENT_CONNECTION
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSACTION
CURRENT_USER
CURSOR
DATE
DAY
DEC
DECIMAL

DECLARE
DEFAULT
DELETE
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
ESCAPE
EXECUTE
EXISTS
EXTERNAL
EXTRACT
FETCH
FILTER
FLOAT
FOR
FOREIGN
FROM
FULL
FUNCTION
GDSCODE
GLOBAL
GRANT
GROUP
HAVING
HOUR
IN
INDEX
INNER
INSENSITIVE
INSERT
INT
INTEGER
INTO
IS
JOIN
LEADING
LEFT
LIKE
LONG
LOWER

MAX
MAXIMUM_SEGMENT
MERGE
MIN
MINUTE
MONTH
NATIONAL
NATURAL
NCHAR
NO
NOT
NULL
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OR
ORDER
OUTER
PARAMETER
PLAN
POSITION
POST_EVENT
PRECISION
PRIMARY
PROCEDURE
RDB$DB_KEY
REAL
RECORD_VERSION
RECREATE
RECURSIVE
REFERENCES
RELEASE
RETURNING_VALUES
RETURNS
REVOKE
RIGHT
ROLLBACK
ROW_COUNT
ROWS
SAVEPOINT

SECOND
SELECT
SENSITIVE
SET
SIMILAR
SMALLINT
SOME
SQLCODE
SQLSTATE (добавлено в версии 2.5.1)
START
SUM
TABLE
THEN
TIME
TIMESTAMP
TO
TRAILING
TRIGGER
TRIM
UNION
UNIQUE
UPDATE
UPPER
USER
USING
VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VIEW
WHEN
WHERE
WHILE
WITH
YEAR

Ключевые слова

Следующие термины имеют особое значение в DSQL Firebird 2.5. Некоторые из них также являются и зарезервированными словами.

!<
^<
^=
^>
,
:=
!=
!>
(
)
<
<=
<>
=
>
>=
||
~<
~=
~>
ABS
ACCENT
ACOS
ACTION
ACTIVE
ADD
ADMIN
AFTER
ALL
ALTER
ALWAYS
AND
ANY
AS
ASC
ASCENDING
ASCII_CHAR
ASCII_VAL
ASIN
AT
ATAN
ATAN2
AUTO

AUTONOMOUS
AVG
BACKUP
BEFORE
BEGIN
BETWEEN
BIGINT
BIN_AND
BIN_NOT
BIN_OR
BIN_SHL
BIN_SHR
BIN_XOR
BIT_LENGTH
BLOB
BLOCK
BOTH
BREAK
BY
CALLER
CASCADE
CASE
CAST
CEIL
CEILING
CHAR
CHAR_LENGTH
CHAR_TO_UUID
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMENT
COMMIT
COMMITTED
COMMON
COMPUTED
CONDITIONAL
CONNECT

CONSTRAINT
CONTAINING
COS
COSH
COT
COUNT
CREATE
CROSS
CSTRING
CURRENT
CURRENT_CONNECTION
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSACTION
CURRENT_USER
CURSOR
DATA
DATABASE
DATE
DATEADD
DATEDIFF
DAY
DEC
DECIMAL
DECLARE
DECODE
DEFAULT
DELETE
DELETING
DESC
DESCENDING
DESCRIPTOR
DIFFERENCE
DISCONNECT
DISTINCT
DO
DOMAIN
DOUBLE
DROP
ELSE
END

ENTRY_POINT
ESCAPE
EXCEPTION
EXECUTE
EXISTS
EXIT
EXP
EXTERNAL
EXTRACT
FETCH
FILE
FILTER
FIRST
FIRSTNAME
FLOAT
FLOOR
FOR
FOREIGN
FREE_IT
FROM
FULL
FUNCTION
GDSCODE
GEN_ID
GEN_UUID
GENERATED
GENERATOR
GLOBAL
GRANT
GRANTED
GROUP
HASH
HAVING
HOUR
IF
IGNORE
IIF
IN
INACTIVE
INDEX
INNER
INPUT_TYPE
INSENSITIVE

INSERT
INSERTING
INT
INTEGER
INTO
IS
ISOLATION
JOIN
KEY
LAST
LASTNAME
LEADING
LEAVE
LEFT
LENGTH
LEVEL
LIKE
LIMBO
LIST
LN
LOCK
LOG
LOG10
LONG
LOWER
LPAD
MANUAL
MAPPING
MATCHED
MATCHING
MAX
MAXIMUM_SEGMENT
MAXVALUE
MERGE
MIDDLENAME
MILLISECOND
MIN
MINUTE
MINVALUE
MOD
MODULE_NAME
MONTH
NAMES

NATIONAL
NATURAL
NCHAR
NEXT
NO
NOT
NULL
NULLIF
NULLS
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OS_NAME
OUTER
OUTPUT_TYPE
OVERFLOW
OVERLAY
PAD
PAGE
PAGE_SIZE
PAGES
PARAMETER
PASSWORD
PI
PLACING
PLAN
POSITION
POST_EVENT
POWER
PRECISION
PRESERVE
PRIMARY
PRIVILEGES
PROCEDURE
PROTECTED
RAND
RDB$DB_KEY

READ
REAL
RECORD_VERSION
RECREATE
RECURSIVE
REFERENCES
RELEASE
REPLACE
REQUESTS
RESERV
RESERVING
RESTART
RESTRICT
RETAIN
RETURNING
RETURNING_VALUES
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLLBACK
ROUND
ROW_COUNT
ROWS
RPAD
SAVEPOINT
SCALAR_ARRAY
SCHEMA
SECOND
SEGMENT
SELECT
SENSITIVE
SEQUENCE
SET
SHADOW
SHARED
SIGN
SIMILAR
SIN
SINGULAR
SINH
SIZE

SKIP
SMALLINT
SNAPSHOT
SOME
SORT
SOURCE
SPACE
SQLCODE
SQLSTATE (2.5.1)
SQRT
STABILITY
START
STARTING
STARTS
STATEMENT
STATISTICS
SUB_TYPE
SUBSTRING
SUM
SUSPEND
TABLE
TAN
TANH
TEMPORARY
THEN
TIME
TIMEOUT
TIMESTAMP
TO
TRAILING
TRANSACTION
TRIGGER
TRIM
TRUNC
TWO_PHASE
TYPE
UNCOMMITTED
UNDO
UNION
UNIQUE
UPDATE
UPDATING
UPPER

USER
USING
UUID_TO_CHAR
VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VIEW
WAIT
WEEK
WEEKDAY
WHEN
WHERE
WHILE
WITH
WORK
WRITE
YEAR
YEARDAY

Приложение C: История документа

Точная и полная история изменений документа хранится в нашем дереве CVS по адресу http://firebird.cvs.sourceforge.net/viewvc/firebird/manual/.

История изменений

0.0— PV создал документ как копию Firebird 2.1 Language Reference Update с обновлённым для версии 2.5 содержанием.

Полную историю документа можно посмотреть по здесь.

Август 2013 года — переведен на русский язык.

5 августа 2014 года — добавлено новое из версий Firebrd 2.5.2 (UUID_TO_CHAR) и 2.5.3 (новые контекстные переменные из пространства SYSTEM).

Приложение D: Лицензионные замечания