



Руководство по языку SQL СУБД **Firebird 4.0**

Дмитрий Филиппов, Александр Карпейкин, Алексей Ковязин, Дмитрий
Кузьменко, Денис Симонов, Paul Vinkenoog, Дмитрий Еманов, Mark
Rotteveel

Версия v1.0, 16.05.2021

Содержание

1. О руководстве по языку SQL Firebird 4.0	2
1.1. Что содержит данный документ	2
1.2. Авторство	2
1.2.1. Авторы	2
1.3. Благодарности	3
2. Структура языка SQL	4
2.1. Общие сведения	4
2.1.1. Подмножества SQL	4
2.1.2. Диалекты SQL	5
2.1.3. Действия при ошибках	6
2.2. Основные сведения: операторы, предложения, ключевые слова	6
2.3. Идентификаторы	7
2.3.1. Правила для обычных идентификаторов	7
2.3.2. Правила для идентификаторов с разделителями	8
2.4. Литералы	8
2.5. Операторы и специальные символы	9
2.6. Комментарии	10
3. Типы данных	11
3.1. Целочисленные типы данных	14
3.1.1. SMALLINT	14
3.1.2. INTEGER	14
3.1.3. BIGINT	14
3.1.4. INT128	15
3.1.5. Шестнадцатеричный формат для целых чисел	15
3.2. Типы данных с плавающей точкой	16
3.2.1. Приблизительные числовые типы	16
FLOAT	17
REAL	17
DOUBLE PRECISION	18
LONG FLOAT	18
3.2.2. Десятичные типы с плавающей точкой	18
DECFLOAT	18
3.3. Типы данных с фиксированной точкой	25
3.3.1. NUMERIC	26
3.3.2. DECIMAL	27
3.3.3. Точность арифметических операций	27
3.4. Типы данных для работы с датой и временем	28
3.4.1. Часовой пояс	29

Получение часового пояса сеанса	30
Региональная семантика TIME WITH TIME ZONE	30
Хранение	31
3.4.2. DATE	31
3.4.3. TIME	32
3.4.4. TIMESTAMP	33
3.4.5. Литералы даты и времени	34
3.4.6. Операции, использующие значения даты и времени	38
3.5. Символьные типы данных	39
3.5.1. UNICODE	39
3.5.2. Набор символов клиента	40
3.5.3. Специальные наборы символов	40
3.5.4. Последовательность сортировки	40
Независимый от регистра поиск	41
Последовательности сортировки для UTF-8	41
3.5.5. Индексирование символьных типов	42
3.5.6. CHAR	43
3.5.7. VARCHAR	43
3.5.8. NCHAR	43
3.5.9. Строковые литералы	44
Альтернативы для апострофов в строковых литералах	44
3.6. Логический тип данных	45
3.6.1. BOOLEAN	45
Оператор IS	46
Примеры BOOLEAN	46
Использование Boolean с другими типами данных	47
3.7. Бинарные типы данных	48
3.7.1. BINARY	48
3.7.2. VARBINARY	48
3.7.3. BLOB	49
Подтипы BLOB	49
Особенности BLOB	50
3.7.4. Массивы	51
Указание явных границ для измерений	51
Добавление дополнительных измерений	52
Использование массивов	52
3.8. Специальные типы данных	53
3.8.1. Тип данных SQL_NULL	53
3.9. Преобразование типов данных	54
3.9.1. Явное преобразование типов данных	54
Преобразование к домену	55

Преобразование к типу столбца	55
Допустимые преобразования для функции CAST	55
Преобразование строк в дату и время	56
3.9.2. Неявное преобразование типов данных	60
Неявное преобразование типов при конкатенации	61
3.10. Пользовательские типы данных — домены	61
3.10.1. Атрибуты домена	61
3.10.2. Переопределение свойств доменов	62
3.10.3. Создание доменов	62
3.10.4. Изменение доменов	63
3.10.5. Удаление доменов	64
3.11. Синтаксис объявления типа данных	64
3.11.1. Синтаксис скалярных типов данных	64
Использование доменов в объявлении	65
Использование TYPE OF COLUMN в объявлении	66
3.11.2. Синтаксис типов данных BLOB	66
3.11.3. Синтаксис массивов	67
4. Общие элементы языка	68
4.1. Выражения	68
4.1.1. Литералы (константы)	70
Строковые литералы (константы)	70
Числовые константы	73
Логические литералы	75
4.1.2. Операторы SQL	75
Приоритет операторов	75
Оператор конкатенации	75
Арифметические операторы	76
Операторы сравнения	76
Логические операторы	77
4.1.3. AT	77
4.1.4. NEXT VALUE FOR	78
4.1.5. Условные выражения	79
CASE	79
4.1.6. NULL в выражениях	80
Выражения возвращающие NULL	81
NULL в логических выражениях	81
4.2. Подзапросы	82
4.2.1. Коррелированные подзапросы	82
4.2.2. Подзапросы возвращающие скалярный результат	83
4.3. Предикаты	84
4.3.1. Утверждения	84

4.3.2. Предикаты сравнения	85
4.3.3. Другие предикаты сравнения	85
BETWEEN	86
LIKE	86
STARTING WITH	88
CONTAINING	89
SIMILAR TO	90
IS DISTINCT FROM	96
Логический IS [NOT]	97
IS [NOT] NULL	98
4.3.4. Предикаты существования.....	98
EXISTS	99
IN	100
SINGULAR	102
4.3.5. Количественные предикаты подзапросов	103
ALL	103
ANY и SOME	104
5. Операторы определения данных (DDL).....	106
5.1. DATABASE	106
5.1.1. CREATE DATABASE.....	106
Использование псевдонимов БД	109
Создание БД на удалённом сервере.....	109
Необязательные параметры CREATE DATABASE	110
Диалект базы данных	111
Кто может создать базу данных?	112
Примеры	112
5.1.2. ALTER DATABASE	115
Добавление вторичного файла	116
Изменение пути и имени дельта файла	116
Перевод базы данных в режим “безопасного копирования”	117
Изменение набора символов по умолчанию	118
Изменение привилегий выполнения по умолчанию	118
LINGER	118
Шифрование базы данных	119
Управление репликацией	120
Кто может выполнить ALTER DATABASE?	122
5.1.3. DROP DATABASE	122
Кто может удалить базу данных?	122
Примеры	122
5.2. SHADOW	123
5.2.1. CREATE SHADOW	123

Режимы AUTO и MANUAL	124
Необязательные параметры CREATE SHADOW	124
Кто может создать теневую копию?	124
Примеры	125
5.2.2. DROP SHADOW	125
Кто может удалить теневую копию?	125
Примеры	126
5.3. DOMAIN	126
5.3.1. CREATE DOMAIN	126
Детали для конкретного типа	128
Кто может создать домен?	130
Примеры	130
5.3.2. ALTER DOMAIN	131
Что не может изменить ALTER DOMAIN	134
Кто может изменить домен?	134
Примеры	134
5.3.3. DROP DOMAIN	135
Кто может удалить домен?	136
Примеры	136
5.4. TABLE	136
5.4.1. CREATE TABLE	136
Символьные столбцы	140
Ограничение NOT NULL	140
Значение по умолчанию	140
Столбцы основанные на домене	141
Столбцы идентификации (автоинкремент)	141
Вычисляемые поля	143
Столбцы типа массив	143
Ограничения	143
Привилегии выполнения	148
Управление репликацией	148
Кто может создать таблицу?	148
Примеры	149
Глобальные временные таблицы (GTT)	152
Внешние таблицы	155
5.4.2. ALTER TABLE	157
Счётчик форматов	161
Предложение ADD	162
Предложение DROP	163
Предложение DROP CONSTRAINT	163
Предложение DROP SQL SECURITY	163

Предложение ALTER [COLUMN]	164
Предложение ALTER SQL SECURITY	168
Управление репликацией	168
Кто может изменить таблицу?	169
5.4.3. DROP TABLE	169
Кто может удалить таблицу?	169
5.4.4. RECREATE TABLE	170
Примеры	170
5.5. INDEX	170
5.5.1. CREATE INDEX	171
Уникальные индексы	171
Направление индекса	172
Вычисляемые индексы или индексы по выражению	172
Ограничения на индексы	172
Максимальное количество индексов на таблицу	173
Кто может создать индекс?	173
Примеры	173
5.5.2. ALTER INDEX	174
Использование ALTER INDEX для индексов ограничений	175
Кто может выполнить ALTER INDEX?	175
Примеры	175
5.5.3. DROP INDEX	176
Кто может удалить индекс?	176
Примеры	176
5.5.4. SET STATISTICS	177
Селективность индекса	177
Кто может обновить статистику?	178
Примеры	178
5.6. VIEW	178
5.6.1. CREATE VIEW	178
Обновляемые представления	179
WITH CHECK OPTIONS	180
Привилегии выполнения	181
Кто может создать представление?	181
Примеры	181
5.6.2. ALTER VIEW	185
Кто может изменить представление?	185
Примеры	186
5.6.3. CREATE OR ALTER VIEW	186
Примеры	187
5.6.4. DROP VIEW	187

Кто может удалить представление?	188
Примеры	188
5.6.5. RECREATE VIEW	188
Примеры	189
5.7. TRIGGER	189
5.7.1. CREATE TRIGGER	189
Привилегии выполнения	192
Тело триггера	193
Терминатор оператора	193
DML триггеры (на таблицу или представление)	193
Триггеры на событие базы данных	196
Триггеры на события изменения метаданных	199
5.7.2. ALTER TRIGGER	206
Допустимые изменения	206
Кто может изменить триггеры?	207
Примеры	207
5.7.3. CREATE OR ALTER TRIGGER	209
Примеры	209
5.7.4. DROP TRIGGER	209
Кто может удалить триггеры?	210
Примеры	210
5.7.5. RECREATE TRIGGER	210
Примеры	211
5.8. PROCEDURE	211
5.8.1. CREATE PROCEDURE	211
Терминатор оператора	214
Параметры	214
Привилегии выполнения	215
Тело хранимой процедуры	216
Внешние хранимые процедуры	216
Кто может создать хранимую процедуру?	216
Примеры	216
5.8.2. ALTER PROCEDURE	218
Кто может изменить хранимую процедуру?	219
Примеры	219
5.8.3. CREATE OR ALTER PROCEDURE	219
Примеры	220
5.8.4. DROP PROCEDURE	220
Кто может удалить хранимую процедуру?	221
Примеры	221
5.8.5. RECREATE PROCEDURE	221

Примеры	222
5.9. FUNCTION	222
5.9.1. CREATE FUNCTION.....	223
Терминатор оператора	225
Входные параметры	226
Использование доменов при объявлении параметров	226
Использование типа столбца при объявлении параметров	226
Возвращаемое значение	226
Детерминированные функции	226
Привилегии выполнения	227
Тело хранимой функции	228
Внешние функции	228
Кто может создать функцию?	228
Примеры	229
5.9.2. ALTER FUNCTION	232
Кто может изменить функцию?	232
Примеры	233
5.9.3. CREATE OR ALTER FUNCTION	233
Примеры	233
5.9.4. DROP FUNCTION	234
Кто может удалить функцию?	234
Примеры	234
5.9.5. RECREATE FUNCTION	235
Примеры	235
5.10. PACKAGE	236
5.10.1. CREATE PACKAGE	236
Привилегии выполнения	239
Терминатор оператора	239
Параметры процедур и функций	239
Детерминированные функции	240
Кто может создать пакет?	240
Примеры	241
5.10.2. ALTER PACKAGE	241
Кто может изменить заголовок пакета?	242
5.10.3. CREATE OR ALTER PACKAGE	243
Примеры	244
5.10.4. DROP PACKAGE	244
Кто может удалить заголовок пакета?	244
Примеры	245
5.10.5. RECREATE PACKAGE	245
Примеры	246

5.11. PACKAGE BODY.....	246
5.11.1. CREATE PACKAGE BODY.....	246
Кто может создать тело пакета?.....	250
Примеры	250
5.11.2. DROP PACKAGE BODY.....	250
Кто может удалить тело пакета?	251
Примеры	251
5.11.3. RECREATE PACKAGE BODY	251
Примеры	252
5.12. EXTERNAL FUNCTION	253
5.12.1. DECLARE EXTERNAL FUNCTION	254
Кто может объявить внешнюю функцию?.....	257
Примеры	257
5.12.2. ALTER EXTERNAL FUNCTION.....	258
Кто может изменить внешнюю функцию?	258
Примеры	259
5.12.3. DROP EXTERNAL FUNCTION.....	259
Кто может удалить внешнюю функцию?.....	259
Примеры	260
5.13. FILTER	260
5.13.1. DECLARE FILTER	260
Задание подтипов	261
Параметры DECLARE FILTER.....	261
Кто может создать BLOB фильтр?	262
Примеры	262
5.13.2. DROP FILTER	262
Кто может удалить BLOB фильтр?.....	263
Примеры	263
5.14. SEQUENCE (GENERATOR)	263
5.14.1. CREATE SEQUENCE	264
Кто может создать последовательность?	264
Примеры	265
5.14.2. ALTER SEQUENCE	265
Кто может изменить последовательность?	266
Примеры	267
5.14.3. CREATE OR ALTER SEQUENCE	267
Примеры	268
5.14.4. DROP SEQUENCE	268
Кто может удалить генератор?	269
Примеры	269
5.14.5. RECREATE SEQUENCE	269

Примеры	270
5.14.6. SET GENERATOR.....	270
Кто может изменить значение генератора?	270
Примеры	271
5.15. COLLATION	271
5.15.1. CREATE COLLATION	271
Специфичные атрибуты	273
Кто может создать сортировку?	274
Примеры	274
5.15.2. DROP COLLATION	275
Кто может удалить сортировку?	276
Примеры	276
5.16. CHARACTER SET	276
5.16.1. ALTER CHARACTER SET	276
Примеры	277
5.17. COMMENTS	277
5.17.1. COMMENT ON	277
Кто может добавить комментарий?	279
Примеры	279
Приложение A: License notice	281

Источник часто копируемых справочных материалов: Paul Vinkenoog
Авторские права © 2017-2021 Firebird Project и всех участвующих авторов
на [Public Documentation License Version 1.0](#). Пожалуйста, обратитесь к
[License Notice in the Appendix](#)

Глава 1. О руководстве по языку SQL Firebird 4.0

Это руководство описывает язык SQL, поддерживаемый СУБД Firebird 4.0.

В руководстве также приводятся практические примеры использования SQL, многие из которых взяты из реальной практики.

1.1. Что содержит данный документ

Данный документ содержит описание языка SQL Firebird. Firebird полностью соответствует международным стандартам SQL, от поддержки типов данных, структур хранения данных, механизмов ссылочной целостности до возможностей управления данными и прав доступа. В СУБД Firebird также реализован надежный процедурный язык — процедурный SQL (PSQL) — для хранимых процедур, триггеров и динамически исполняемых блоков кода. Это те области, о которых идет речь в этом руководстве.

В этом документе не рассматриваются вопросы конфигурация, инструменты командной строки и описание API, и другое не относящееся к языку SQL.

1.2. Авторство

1.2.1. Авторы

Прямой контент

- Дмитрий Филиппов (писатель)
- Александр Карпейкин (писатель)
- Алексей Ковязин (писатель, редактор)
- Дмитрий Кузьменко (писатель, редактор)
- Денис Симонов (писатель, редактор)
- Paul Vinkenoog (писатель, дизайнер)
- Mark Rotteveel (писатель)

Ресурсы

- Дмитрий Еманов
- Adriano dos Santos Fernandes
- Александр Пешков
- Владислав Хорсун
- Claudio Valderrama
- Helen Borrie

- и другие

1.3. Благодарности

Спонсоры

Смотри список спонсоров Firebird 2.5 Language Reference.

Спонсоры Руководства по языку SQL на русском языке

Московская биржа

Московская Биржа — крупнейший в России и Восточной Европе биржевой холдинг, образованный 19 декабря 2011 года в результате слияния биржевых групп ММВБ (основана в 1992) и РТС (основана в 1995). Московская Биржа входит в двадцатку ведущих мировых площадок по объёму торгов ценными бумагами, суммарной капитализации торгуемых акций и в десятку крупнейших бирж производных финансовых инструментов.

IBSurgeon (ibase.ru)

Техническая поддержка и инструменты разработчика и администратора для СУБД Firebird.

Есть несколько способов внести свой вклад в документацию Firebird или Firebird в целом:

- Участвуйте в списках рассылки (см. <https://www.firebirdsql.org/en/mailing-lists/>)
- Сообщайте об ошибках или отправляйте запросы на включение на GitHub (<https://github.com/FirebirdSQL/>)
- Станьте разработчиком (для документации свяжитесь с нами по firebird-docs, для Firebird в целом используйте список рассылки Firebird-devel)
- Пожертвуйте в Firebird Foundation (см. <https://www.firebirdsql.org/en/donate/>)
- Станьте платным членом или спонсором Firebird Foundation (см. <https://www.firebirdsql.org/en/firebird-foundation/>)

Глава 2. Структура языка SQL

В этом справочнике описан язык SQL, поддерживаемый Firebird.

2.1. Общие сведения

Для начала ознакомьтесь с некоторыми замечаниями о некоторых характеристиках, лежащих в основе языковой реализации Firebird.

2.1.1. Подмножества SQL

SQL имеет четыре подмножества SQL, используемых в различных областях применения:

- Динамический SQL (DSQL, Dynamic SQL)
- Процедурный SQL (PSQL, Procedural SQL)
- Встроенный SQL (ESQL, Embedded SQL)
- Интерактивный SQL (ISQL, Interactive SQL)

Динамический SQL является основной частью языка, которая соответствует Части 2 (SQL/Foundation – SQL/Основы) спецификации SQL. DSQL представляет собой конструкции, которые передаются клиентскими приложениями с помощью Firebird API и обрабатываются сервером базы данных.

Процедурный SQL является расширением Динамического SQL, в котором дополнительно присутствуют составные операторы, содержащие локальные переменные, присваивание, циклы и другие процедурные конструкции. PSQL относится к Части 4 (SQL/PSM) спецификации SQL. Изначально расширение PSQL было доступно только лишь в постоянно хранимых в базе модулях (процедурах и триггерах), но сравнительно недавно они стали также доступны в Динамическом SQL (смотри EXECUTE BLOCK).

Встроенный SQL определяет подмножество DSQL, поддерживаемое средством Firebird GPRE. GPRE — приложение-препроцессор, которое позволяет вам внедрять SQL конструкции в ваш непосредственный язык программирования (C, C++, Pascal, Cobol и так далее) и производить обработку этих внедрённых конструкций в правильные вызовы Firebird API. Обратите внимание, что ESQL поддерживает только часть конструкций и выражений DSQL.

Интерактивный SQL подразумевает собой язык, который может быть использован для работы с приложением командной строки Firebird ISQL для интерактивного доступа к базам данных. isql является обычным клиентским приложением. Для него обычный язык — это язык DSQL. Однако приложение поддерживает несколько дополнительных команд.

Оба языковых подмножества, как DSQL, так и PSQL полностью представлены в данном руководстве. Из набора инструментария ни ESQL, ни ISQL не описаны здесь отдельно, за исключением тех мест, где это указано явно.

2.1.2. Диалекты SQL

SQL диалект — это термин, определяющий специфические особенности языка SQL, которые доступны во время доступа с его помощью к базе данных. SQL диалект может быть определён как на уровне базы данных, так и на уровне соединения с базой данных. В настоящее время доступны три диалекта:

- Диалект 1 предназначен исключительно для обеспечения обратной совместимости с устаревшими базами данных из очень старых версий InterBase, v.5 и ниже. Базы данных Dialect 1 сохраняют определенные языковые особенности, которые отличаются от Dialect 3, используемого по умолчанию для баз данных Firebird.
 - Информация о дате и времени хранится в типе данных DATE. Имеется тип данных TIMESTAMP, который идентичен DATE.
 - Двойные кавычки могут использоваться как альтернатива апострофам для разделения строковых данных. Это противоречит стандарту SQL - двойные кавычки зарезервированы для особых синтаксических целей как в стандартном SQL, так и в диалекте 3. Поэтому строки с двойными кавычками следует избегать.
 - Точность типов данных NUMERIC и DECIMAL меньше, чем в 3-м диалекте и в случае, если значение точности более 9, Firebird хранит такие значения как длинные значения с плавающей точкой.
 - BIGINT не является доступным типом данных.
 - Идентификаторы нечувствительны к регистру и всегда должны соответствовать правилам для обычных идентификаторов — см. Раздел [Идентификаторы](#) ниже.
 - Хотя значения генератора хранятся как 64-битные целые числа, запрос клиента Dialect 1, например, `SELECT GEN_ID (MyGen, 1)`, вернет значение генератора, усеченное до 32 бит.
- Диалект 2 доступен только в клиентском соединении к Firebird и не может быть применён к базе данных. Он предназначен для того, чтобы помочь в отладке в случае возможных проблем с целостностью данных при проведении миграции с диалекта 1 на 3.
- В базах данных Диалекта 3:
 - Числа с типами данных DECIMAL и NUMERIC хранятся как длинные значения с фиксированной точкой (масштабируемые целые числа) в случае если точность числа больше 9.
 - Тип данных TIME доступен и используется для хранения значения только времени.
 - Тип данных DATE хранит информацию только о дате.
 - Тип данных BIGINT доступен в качестве целого 64-х битного типа данных.
 - Двойные кавычки могут использоваться, но только для идентификаторов, которые являются зависимыми от регистра, а не для строковых данных.
 - Все строки должны быть разделены одинарными кавычками (апострофами).
 - Значения генераторов возвращаются как 64-битное целое.

Для вновь разрабатываемых баз данных и приложений настоятельно рекомендуется использовать 3-й диалект. Диалект при соединении с базой данных должен быть таким же, как и базы данных. Исключением является случай миграции с 1-го в 3-й диалект, когда в строке соединения с базой данных используется 2-й диалект.

По умолчанию это руководство описывает семантику SQL третьего диалекта, если только в тексте явно не указывается диалект.



2.1.3. Действия при ошибках

Обработка любого оператора либо успешно завершается, либо прерывается из-за вызванной определёнными условиями ошибки. Обработку ошибок можно производить, как в клиентском приложении, так и на стороне сервера средствами SQL.

2.2. Основные сведения: операторы, предложения, ключевые слова

Основная конструкция SQL — оператор (statement). Оператор описывает, что должна выполнить система управления базами данных с конкретным объектом данных или метаданных, обычно не указывая, как именно это должно быть выполнено. Достаточно сложные операторы содержат более простые конструкции — предложения (clause) и варианты, альтернативы (options).

Предложения (clause)

Предложение описывает некую законченную конструкцию в операторе. Например, предложение WHERE в операторе SELECT и в ряде других операторов (UPDATE, DELETE) задаёт условия поиска данных в таблице (таблицах), подлежащих выборке, изменению, удалению. Предложение ORDER BY задаёт характеристики упорядочения выходного, результирующего, набора данных.

Альтернативы (options)

Альтернативы, будучи наиболее простыми конструкциями, задаются при помощи конкретных ключевых слов и определяют некоторые дополнительные характеристики элементов предложения (допустимость дублирования данных, варианты использования и др.).

Ключевые слова

В SQL существуют ключевые слова и зарезервированные слова. Ключевые слова — это все слова, входящие в лексику (словарь) языка SQL. Ключевые слова можно (но не рекомендуется) использовать в качестве имён, идентификаторов объектов базы данных, внутренних переменных и параметров. Зарезервированные слова — это те ключевые слова, которые нельзя использовать в качестве имён объектов базы данных, переменных или параметров.

Например, следующий оператор будет выполнен без ошибок потому, что ABS является ключевым, но не зарезервированным словом.

```
CREATE TABLE T (ABS INT NOT NULL);
```

При выполнении такого оператора будет выдана ошибка потому, что ADD является ключевым и зарезервированным словом.

```
CREATE TABLE T (ADD INT NOT NULL);
```

Список зарезервированных и ключевых слов представлен в приложении Зарезервированные и ключевые слова.

2.3. Идентификаторы

Все объекты базы данных имеют имена, которые иногда называют идентификаторами. Максимальная длина идентификатора составляет 63 символа. Существует два типа идентификаторов — имена, похожие по форме на имена переменных в обычных языках программирования, и имена с разделителями (delimited name), которые являются отличительной особенностью языка SQL.

2.3.1. Правила для обычных идентификаторов

- Длина идентификатора не должна превышать 63 символа
- Обычное имя должно начинаться с буквы латинского алфавита (первые 7 бит таблицы ASCII), за которой могут следовать буквы (латинского алфавита), цифры, символ подчёркивания и знак доллара. В имени нельзя использовать буквы кириллицы, пробелы, другие специальные символы. Такое имя нечувствительно к регистру, его можно записывать как строчными, так и прописными буквами. Следующие имена с точки зрения системы являются одинаковыми:

```
fullname  
FULLNAME  
FuLLNaMe  
FullName
```

Синтаксис обычных идентификаторов

```

<name> ::= 
    <letter> | <name><letter> | <name><digit> | <name>_ | <name>$

<letter> ::= <upper letter> | <lower letter>

<upper letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                    N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lower letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                    n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

2.3.2. Правила для идентификаторов с разделителями

- Длина идентификатора не должна превышать 63 символа.
- Имя должно быть заключено в двойные кавычки, например "anIdentifier".
- Идентификатор может содержать любой символ из набора символов UTF-8, включая символы с диакритическими знаками, пробелы и специальные символы.
- Идентификатор может быть зарезервированным словом.
- Идентификаторы с разделителями чувствительны к регистру во всех контекстах.
- Завершающие пробелы в именах с разделителями игнорируются, как и в случае любой строковой константы.
- Идентификаторы с разделителями доступны только в Диалекте 3. Подробнее о диалектах см. [Диалекты SQL](#)

Синтаксис идентификаторов с разделителями

```
<delimited name> ::= "<permitted_character>[<permitted_character> ... ]"
```

 Идентификатор с разделителями, например "FULLNAME", совпадает с обычными идентификаторами FULLNAME, fullname, FullName и т. д. Причина в том, что Firebird хранит обычные идентификаторы в верхнем регистре, независимо от того, как они были определены или объявлены. Идентификаторы с разделителями всегда сохраняются так как их определили или объявили. Таким образом, идентификатор "FullName" (в кавычках) отличается от FullName (без кавычек), который хранится в метаданных как FULLNAME.

2.4. Литералы

Литералы служат для непосредственного представления данных. Ниже приведены примеры стандартных литералов:

- целочисленные — 0, -34, 45, 0X080000000;
- числа с фиксированной точкой — 0.0, -3.14;
- вещественные — 3.23e-23;
- строковые — 'текст', 'don"t!';
- двоичные строки — x'48656C6C6F20776F726C64';
- дата — DATE '10.01.2014';
- время — TIME '15:12:56';
- временная отметка — TIMESTAMP '10.01.2014 13:32:02';
- логические — TRUE, FALSE, UNKNOWN;
- неопределённое состояние — null.

Подробнее о литералах для каждого из типов данных см. [Типы и подтипы данных](#).

2.5. Операторы и специальные символы

Существует набор специальных символов, используемых в качестве разделителей.

```
<special char> ::=  
    <space> | " | % | & | ' | ( | ) | * | + | , | -  
    | . | / | : | ; | < | = | > | ? | [ | ] | ^ | { | }
```

Часть этих символов, а так же их комбинации могут быть использованы как операторы (арифметические, строковые, логические), как разделители команд SQL, для квотирования идентификаторов, и для обозначения границ строковых литералов или комментариев.

Синтаксис операторов

```
<operator> ::=  
    <string concatenation operator>  
    | <arithmetic operator>  
    | <comparison operator>  
    | <logical operator>  
  
<string concatenation operator> ::= "||"  
  
<arithmetic operator> ::= * | / | + | - |  
  
<comparison operator> ::=  
    = | <> | != | ~= | ^= | > | < | >= | <=  
    | !> | ~> | ^> | !< | ~< | ^<  
  
<logical operator> ::= NOT | AND | OR
```

Подробнее об операторах см. Выражения.

2.6. Комментарии

В SQL скриптах, операторах SQL и PSQL модулях могут встречаться комментарии. Комментарий — это произвольный текст заданный пользователем, предназначенный для пояснения работы отдельных частей программы. Синтаксический анализатор игнорирует текст комментариев.

В Firebird поддерживаются два типа комментариев: блочные и однострочные.

Синтаксис

```
<comment> ::= <block comment> | <single-line comment>

<block comment> ::=
/* <character>[<character> …] */

<single-line comment> ::=
-- <character>[<character> …]<end line>
```

Блочные комментарии начинаются с символов `/*` и заканчиваются символом `*/`. Блочные комментарии могут содержать текст произвольной длины и занимать несколько строк.

Однострочные комментарии начинаются с символов `--` и действуют до конца текущей строки.

Пример 1. Комментарии

```
CREATE PROCEDURE P(APARAM INT)
RETURNS (B INT)
AS
BEGIN
/* Данный текст не будет учитываться
при работе процедуры, т.к. является комментарием
*/
B = A + 1; -- Однострочный комментарий
SUSPEND;
END
```

Глава 3. Типы данных

Типы данных используются в случае:

- определения столбца в таблице базы данных в операторе CREATE TABLE или для его изменения с использованием ALTER TABLE;
- при объявлении и редактировании домена оператором CREATE DOMAIN/ALTER DOMAIN;
- при объявлении локальных переменных в хранимых процедурах, функциях, PSQL-блоках и триггерах, при указании аргументов хранимых процедур и функций;
- при описании внешних функций (UDF – функций, определённых пользователем) для указания аргументов и возвращаемых значений;
- при явном преобразовании типов данных в качестве аргумента для функции CAST.

Таблица 1. Обзор типов данных

Наименование	Размер	Точность и ограничения	Описание
BIGINT	64 бита	от -2^{63} до $(2^{63} - 1)$	64 битное целое. Тип данных доступен только в 3 диалекте.
BINARY(n)	n байт	от 1 до 32 767 байт	Бинарный тип данных фиксированной длины. Является псевдонимом типа CHAR(n) CHARACTER SET OCTETS.
BLOB	Переменный	Размер сегмента BLOB ограничивается 64К. Максимальный размер поля BLOB 32 Гб. Для размера страницы 4096 максимальный размер BLOB поля несколько ниже 2 Гб.	Тип данных с динамически изменяемым размером для хранения больших данных, таких как графика, тексты, оцифрованные звуки. Для сегментированных BLOB базовой структурной единицей является сегмент. Подтип BLOB описывает содержимое.
BOOLEAN	8 бит	false, true, unknown	Логический тип данных.

Наименование	Размер	Точность и ограничения	Описание
CHAR(<i>n</i>), CHARACTER(<i>n</i>)	<i>n</i> символов. Размер в байтах зависит от кодировки и количества байт на символ.	от 1 до 32,767 байт	Символьный тип данных фиксированной длины. При извлечении данных, строка дополняется пробелами справа до указанной длины. Если количество символов <i>n</i> не указано, то по умолчанию принимается 1.
DATE	32 бита	от 01.01.0001 до 31.12.9999	ISC_DATE. Только дата без временной части.
DECIMAL (precision, scale)	16, 32, 64 или 128 бит в зависимости от точности	<i>precision</i> = от 1 до 38, указывает, по меньшей мере, количество цифр для хранения; <i>scale</i> = от 0 до 38, задаёт количество знаков после десятичной точки.	Число с десятичной точкой, которое после десятичной точки имеет <i>scale</i> разрядов. <i>scale</i> должно быть меньше или равно <i>precision</i> . Пример: DECIMAL(10,3) содержит число точно в следующем формате: pppppppp.sss.
DECFLOAT(precision)	64 или 128 бит в зависимости от точности	<i>precision</i> = 16 или 34, количество значащих цифр (точность)	SQL:2016 совместимый тип данных точно хранящий десятичные числа с плавающей запятой, основанный на стандарте IEEE 754-2008.
DOUBLE PRECISION	64 бита	от $2.225 * 10^{-308}$ до $1.797 * 10^{308}$	IEEE двойной точности, 15 цифр, размер зависит от платформы.
FLOAT	32 бита	от $1.175 * 10^{-38}$ до $3.402 * 10^{38}$	IEEE одинарной точности, 7 цифр
FLOAT(precision)	32 или 64 бита в зависимости от точности	<i>precision</i> — точность в двоичных числах, может находиться в диапазоне от 1 до 53.	Если <i>precision</i> от 1 до 32 — 32-битное одинарной точности (сионим типа FLOAT). Если <i>precision</i> от 33 до 53 — 64-битное двойной точности (сионим типа DOUBLE PRECISION).
INTEGER, INT	32 бита	от -2147483648 до 2147483647	Знаковое целое
INT128	128 бит	от -2^{127} до $2^{128}-1$	128-битное целое.

Наименование	Размер	Точность и ограничения	Описание
NUMERIC (precision, scale)	16, 32, 64 или 128 бит в зависимости от точности	precision = от 1 до 38, указывает, по меньшей мере, количество цифр для хранения; scale = от 0 до 38, задаёт количество знаков после десятичной точки.	Число с десятичной точкой, которое после десятичной точки имеет scale разрядов. scale должно быть меньше или равно precision. Пример: NUMERIC(10,3) содержит число точно в следующем формате: pppppppp.sss.
REAL	32 бита	от $1.175 * 10^{-38}$ до $3.402 * 10^{38}$	Является синонимом типа FLOAT.
SMALLINT	16 бита	от -32,768 до 32,767	Короткое знаковое целое.
TIME [WITHOUT TIME ZONE]	32 бита	0:00 to 23:59:59.9999	ISC_TIME. Время дня без информации о часовом поясе
TIME WITH TIME ZONE	6 байт	0:00 to 23:59:59.9999	Время дня с информацией о часовом поясе
TIMESTAMP [WITHOUT TIME ZONE]	64 бита (2 X 32 бита)	от 01.01.0001 до 31.12.9999	Дата включающая время без информации о часовом поясе
TIMESTAMP WITH TIME ZONE	10 байт	от 01.01.0001 до 31.12.9999	Дата включающая время с информацией о часовом поясе
VARBINARY(<i>n</i>), BINARY VARYING(<i>n</i>)	<i>n</i> байт.	от 1 до 32,765 байт	Бинарный тип данных переменной длины. Является псевдонимом типа VARCHAR(<i>n</i>) CHARACTER SET OCTETS.
VARCHAR(<i>n</i>), CHAR VARYING(<i>n</i>), CHARACTER VARYING(<i>n</i>)	<i>n</i> символов. Размер в байтах зависит от кодировки и количества байт на символ.	от 1 до 32,765 байт	Размер символов в байтах с учётом их кодировки не может быть больше 32765. Для этого типа данных, в отличие от CHAR (где по умолчанию предполагается количество символов 1), количество символов <i>n</i> обязательно должно быть указано.



Следует иметь в виду, что временной ряд из дат прошлых веков рассматривается без учёта реальных исторических фактов и так, как будто бы во всем этом диапазоне ВСЕГДА действовал только Григорианский календарь.

3.1. Целочисленные типы данных

Для целых чисел используют целочисленные типы данных SMALLINT, INTEGER, BIGINT (в 3-м диалекте) и INT128. Firebird не поддерживает беззнаковый целочисленный тип данных.

3.1.1. SMALLINT

Тип данных SMALLINT представляет собой 16-битное целое. Он применяется в случае, когда не требуется широкий диапазон возможных значений для хранения данных.

Числа типа SMALLINT находятся в диапазоне от -2^{15} до $2^{15} - 1$, или от -32768 до 32767.

Пример 2. Использование SMALLINT

```
CREATE DOMAIN DFLAG AS SMALLINT DEFAULT 0 NOT NULL
  CHECK (VALUE=-1 OR VALUE=0 OR VALUE=1);

CREATE DOMAIN RGB_VALUE AS SMALLINT;
```

3.1.2. INTEGER

Тип данных INTEGER представляет собой 32-битное целое. Сокращённый вариант записи типа данных INT.

Числа типа INTEGER находятся в диапазоне от -2^{31} до $2^{31} - 1$, или от -2,147,483,648 до 2,147,483,647.

Пример 3. Использование INTEGER

```
CREATE TABLE CUSTOMER (
  CUST_NO INTEGER NOT NULL,
  CUSTOMER VARCHAR(25) NOT NULL,
  CONTACT_FIRST VARCHAR(15),
  CONTACT_LAST VARCHAR(20),
  ...
  PRIMARY KEY (CUST_NO)
);
```

3.1.3. BIGINT

BIGINT — это SQL:99-совместимый 64 битный целочисленный тип данных. Он доступен только в 3-м диалекте. При использовании клиентом диалекта 1, передаваемое сервером значение генератора усекается до 32-х битного целого (INTEGER). При подключении в 3-м диалекте значение генератора имеет тип BIGINT.

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

Пример 4. Использование BIGINT

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

3.1.4. INT128

INT128 — не стандартный 128 битный целочисленный тип данных.

Числа типа INT128 находятся в диапазоне от -2^{127} до $2^{127} - 1$.

Пример 5. Использование INT128

```
CREATE PROCEDURE PROC1 (PAR1 INT128)
AS
BEGIN
    -- текст процедуры
END
```

3.1.5. Шестнадцатеричный формат для целых чисел

Начиная с Firebird 2.5, константы трех целочисленных типов можно указать в шестнадцатеричном формате с помощью 9-16 шестнадцатеричных цифр для BIGINT или 1 до 8 цифр для INTEGER. Запись SMALLINT в шестнадцатеричном представлении не поддерживается в явном виде, но Firebird будет прозрачно преобразовывать шестнадцатеричное число в SMALLINT, если это необходимо, при условии что оно попадает в допустимый диапазон положительных и отрицательных значений для SMALLINT.

Использование и диапазоны значений чисел шестнадцатеричной нотации более подробно описаны в ходе обсуждения числовых констант в главе под названием Общие элементы языка.

Пример 6. Использование целых чисел заданных шестнадцатеричном виде

```
INSERT INTO MYBIGINTS VALUES (
    -236453287458723,
    328832607832,
    22,
    -56786237632476,
    0X6F55A09D42, -- 478177959234
    0X7FFFFFFFFFFFFF, -- 9223372036854775807
    0xFFFFFFFFFFFFFF, -- -1
    0X80000000, -- -2147483648, т.е. INTEGER
    0X0800000000, -- 2147483648, т.е. BIGINT
    0xFFFFFFFF, -- -1, т.е. INTEGER
    0X0FFFFFFF -- 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. Значение 0x80000000 (8 цифр) будет сохранено в формате INTEGER, а 0x080000000 (9 цифр) как BIGINT.

3.2. Типы данных с плавающей точкой

Типы данных с плавающей точкой хранятся в двоичном формате IEEE 754, который включает в себя знак, показатель степени и мантиссу. Firebird имеет две формы типов с плавающей точкой:

- приблизительные числовые типы (или двоичные типы с плавающей точкой);
- десятичные типы с плавающей точкой.

3.2.1. Приблизительные числовые типы

Приблизительные числовые типы плавающей запятой, поддерживаемые Firebird представлены типами 32-битной одинарной точностью и 64-битной двойной точности. Эти типы доступны со следующими именами стандартных типов SQL:

- REAL — 32-битный одинарной точности (сионим типа FLOAT);
- FLOAT — 32-битный одинарной точности;
- DOUBLE PRECISION — 64-битный двойной точности;
- FLOAT(*p*), где *p* — точность в двоичных числах
 - 1 <= *p* <= 32 — 32-битное одинарной точности (сионим типа FLOAT)
 - 33 <= *p* <= 53 — 64-битное двойной точности (сионим типа DOUBLE PRECISION)

Кроме того, в Firebird имеются нестандартные имена типов:

- LONG FLOAT — 64-двойной точности (сионим типа DOUBLE PRECISION);
- LONG FLOAT(p), где p — точность в двоичных числах. $1 \leq p \leq 53$ — 64-битное двойной точности (сионим типа DOUBLE PRECISION)

Точность этого типов FLOAT и DOUBLE PRECISION является динамической, что соответствует физическому формату хранения, который составляет 4 байта для типа FLOAT и 8 байт для типа DOUBLE PRECISION.

Учитывая особенности хранения чисел с плавающей точкой, этот тип данных не рекомендуется использовать для хранения денежных данных. По тем же причинам не рекомендуется использовать столбцы с данными такого типа в качестве ключей и применять к ним ограничения уникальности.

При проверке данных столбцов с типами данных с плавающей точкой рекомендуется вместо точного равенства использовать выражения проверки вхождения в диапазон, например BETWEEN.

При использовании таких типов данных в выражениях рекомендуется крайне внимательно и серьёзно подойти к вопросу округления результатов расчётов.

FLOAT

FLOAT — 32-битный тип данных для хранения чисел с плавающей точкой. Он обладает приблизительной точностью 7 цифр после запятой. Числа типа FLOAT находятся в диапазоне от 1.175×10^{-38} до 3.402×10^{38} .

FLOAT с указанием точности

У типа FLOAT может быть указана точность в двоичных числах

Синтаксис

FLOAT(p)

Указанная точность p влияет на способ хранения числа.

- $1 \leq p \leq 32$ — 32-битное одинарной точности (сионим типа FLOAT)
- $33 \leq p \leq 53$ — 64-битное двойной точности (сионим типа DOUBLE PRECISION)



В Firebird 3.0 и более ранних версиях поддерживался синтаксис FLOAT(p), где p — приблизительная точность в десятичных знаках. Если $0 \leq p \leq 7$, то тип отображался на 32-битный одинарной точности. Если $p > 7$, то отображался на 64-битный двойной точности. Это нестандартное поведение. Данный синтаксис не был документирован ранее.

REAL

Тип REAL является синонимом типа FLOAT.

DOUBLE PRECISION

`DOUBLE PRECISION` — 64-битный тип данных для хранения чисел с плавающей точкой. Он обладает приблизительной точностью 15 цифр после запятой. Числа типа `DOUBLE PRECISION` находятся в диапазоне от 2.225×10^{-308} до 1.797×10^{308} .

LONG FLOAT

Синтаксис:

```
LONG FLOAT[(precision)]
<precision> ::= 1..53
```

Тип `LONG FLOAT` является синонимом типа `DOUBLE PRECISION`. У этого типа может быть указана точность в двоичных числах. Указанная точность $1 \leq p \leq 53$ не влияет на способ хранения — число всегда хранится как 64-битное двойной точности.



В Firebird 3.0 и более ранних версиях поддерживался синтаксис `LONG FLOAT(p)`, где p — приблизительная точность в десятичных знаках. Независимо от указанной точности число всегда хранится как 64-битное двойной точности. Данный синтаксис не был документирован ранее.



Эти нестандартные имена типов устарели и могут быть удалены в будущей версии.

3.2.2. Десятичные типы с плавающей точкой

Начиная с Firebird 4.0 поддерживаются типы десятичных чисел с плавающей запятой.

DECFLOAT

`DECFLOAT` является числовым типом из стандарта SQL:2016, который точно хранит числа с плавающей запятой. В отличие от `DECFLOAT` типы `FLOAT` или `DOUBLE PRECISION` обеспечивают двоичное приближение предполагаемой точности. Firebird в соответствии со стандартом IEEE 754-1985 (IEEE 754-2008) реализует типы `DECIMAL64` и `DECIMAL128`, что обеспечивает точность 16 и 34 значащих цифр, и занимает 8 и 16 байт памяти соответственно. Если точность не указана, то по умолчанию используется точность 34 значащих цифры.

Все промежуточные вычисления осуществляются с использованием 34-значными значениями.

Синтаксис

```
DECFLOAT[(precision)]
precision ::= 16 | 34
```

Тип DECFLOAT следует использовать если вам необходимы вычисления и хранение чисел с большой точностью.

Тип DECFLOAT(16) имеет точность 16 значащих цифр, занимает 8-байт, и позволяет хранить числа в диапазоне

от $-9.999999999999999 \times 10^{384}$.. -1.0×10^{-383}
до 1.0×10^{-383} .. $9.999999999999999 \times 10^{384}$

Тип DECFLOAT(34) имеет точность 34 значащих цифр, занимает 16-байт, и позволяет хранить числа в диапазоне

Пример 7. Использование типа DECFLOAT при определении таблицы

```
CREATE TABLE StockPrice (
    id      INT NOT NULL PRIMARY KEY,
    stock  DECFLOAT(16),
    ...
);
```

Пример 8. Использование типа DECFLOAT в PSQL

```
DECLARE VARIABLE v DECFLOAT(34);
```

Математические функции ABS, CEILING, EXP, FLOOR, LN, LOG, LOG10, POWER, SIGN, SQRT поддерживают работу со значениями типа DECFLOAT. Агрегатные функции SUM, AVG, MIN и MAX тоже работают с типом DECFLOAT. Все статистические агрегатные функции (такие как STDDEV или CORR, но не ограничено ими) могут работать с данными типа DECFLOAT.

Firebird поддерживает 4 функции, которые созданы специально для поддержки типа DECFLOAT: **COMPARE DECFLOAT**, **NORMALIZE DECFLOAT**, **QUANTIZE**, **TOTALORDER**.

Режимы округления

CEILING

Округление сверху. Если все отбрасываемые цифры равны нулю или знак числа отрицателен, последняя не отбрасываемая цифра не меняется. В противном случае последняя не отбрасываемая цифра инкрементируется на единицу (округляется в большую сторону).

UP

Округление по направлению от нуля (усечение с приращением). Отбрасываемые значения игнорируются.

HALF_UP

Округление к ближайшему значению. Если результат равноудаленный, выполняется округление в большую сторону. Если отбрасываемые значения больше чем или равны половине (0,5) единицы в следующей левой позиции, последняя не отбрасываемая цифра инкрементируется на единицу (округляется в большую сторону). В противном случае отбрасываемые значения игнорируются.

HALF_EVEN

Округление к ближайшему значению. Если результат равноудаленный, выполняется округление так, чтобы последняя цифра была четной. Если отбрасываемые значения больше половины (0,5) единицы в следующей левой позиции, последняя не отбрасываемая цифра инкрементируется на единицу (округляется в большую сторону). Если они меньше половины, результат не корректируется (то есть отбрасываемые знаки игнорируются). В противном случае, когда отбрасываемые значения точно равны половине, последняя не отбрасываемая цифра не меняется, если она является четной и инкрементируется на единицу (округляется в большую сторону) в противном случае (чтобы получить четную цифру). Этот режим округления называется также банковским округлением и дает ощущение справедливого округления.

HALF_DOWN

Округление к ближайшему значению. Если результат равноудаленный, выполняется округление в меньшую сторону. Если отбрасываемые значения больше чем или равны половине (0,5) единицы в следующей левой позиции, последняя не отбрасываемая цифра декрементируется на единицу (округляется в меньшую сторону). В противном случае отбрасываемые значения игнорируются.

DOWN

Округление по направлению к нулю (усечение). Отбрасываемые значения игнорируются.

FLOOR

Округление снизу. Если все отбрасываемые цифры равны нулю или знак положителен, последняя не отбрасываемая цифра не меняется. В противном случае (знак отрицателен) последняя не отбрасываемая цифра инкрементируется на единицу.

REROUND

Округление к большему значению, если округляется 0 или 5, в противном случае округление происходит к меньшему значению.

Режимы округления	12.341	12.345	12.349	12.355	12.405	-12.345
CEILING	12.35	12.35	12.35	12.36	12.41	-12.34
UP	12.35	12.35	12.35	12.36	12.41	-12.35
HALF_UP	12.34	12.35	12.35	12.36	12.41	-12.35

Режимы округления	12.341	12.345	12.349	12.355	12.405	-12.345
HALF_EVEN	12.34	12.34	12.35	12.36	12.40	-12.34
HALF_DOWN	12.34	12.34	12.35	12.35	12.40	-12.34
DOWN	12.34	12.34	12.34	12.35	12.40	-12.34
FLOOR	12.34	12.34	12.34	12.35	12.40	-12.35
REROUND	12.34	12.34	12.34	12.36	12.41	-12.34

Режим округления можно изменить для текущей сессии используя оператор `SET DECFLOAT ROUND`. По умолчанию используется режим округления `HALF_UP`.

Синтаксис

```
SET DECFLOAT ROUND mode
```

Пример 9. Изменение режима округления

```
SET DECFLOAT ROUND HALF_DOWN;
```



Данный SQL оператор работает вне механизма управления транзакциями, изменения выполненные им вступают в силу немедленно. Его использование разрешено, в том числе и в PSQL коде.

Семантика сравнения

Замыкающие нули в значениях десятичных чисел с плавающей запятой сохраняются. Например, 1.0 и 1.00 — это два различных представления. Это порождает различные семантики сравнения для типа данных `DECFL0AT`, как показано ниже.

Сравнение числовых значений

Замыкающие нули игнорируются в сравнениях. Например, 1.0 равно 1.00. По умолчанию такой тип сравнения используется для индексирования, сортировки, разбивки таблицы, оценки предикатов и других функций — короче говоря, везде, где сравнение выполняется неявно или в предикатах.

Пример 10. Сравнение числовых значений

```

create table stockPrice (stock DECFLOAT(16));

insert into stockPrice
values (4.2);

insert into stockPrice
values (4.2000);

insert into stockPrice
values (4.6125);

insert into stockPrice
values (4.20);

commit;

select * from stockPrice where stock = 4.2;
-- Возвращает три значения 4.2, 4.2000, 4.20

select * from stockPrice where stock > 4.20;
-- Возвращает одно значение 4.6125

select * from stockPrice order by stock;
-- Возвращает все значения, 4.2, 4.2000, 4.20, 4.6125.
-- Первые три значения возвращаются в неопределенном порядке.

```

Сравнение TotalOrder

Замыкающие нули учитываются при сравнении. Например, $1.0 > 1.00$. Каждое значение DECFLOAT имеет порядок в семантике сравнения TotalOrder.

Согласно семантике TotalOrder, порядок различных значений определяется так, как показано в следующем примере:

```
-nan < -snan < -inf < -0.1 < -0.10 < -0 < 0 < 0.10 < 0.1 < inf < snan < nan
```



Обратите внимание на то, что отрицательный нуль меньше положительного нуля при сравнении TotalOrder

Запросить сравнение TotalOrder в предикатах можно при помощи встроенной функции TOTALORDER().

Пример 11. Сравнение TotalOrder

Для курсов акций может быть важным знать точность данных. Например, если курсы обычно указываются с точностью в пять знаков после запятой, а курс равен \$4.2, тогда неясно, равна цена \$4.2000, \$4.2999 или чему-то, лежащему между этими двумя значениями.

```
create table stockPrice (stock DECFLOAT(16));

insert into stockPrice
values (4.2);

insert into stockPrice
values (4.2000);

insert into stockPrice
values (4.6125);

insert into stockPrice
values (4.20);

commit;

select * from stockPrice where TOTALORDER(stock, 4.2000) = 0;
-- Возвращает только значение 4.2000

select * from stockPrice where TOTALORDER(stock, 4.20) = 1;
-- Возвращает два значения 4.2 и 4.6125, которое больше 4.20
```

Порядок, в котором возвращаются арифметически одинаковые значения, имеющие различное количество замыкающих нулей, не определен. Следовательно, ORDER BY по столбцу DECFLOAT со значениями 1.0 и 1.00 возвращает два значения в произвольном порядке. Аналогично, DISTINCT возвращает либо 1.0, либо 1.00.

Обработка ошибок

В процессе вычисления выражений могут возникнуть различные ситуации, которые могут вызвать исключение или проигнорированы. Установить какие ситуации приведут к возбуждению исключения можно с помощью оператора SET DECFLOAT TRAPS TO.

Синтаксис:

```
SET DECFLOAT TRAPS TO <traps list>

<traps list> ::= <trap>[, <trap>[, ... <trap>]]

<trap> ::=
  Division_by_zero
  | Inexact
  | Invalid_operation
  | Overflow
  | Underflow
```

По умолчанию исключения генерируются для следующих ситуаций: `Division_by_zero`, `Invalid_operation`, `Overflow`.

Пример 12. Установка ситуаций для которых будут генерироваться исключения

```
SET DECFLOAT TRAPS TO Division_by_zero, Inexact, Invalid_operation, Overflow,
Underflow;
```



Данный SQL оператор работает вне механизма управления транзакциями, изменения выполненные им вступают в силу немедленно. Его использование разрешено, в том числе и в PSQL коде.

Поддержка в клиентских приложениях

Библиотека `fbclient` версии 4.0 имеет нативную поддержку типа `DECFLOAT`. Однако более старые версии клиентской библиотеки ничего не знают о типе `DECFLOAT`. Для того чтобы старые приложения умели работать с типом `DECFLOAT` вы можете настроить отображение значений `DECFLOAT` на другие доступные типы данных с помощью оператора `SET BIND OF`.

Примеры:

```
SET BIND OF DECFLOAT TO LEGACY;
-- значения столбцов типа DECFLOAT будут преобразованы в тип DOUBLE PRECISION

-- другой вариант
SET BIND OF DECFLOAT TO DOUBLE PRECISION;

SET BIND OF DECFLOAT(16) TO CHAR;
-- значения столбцов типа DECFLOAT(16) будут преобразованы в тип CHAR(23)

SET BIND OF DECFLOAT(34) TO CHAR;
-- значения столбцов типа DECFLOAT(34) будут преобразованы в тип CHAR(42)

SET BIND OF DECFLOAT TO NUMERIC(18, 4);
-- значения столбцов типа DECFLOAT будут преобразованы в тип NUMERIC(18, 4)

SET BIND OF DECFLOAT TO NATIVE;
-- возвращает значения столбцов типа DECFLOAT в нативном типе
```

Различные привязки полезны, если вы планируете использовать значения DECFLOAT со старым клиентом, не поддерживающим собственный формат. Можно выбирать между строками (идеальная точность, но плохая поддержка для дальнейшей обработки), значения с плавающей запятой (идеальная поддержка для дальнейшей обработки, но с плохой точностью) или масштабированные целые числа (хорошая поддержка дальнейшей обработки и требуемая точность, но диапазон значений очень ограничен). Когда используется инструмент, подобный универсальному GUI-клиенту, выбор привязки к CHAR подходит в большинстве случаев.

Литералы констант DECFLOAT

Длина литералов типа DECFLOAT ограничена 1024 символами. Для более длинных значений вам придётся использовать научную нотацию. Например, значение $0.0<1020\text{ zeroes}>11$ не может быть записано как литерал, вместо него вы можете использовать аналогичную научную нотацию: $1.1\text{E-}1022$. Аналогично $10<1022\text{ zeroes}>0$ может быть записано как $1.0\text{E}1024$.

3.3. Типы данных с фиксированной точкой

Данные типы данных позволяют применять их для хранения денежных значений и обеспечивают предсказуемость операций умножения и деления.

Firebird предлагает два типа данных с фиксированной точкой: NUMERIC и DECIMAL. В соответствии со стандартом оба типа ограничивают хранимое число объявленным масштабом (количеством чисел после запятой). При этом подход к тому, как ограничивается точность для типов разный: для столбцов NUMERIC точность является такой, "как объявлено", в то время как DECIMAL столбцы могут получать числа, чья точность, по меньшей мере, равна тому, что было объявлено.

Например, NUMERIC(4, 2) описывает число, состоящее в общей сложности из четырёх цифр,

включая 2 цифры после запятой; итого 2 цифры до запятой, 2 после. При записи в столбец с этим типом данных значений 3.1415 в столбце NUMERIC(4, 2) будет сохранено значение 3,14.

Для данных с фиксированной точкой общим является форма декларации, например NUMERIC(p, s). Здесь важно понять, что в этой записи s — это масштаб, а не интуитивно предсказываемое "количество знаков после запятой". Для "визуализации" механизма хранения данных запомните для себя процедуру:

- При сохранении в базу данных число умножается на 10^{10^s} , превращаясь в целое;
- При чтении данных происходит обратное преобразование числа.

Способ физического хранения данных в СУБД зависит от нескольких факторов: декларируемой точности,ialectа базы данных, типа объявления.

Таблица 2. Способ физического хранения чисел с фиксированной точкой

Точность	Тип данных	Диалект 1	Диалект 3
1 - 4	NUMERIC	SMALLINT	SMALLINT
1 - 4	DECIMAL	INTEGER	INTEGER
5 - 9	NUMERIC и DECIMAL	INTEGER	INTEGER
10 - 18	NUMERIC и DECIMAL	DOUBLE PRECISION	BIGINT
19 - 38	NUMERIC и DECIMAL	INT128	INT128

3.3.1. NUMERIC

Формат объявления данных

```

NUMERIC
| NUMERIC(precision)
| NUMERIC(precision, scale)

```

Таблица 3. Параметры типа NUMERIC

Параметр	Описание
precision	Точность. Может быть в диапазоне от 1 до 38. По умолчанию 9.
scale	Масштаб. Может быть в диапазоне от 0 до <i>precision</i> . По умолчанию 0.

В зависимости от точности *precision* и масштаба *scale* СУБД хранит данные по-разному.

Приведём примеры того, как СУБД хранит данные в зависимости от формы их объявления:

```

NUMERIC(4)      stored as    SMALLINT (exact data)
NUMERIC(4,2)        SMALLINT (data * 102)
NUMERIC(10,4) (Dialect 1) DOUBLE PRECISION
                  (Dialect 3) BIGINT (data * 104)
NUMERIC(20, 2)       INT128 (data * 102)

```



Всегда надо помнить, что формат хранения данных зависит от точности. Например, вы задали тип столбца NUMERIC(2, 2), предполагая, что диапазон значений в нем будет -0.99...0.99. Однако в действительности диапазон значений в столбце будет -327.68..327.67, что объясняется хранением типа данных NUMERIC(2, 2) в формате SMALLINT. Фактически типы данных NUMERIC(4, 2), NUMERIC(3, 2) и NUMERIC(2, 2) являются одинаковыми.

Таким образом, для реального хранения данных в столбце с типом данных NUMERIC(2, 2) в диапазоне -0.99...0.99 для него надо создавать ограничение.

3.3.2. DECIMAL

Формат объявления данных

```
DECIMAL
| DECIMAL(precision)
| DECIMAL(precision, scale)
```

Таблица 4. Параметры типа DECIMAL

Параметр	Описание
precision	Точность. Может быть в диапазоне от 1 до 38. По умолчанию 9.
scale	Масштаб. Может быть в диапазоне от 0 до <i>precision</i> . По умолчанию 0.

Формат хранения данных в базе во многом аналогичен NUMERIC, хотя существуют некоторые особенности, которые проще всего пояснить на примере.

Приведём примеры того, как СУБД хранит данные в зависимости от формы их объявления:

```
NUMERIC(4)      stored as    INTEGER (exact data)
NUMERIC(4,2)     INTEGER (data * 102)
NUMERIC(10,4)   (Dialect 1) DOUBLE PRECISION
                (Dialect 3) BIGINT (data * 104)
NUMERIC(20, 2)   INT128 (data * 102)
```

3.3.3. Точность арифметических операций

Функции MIN, MAX, SUM, AVG работают со всеми точными числовыми типами. SUM и AVG являются точными, если обрабатываемая запись имеет точный числовой тип, а масштабированная сумма соответствует 64 или 128 битам: в противном случае возникает исключение переполнения. SUM и AVG никогда не вычисляются с использованием арифметики с плавающей запятой, если тип данных столбца не является приблизительным числом.

Функции MIN и MAX для точного числового столбца возвращают точный числовой результат, имеющий ту же точность и масштаб, что и столбец. SUM и AVG для точного числового типа возвращают результат типа NUMERIC ({18 | 38}, S) или DECIMAL ({18 | 38}, S),

где S - масштаб столбца. Стандарт SQL определяет масштаб результата в таких случаях, в то время как точность SUM или AVG для столбцов с фиксированной точкой определяется реализацией: мы определяем его как 18 или 38 (если точность аргумента 18 или 38).

Если два операнда OP1 и OP2 являются точными числами с масштабами S1 и S2 соответственно, то OP1 + OP2 и OP1 - OP2 являются точными числами с точностью 18 или 38 (если один из аргументов с точностью 38) и масштабом равному наибольшему из значений S1 и S2, тогда как для OP1 * OP2 и OP1 / OP2 являются точными числами с точностью 18 или 38 (если точность аргументов 18 или 38) и шкалой S1 + S2. Масштабы этих операций, кроме деления, определяются стандартом SQL. Точность всех этих операций и масштаб при делении стандартом не регламентируются, а определяются реализацией: Firebird определяет точность как 18 или 38 (если точность аргументов 18 или 38), а масштаб деления как S1 + S2, такой же, что определён стандартом в для умножения.

Всякий раз, когда выполняется арифметические операции с точными числовыми типами, в случае потери точности будет сообщено об ошибке переполнения, а не возвращено неправильное значение. Например, если столбец DECIMAL (18,4) содержит наиболее отрицательное значение этого типа, -922337203685477.5808, попытка разделить этот столбец на -1 будет сообщать об ошибке переполнения, поскольку истинный результат превышает наибольшее положительное значение, которое может быть представлено в типе, а именно 922337203685477.5807.

Если один operand является точным числом, а другой приблизительным числом, то результатом любого из четырех диадических операторов будет типа DOUBLE PRECISION. (В стандарте говорится, что результат является приблизительным числом с точностью, по крайней мере, такой же как точность приблизительного числового операнда: Firebird удовлетворяет этому требованию, всегда используя DOUBLE PRECISION, поскольку этот тип является максимальным приблизительным числовым типом, который предоставлен в Firebird.)

3.4. Типы данных для работы с датой и временем

В СУБД Firebird для работы с данными, содержащими дату и время, используются типы данных DATE, TIME и TIMESTAMP. В 3-м диалекте присутствуют все три вышеназванных типа данных, а в 1-м для операций с датой и временем доступен только тип данных DATE, который не тождественен типу данных DATE 3-го диалекта, а является типом данных TIMESTAMP из 3-го диалекта.



В диалекте 1 тип DATE может быть объявлен как TIMESTAMP. Такое объявление является рекомендуемым для новых баз данных в 1-м диалекте.

В типах TIMESTAMP и TIME Firebird хранит секунды с точностью до десятитысячных долей. Если вам необходима более низкая гранулярность, то точность может быть указана явно в виде тысячных, сотых или десятых долей секунды в базах данных в 3 диалекте и ODS 11 и выше.

Несколько полезных сведений о точности секунд

Временная часть типов TIME или TIMESTAMP представляет собой 4-байтный целое (WORD) вмещающее значение времени с долями секунды, и хранящаяся как количество десятитысячных долей секунды прошедших с полуночи. Фактическая точность значений полученных из time(stamp) функций и переменных будет следующей:

- CURRENT_TIME — по умолчанию имеет точность до секунды, точность до миллисекунд может быть указана следующим образом CURRENT_TIME (0 | 1 | 2 | 3)
- CURRENT_TIMESTAMP — по умолчанию имеет точность до миллисекунды, точность от секунд до миллисекунд может быть указана следующим образом CURRENT_TIMESTAMP (0 | 1 | 2 | 3)
- LOCALTIME — по умолчанию имеет точность до секунды, точность до миллисекунд может быть указана следующим образом LOCALTIME (0 | 1 | 2 | 3)
- LOCALTIMESTAMP — по умолчанию имеет точность до миллисекунды, точность от секунд до миллисекунд может быть указана следующим образом LOCALTIMESTAMP (0 | 1 | 2 | 3)
- Литерал 'NOW' имеет точность до миллисекунд;
- Функции DATEADD и DATEDIFF поддерживают точность до десятых долей миллисекунд.
- Функция EXTRACT возвращает значения с точностью до десятых долей миллисекунды для аргументов SECOND и MILLISECOND;



3.4.1. Часовой пояс

Типы TIMESTAMP и TIME могут содержать или не содержать сведения о часовом поясе в зависимости от способа объявления. По умолчанию типы TIMESTAMP и TIME не содержат часовой пояса.

При преобразовании из/в TIME WITH TIME ZONE или TIMESTAMP WITH TIME ZONE учитывайте что типы TIME WITHOUT TIME ZONE, TIMESTAMP WITHOUT TIME ZONE и DATE определены для использования в часовом поясе сеанса.

Часовой пояс сеанса как следует из названия может быть разным для каждого соединения с базой данных. Он может быть установлен с помощью DPB `isc_dpb_session_time_zone`, а если нет, то он будет считан из параметра `DefaultTimeZone` конфигурации `firebird.conf`. Если параметр `DefaultTimeZone` не установлен, то часовой пояс сеанса будет тем же, что используется операционной системой в которой запущен процесс Firebird.

Часовой пояс сеанса может быть изменён с помощью оператора `SET TIME ZONE` или сброшен в исходное значение с помощью `SET TIME ZONE LOCAL`.

Часовой пояс может быть задан строкой с регионом часового пояса (например, America/Sao_Paulo), или в виде смещения часов:минут относительно GMT (например, -03:00).

Список региональных часовых поясов и их идентификаторов можно посмотреть в таблице `RDB$TIME_ZONES`. Правила преобразования региональных часовых поясов в смещение в минутах можно получить с помощью процедуры `RDB$TIME_ZONE_UTIL.TRANSITIONS`.

`{TIME | TIMESTAMP} WITH TIMEZONE` считается равным другому `{TIME | TIMESTAMP} WITH TIMEZONE`, если их преобразование в UTC равно, например `time '10:00 -02' = time '09:00 -03'`, поскольку оба времени эквивалентны `time '12:00 GMT'`. Это также справедливо в контексте ограничения `UNIQUE` и для сортировки.



`EXTENDED {TIME | TIMESTAMP} WITH TIMEZONE` предназначены для использования только при общении с клиентами, они решают проблему представления правильного времени на клиентах, у которых отсутствует библиотека ICU. Нельзя использовать расширенные типы данных в таблицах, процедурах и т.д. Единственный способ использовать эти типы данных — это приведение типов данных, включая инструкцию `SET BIND` (дополнительную информацию смотри в [SET BIND OF](#)).

Получение часового пояса сеанса

Получить текущий часовой пояс сеанса можно с использованием функции `RDB$GET_CONTEXT` с аргументами '`SYSTEM`' для пространства имён и '`SESSION_TIMEZONE`' в качестве имени переменной.

Пример 13. Получение часового пояса сеанса

```
set time zone '-02:00';
select rdb$get_context('SYSTEM', 'SESSION_TIMEZONE') from rdb$database;
-- returns -02:00

set time zone 'America/Sao_Paulo';
select rdb$get_context('SYSTEM', 'SESSION_TIMEZONE') from rdb$database;
-- returns America/Sao_Paulo
```

Региональная семантика TIME WITH TIME ZONE

По определению региональные часовые пояса зависят от момента (дата и время — или `timestamp`), чтобы узнать его смещение UTC относительно GMT. Но Firebird также поддерживает региональные часовые пояса в значениях `TIME WITH TIMEZONE`.

При построении значения `TIME WITH TIMEZONE` из литерала или его преобразования, значение UTC должно быть вычислено и не может быть изменено, поэтому текущая дата может не использоваться. В этом случае используется фиксированная дата `2020-01-01`. Таким образом, при сравнении `TIME WITH TIMEZONE` с различными часовыми поясами сравнение выполняется аналогично тому, как они представляют собой значения `TIMESTAMP WITH TIMEZONE` на заданную дату.

Однако при преобразовании между типами `TIMESTAMP` в `TIME WITH TIMEZONE` эта

фиксированная дата не используется, в противном случае могут наблюдаться некоторые странные преобразования, когда текущая дата имеет другое смещение (из-за изменений летнего времени), чем в 2020-01-01. В этом случае при преобразовании TIME WITH TIME ZONE в TIMESTAMP WITH TIME ZONE сохраняется часть времени (если это возможно). Например, если текущая дата 2020-05-03, эффективное смещение в часовом поясе America/Los_Angeles равно -420, а его эффективное смещение в 2020-01-01 равно -480, но cast(time '10:00:00 America/Los_Angeles' as timestamp with time zone) даст в результате 2020-05-03 10:00:00.0000 America/Los_Angeles вместо корректировки временной части.

Но в дату, когда начинается летнее время, пропущен час, например, для часовогого пояса America/Los_Angeles в 2021-03-14 нет времени с 02:00:00 до 02:59:59. В этом случае преобразование выполняется как построение литерала, и час корректируется до следующего допустимого значения. Например, в 2021-03-14 cast(time '02:10:00 America/Los_Angeles' as timestamp with time zone) даст результат 2021-03-14 03:10:00.0000 America/Los_Angeles.

Хранение

{TIME | TIMESTAMP} WITH TIME ZONE хранится так же как {TIME | TIMESTAMP} WITHOUT TIME ZONE плюс 2 байта для идентификации часовогого пояса или смещения.

TIME/TIMESTAMP часть хранится в UTC (и переводится в сохранённый часовий пояс).

EXTENDED {TIME | TIMESTAMP} WITH TIME ZONE хранится так же как {TIME | TIMESTAMP} WITH TIME ZONE плюс 2 байта, которые содержат абсолютное смещение относительно GMT в минутах.

3.4.2. DATE

В 3-м диалекте тип данных DATE, как это и следует предположить из названия, хранит только одну дату без времени. В 1-м диалекте тип DATE эквивалентен типу TIMESTAMP и хранит дату вместе со временем.

Допустимый диапазон хранения от 01.01.0001 н.э. до 31.12.9999 н.э.



В случае необходимости сохранять в 1 диалекте только значения даты, без времени, при записи в таблицу добавляйте время к значению даты в виде литерала '00:00:00.0000'.

Пример 14. Пример использования DATE

```
CREATE TABLE DataLog(
    id BIGINT NOT NULL,
    bydate DATE
);
```

```
...
AS
DECLARE BYDATE DATE;
BEGIN
...
```

См. также [EXTRACT](#), [CURRENT_DATE](#), [Литералы дат](#).

3.4.3. TIME

Синтаксис

TIME [{WITH | WITHOUT} TIME ZONE]

EXTENDED TIME WITH TIME ZONE

Этот тип данных доступен только в 3-мialectе. Позволяет хранить время дня в диапазоне от 00:00:00.0000 до 23:59:59.9999. По умолчанию тип TIME не содержит информацию о часовом поясе. Для того чтобы тип TIME включал информацию о часовом поясе необходимо использовать его с модификатором WITH TIME ZONE.

EXTENDED TIME WITH TIME ZONE предназначен для использования только при общении с клиентами, он решает проблему представления правильного времени на клиентах, у которых отсутствует библиотека ICU. Нельзя использовать расширенные типы данных в таблицах, процедурах и т.д. Единственный способ использовать эти типы данных — это приведение типов данных, включая инструкцию SET BIND (дополнительную информацию смотри в [SET BIND OF](#)).



Пример 15. Пример использования TIME

```
CREATE TABLE DataLog(
    id BIGINT NOT NULL,
    bytime TIME WITH TIME ZONE
);
```

```
...
AS
DECLARE BYTIME TIME; -- без часового пояса
DECLARE BYTIME2 TIME WITHOUT TIME ZONE; -- без часового пояса
DECLARE BYTIME3 TIME WITH TIME ZONE; -- с информацией о часовом поясе
BEGIN
...

```

См. также [EXTRACT](#), [AT](#), [LOCALTIME](#), [CURRENT_TIME](#), [Литералы времени](#).

3.4.4. TIMESTAMP

Синтаксис

```
TIMESTAMP [{WITH | WITHOUT} TIME ZONE]
```

```
EXTENDED TIMESTAMP WITH TIME ZONE
```

Этот тип данных хранит временную метку (дату вместе со временем) в диапазоне от 01.01.0001 00:00:00.0000 до 31.12.9999 23:59:59.9999. По умолчанию тип `TIMESTAMP` не содержит информацию о часовом поясе. Для того чтобы тип `TIMESTAMP` включал информацию о часовом поясе необходимо использовать его с модификатором `WITH TIME ZONE`.

 `EXTENDED TIMESTAMP WITH TIME ZONE` предназначен для использования только при общении с клиентами, он решает проблему представления правильного времени на клиентах, у которых отсутствует библиотека ICU. Нельзя использовать расширенные типы данных в таблицах, процедурах и т.д. Единственный способ использовать эти типы данных — это приведение типов данных, включая инструкцию `SET BIND` (дополнительную информацию смотри в [SET BIND OF](#)).

Пример 16. Пример использования TIME

```
CREATE TABLE DataLog(
    id BIGINT NOT NULL,
    bydate TIMESTAMP WITH TIME ZONE
);

...
AS
DECLARE BYDATE TIMESTAMP; -- без часового пояса
DECLARE BYDATE2 TIMESTAMP WITHOUT TIME ZONE; -- без часового пояса
DECLARE BYDATE3 TIMESTAMP WITH TIME ZONE; -- с информацией о часовом поясе
BEGIN
...

```

См. также [EXTRACT](#), [AT](#), [LOCALTIMESTAMP](#), [CURRENT_TIMESTAMP](#), [Литералы временных меток](#).

3.4.5. Литералы даты и времени

Для записи литералов даты и времени в Firebird используются сокращенные "C-style" выражения. Строковое представление даты и времени должно быть в одном из разрешённых форматов.

Синтаксис

```

<date_literal> ::= DATE <date>

<time_literal> ::= TIME <time>

<timestamp_literal> ::= TIMESTAMP <timestamp>

<date> ::=
[YYYY<p>]MM<p>DD |  

MM<p>DD[<p>YYYY] |  

DD<p>MM[<p>YYYY] |  

MM<p>DD[<p>YY] |  

DD<p>MM[<p>YY]

<time> ::= HH[:mm[:SS[.NNNN]]] [<time zone>]

<timestamp> ::= <date> <time>

<time zone> ::=
<time zone region> |
[+/-] <hour displacement> [: <minute displacement>]

<p> ::= whitespace | . | : | , | - | /

```

Таблица 5. Описание формата даты и времени

Аргумент	Описание
datetime	Строковое представление даты-времени.
date	Строковое представление даты.
time	Строковое представление времени.
YYYY	Год из четырёх цифр.
YY	Последние две цифры года (00-99).
MM	Месяц. Может содержать 1 или 2 цифры (1-12 или 01-12). В качестве месяца допустимо также указывать трёхбуквенное сокращение или полное наименование месяца на английском языке, регистр не имеет значение.
DD	День. Может содержать 1 или 2 цифры (1-31 или 01-31).
HH	Час. Может содержать 1 или 2 цифры (0-23 или 00-23).
mm	Минуты. Может содержать 1 или 2 цифры (0-59 или 00-59).

Аргумент	Описание
SS	Секунды. Может содержать 1 или 2 цифры (0-59 или 00-59).
NNNN	Десятитысячные доли секунды. Может содержать от 1 до 4 цифр (0-9999).
p	Разделитель, любой из разрешённых символов, лидирующие и завершающие пробелы игнорируются.
time zone region	Один из часовых поясов связанных с регионом.
hour displacement	Смещение времени для часов относительно GMT.
minute displacement	Смещение времени для минут относительно GMT.

Правила:

- В формате Год-Месяц-День, год обязательно должен содержать 4 цифры;
- Для дат в формате с завершающим годом, если в качестве разделителя дат используется точка ".", то дата интерпретируется в форме День-Месяц-Год, для остальных разделителей она интерпретируется в форме Месяц-День-Год;
- Если год не указан, то в качестве года берётся текущий год;
- Если указаны только две цифры года, то для получения столетия Firebird использует алгоритм скользящего окна. Задача заключается в интерпретации двухсимвольного значения года как ближайшего к текущему году в интервале предшествующих и последующих 50 лет;
- Если в строковом представлении времени присутствует часовой пояс или смещение времени, то тип литерала будет WITH TIME ZONE, в противном случае WITHOUT TIME ZONE;
- Если не указан один из элементов времени, то оно принимается равным 0.



Настоятельно рекомендуем в литералах дат использовать только формы с полным указанием года в виде 4 цифр во избежание путаницы.

Пример 17. Примеры литералов дат и времени

```

SELECT
  date '04.12.2014' AS d1, -- DD.MM.YYYY
  date '12-04-2014' AS d2, -- MM-DD-YYYY
  date '12/04/2014' AS d3, -- MM/DD/YYYY
  date '04.12.14' AS d4, -- DD.MM.YY
  -- DD.MM в качестве года берётся текущий
  date '04.12' AS d5,
  -- MM/DD в качестве года берётся текущий
  date '12/4' AS d6,
  date '2014/12/04' AS d7, -- YYYY/MM/DD
  date '2014.12.04' AS d8, -- YYYY.MM.DD
  date '2014-12-04' AS d9, -- YYYY-MM-DD
  time '11:37' AS t1, -- HH:mm
  time '11:37:12' AS t2, -- HH:mm:ss
  time '11:31:12.1234' AS t3, -- HH:mm:ss.nnnn
  -- HH:mm:ss.nnnn +hh
  time '11:31:12.1234 +03' AS t4,
  -- HH:mm:ss.nnnn +hh:mm
  time '11:31:12.1234 +03:30' AS t5,
  -- HH:mm:ss.nnnn tz
  time '11:31:12.1234 Europe/Moscow' AS t5,
  -- HH:mm tz
  time '11:31 Europe/Moscow' AS t6,
  -- DD.MM.YYYY HH:mm
  timestamp '04.12.2014 11:37' AS dt1,
  -- MM/DD/YYYY HH:mm:ss
  timestamp '12/04/2014 11:37:12' AS dt2,
  -- DD.MM.YYYY HH:mm:ss.nnnn
  timestamp '04.12.2014 11:31:12.1234' AS dt3,
  -- YYYY-MM-DD HH:mm:ss.nnnn +hh:mm
  timestamp '2014-12-04 11:31:12.1234 +03:00' AS dt4,
  -- DD.MM.YYYY HH:mm:ss.nnnn tz
  timestamp '04.12.2014 11:31:12.1234 Europe/Moscow' AS dt5
FROM rdb$database

```

Обратите внимание, что эти сокращённые выражения вычисляются сразу же во время синтаксического анализа (подготовки запроса или компиляции процедуры, функции или триггера). До Firebird 4.0 сокращённые выражения позволялись также для специальных строковых литералов 'NOW', 'TODAY', 'TOMORROW', 'YESTERDAY'. Использование таких выражений в компилируемом PSQL приводило к тому, что значение "замораживалось" на момент компиляции, и возвращалось не актуальное значение. Поэтому в Firebird 4.0 сокращённые выражения для таких строковых литералов запрещены, однако вы можете использовать их при приведении типа оператором CAST.



См. также:

Преобразование строк в дату и время.

3.4.6. Операции, использующие значения даты и времени

Благодаря способу хранения даты и времени с этими типами возможны арифметические операции вычитания из более поздней даты (времени) более раннюю. Дата представлена количеством дней с "нулевой даты" – 17 ноября 1858 г. Время представлено количеством секунд (с учётом десятитысячных долей), прошедших с полуночи.

Таблица 6. Арифметические операции для типов данных даты и времени

Операнд 1	Оператор	Операнд 2	Результат
DATE	+	TIME	TIMESTAMP
DATE	+	Числовое значение n	DATE, увеличенная на n целых дней (дробная часть игнорируется).
TIME	+	DATE	TIMESTAMP
TIME	+	Числовое значение n	TIME, увеличенное на n секунд (дробная часть учитывается)
TIMESTAMP	+	Числовое значение n	TIMESTAMP, где дни увеличены на целую часть числа n, плюс дробная часть числа n (если указана) как количество секунд в дне (с точностью до десятитысячных долей секунды).
DATE	-	DATE	Количество дней в интервале DECIMAL (9, 0)
DATE	-	Числовое значение n	DATE, уменьшенная на n целых дней (дробная часть игнорируется)
TIME	-	TIME	Количество секунд в интервале DECIMAL (9, 4)
TIME	-	n	TIME, уменьшенное на n секунд (дробная часть учитывается)

Операнд 1	Оператор	Операнд 2	Результат
TIMESTAMP	-	TIMESTAMP	Количество дней и части дня в интервале DECIMAL (18, 9)
TIMESTAMP	-	п	TIMESTAMP, где дни уменьшены на целую часть числа п, плюс дробная часть числа п (если указана) как количество секунд в дне (с точностью до десятитысячных долей секунды).

Одно значение даты/времени может быть вычтено из другого если:

- Оба значения имеют один и тот же тип даты/времени;
- Первый operand является более поздним, чем второй.



В диалекте 1 тип DATE рассматривается как TIMESTAMP.

См. также:

[DATEADD](#), [DATEDIFF](#).

3.5. Символьные типы данных

В СУБД Firebird для работы с символьными данными есть типы фиксированной длины CHAR и переменной длины VARCHAR. Максимальный размер текстовых данных, хранящийся в этих типах данных, составляет 32767 байт для CHAR и 32765 байт для VARCHAR. Максимальное количество символов, которое поместится в этот объём, зависит от используемого набора символов CHARACTER SET. Последовательность сортировки, задаваемая предложением COLLATE, не влияет на этот максимум, хотя может повлиять на максимальный размер любого индекса, который включает столбец.

В случае отсутствия явного указания набора символов при описании текстового объекта базы данных будет использоваться набор символов по умолчанию, заданный при создании базы данных. При отсутствии явного указания набора символов, а также отсутствия набора символов по умолчанию для базы данных, поле получает набор символов CHARACTER SET NONE.

3.5.1. UNICODE

В настоящее время все современные средства разработки поддерживают Unicode. При возникновении необходимости использования восточноевропейских текстов в строковых полях базы данных или для более экзотических алфавитов, рекомендуется работать с набором символов UTF8. При этом следует иметь в виду, что на один символ в данном наборе приходится до 4 байт. Следовательно, максимальное количество символов в

символьных полях составит $32765/4 = 8191$. При этом следует обратить внимание, что фактически значение параметра “bytes per character” зависит от диапазона, к которому принадлежит символ: английские буквы занимают 1 байт, русские буквы—2 байта, остальные символы — могут занимать до 4-х байт.

Набор символов UTF8 поддерживает последнюю версию стандарта Unicode, до 4 байт на символ, поэтому для интернациональных баз рекомендуется использовать именно эту реализацию поддержки Unicode в Firebird.

3.5.2. Набор символов клиента

При работе со строками важно помнить о наборе символов клиентского соединения. В случае различия набора символов, при выдаче результата для строковых столбцов происходит автоматическая перекодировка как при передаче данных с клиента на сервер, так и в обратном направлении с сервера на клиента. То есть, совершенно нормальной является ситуация, когда база создана в кодировке WIN1251, а в настройках клиента в параметрах соединения стоит KOI8R или UTF8.

3.5.3. Специальные наборы символов

Набор символов NONE

Набор символов NONE относится к специальным наборам символов. Его можно охарактеризовать тем, что каждый байт является частью строки, но в системе хранится без указаний, к какому фактическому набору символов они относятся. Разбираться с такими данными должно клиентское приложение, на него возлагается ответственность в правильной трактовке символов из таких полей.

Набор символов OCTETS

Также к специальным наборам символов относится OCTETS. В этом случае данные рассматриваются как байты, которые могут в принципе не интерпретироваться как символы. OCTETS позволяет хранить бинарные данные и/или результаты работы некоторых функций Firebird. Правильное отображение данных пользователю, хранящихся в полях с CHARACTER SET OCTETS, также становится заботой клиентской стороны. При работе с подобными данными следует также помнить, что СУБД не контролирует их содержимое и возможно возникновение исключения при работе кода, когда идёт попытка отображения бинарных данных в желаемой кодировке.

3.5.4. Последовательность сортировки

Каждый набор символов имеет последовательность сортировки (сопоставления) по умолчанию (COLLATE), которая определяет порядок сопоставления. Обычно он обеспечивает упорядочивание на основе числового кода символов и базовое сопоставление символов верхнего и нижнего регистра. Если для строк требуется какое-то поведение, которое не обеспечивается последовательностью сортировки по умолчанию, и для этого набора символов поддерживается подходящее альтернативная сортировка, то в определении столбца можно указать предложение COLLATE collation.

Предложение COLLATE collation может применяться в других контекстах помимо

определения столбца. Для операций сравнения больше/меньше его можно добавить в предложение WHERE оператора SELECT. Если вывод необходимо отсортировать в специальной алфавитной последовательности или без учета регистра и существует соответствующее сопоставление, то предложение COLLATE может быть использовано в предложении ORDER BY, когда строки сортируются по символьному полю, и в предложении GROUP BY в случае групповых операций.

Независимый от регистра поиск

Для независимого от регистра поиска можно воспользоваться функцией UPPER.

Для поиска без учета регистра вы можете воспользоваться функция UPPER для преобразования как аргумента поиска, так и искомых строк в верхний регистр перед попыткой сопоставления.

```
...
WHERE UPPER(name) = UPPER(:flt_name)
```

Для строк в наборе символов, для которых доступна сортировка без учета регистра, вы можете просто применить сопоставление, чтобы напрямую сравнить аргумент поиска и искомые строки. Например, при использовании набора символов WIN1251 вы можете использовать для этой цели сортировку PXW_CYRL не чувствительную к регистру символов.

```
...
WHERE FIRST_NAME COLLATE PXW_CYRL >= :FLT_NAME
...
ORDER BY NAME COLLATE PXW_CYRL
```

См. также:

[CONTAINING](#).

Последовательности сортировки для UTF-8

Ниже приведена таблица возможных последовательностей сортировки для набора символов UTF8.

Таблица 7. Последовательности сортировки для UTF8

COLLATION	Комментарии
UCS_BASIC	Сортировка работает в соответствии с положением символа в таблице (бинарная).
UNICODE	Сортировка работает в соответствии с алгоритмом UCA (Unicode Collation Algorithm) (алфавитная).
UTF-8	По умолчанию используется двоичное сопоставление, идентичное UCS_BASIC, которое было добавлено для совместимости с SQL стандартом.

COLLATION	Комментарии
UNICODE_CI	Сортировка без учета регистра символов.
UNICODE_CI_AI	Сортировка без учета регистра и без учета диакритических знаков в алфавитном порядке.

Пример сортировки строк для набора символов UTF8 без учёта регистра символов и диакритических знаков.

```
ORDER BY NAME COLLATE UNICODE_CI_AI
```

3.5.5. Индексирование символьных типов

При построении индекса по строковым полям необходимо учитывать ограничение на длину ключа индекса. Максимальная используемая длина ключа индекса равна 1/4 размера страницы, то есть от 1024 (для страницы размером 4096) до 8192 байтов (для страницы размером 32768). Максимальная длина индексируемой строки на 9 байтов меньше, чем максимальная длина ключа. В таблице приведены данные для максимальной длины индексируемой строки (в символах) в зависимости от размера страницы и набора символов, её можно вычислить по следующей формуле:

$$\text{max_char_length} = \text{FLOOR}((\text{page_size} / 4 - 9) / N),$$

где N — число байтов на представление символа.

Таблица 8. Длина индексируемой строки и набор символов

Размер страницы	Максимальная длина индексируемой строки для набора символов, байт/символ				
	1	2	3	4	6
4096	1015	507	338	253	169
8192	2039	1019	679	509	339
16384	4087	2043	1362	1021	681
32768	8183	4091	2727	2045	1363



В кодировках, нечувствительных к регистру ("CI"), один символ в _индексе будет занимать не 4, а 6 байт, поэтому максимальная длина ключа для страницы, например для страницы 4096 байт составит 169 символов.

Последовательность сортировки (COLLATE) тоже может повлиять на максимальную длину индексируемой строки. Полный список доступных наборов символов и нестандартных порядков сортировки доступен в приложении [Наборы символов и порядки сортировки](#).

См. также

[CREATE DATABASE](#), Порядок сортировки, [SELECT](#), [WHERE](#), [GROUP BY](#), [ORDER BY](#)

3.5.6. CHAR

CHAR является типом данных фиксированной длины. Если введённое количество символом меньше объявленной длины, то поле дополнится концевыми пробелами. В общем случае символ заполнитель может и не являться пробелом, он зависит от набора символов, так например, для набора символов OCTETS — это ноль.

Полное название типа данных CHARACTER, но при работе нет необходимости использовать полные наименования; инструменты по работе с базой прекрасно понимают и короткие имена символьных типов данных.

Синтаксис:

```
{CHAR | CHARACTER} [(length)]
[CHARACTER SET <charset>] [COLLATE <collate>]
```

В случае если не указана длина, то считается, что она равна единице.

Данный тип символьных данных можно использовать для хранения в справочниках кодов, длина которых стандартна и определённой “ширины”. Примером такого может служить почтовый индекс в России – 6 символов.

3.5.7. VARCHAR

VARCHAR является базовым строковым типом для хранения текстов переменной длины, поэтому реальный размер хранимой структуры равен фактическому размеру данных плюс 2 байта, в которых задана длина поля.

Все символы, которые передаются с клиентского приложения в базу данных, считаются как значимые, включая начальные и конечные пробельные символы.

Полное название CHARACTER VARYING. Имеется и сокращённый вариант записи CHAR VARYING.

Синтаксис

```
{VARCHAR | {CHAR | CHARACTER} VARYING} (length)
[CHARACTER SET <charset>] [COLLATE <collate>]
```

3.5.8. NCHAR

Представляет собой символьный тип данных фиксированной длины с предопределённым набором символов ISO8859_1.

Синтаксис

```
{NCHAR | NATIONAL {CHAR | CHARACTER}} [(length)]
```

Синонимом является написание NATIONAL CHAR.

Аналогичный тип данных доступен для строкового типа переменной длины: NATIONAL CHARACTER VARYING.

3.5.9. Строковые литералы

Строковые литералы могут содержать произвольные символы, допустимые для применяемого набора символов. Весь литерал заключается в апострофы. Апостроф внутри символьного литерала должен повторяться два раза, чтобы отличить его от признака завершения литерала. Максимальная длина строкового литерала составляет 65535 Байт.



Необходимо быть осторожным с длиной строки, если значение должно быть записано в столбец типа VARCHAR. Максимальная длина строки для типа VARCHAR составляет 32765 байт (32767 для типа CHAR). Если значение должно быть записано в столбец типа BLOB, то максимальная длина строкового литерала составляет 65535 байт.

Пример 18. Строковый литерал

```
'Mrs. Hunt''s husband'
```

Альтернативы для апострофов в строковых литералах

Вместо двойного (экранированного) апострофа вы можете использовать другой символ или пару символов.

Ключевое слово q или Q предшествующее строке в кавычках сообщает парсеру, что некоторые левые и правые пары одинаковых символов являются разделителями для встроенного строкового литерала.

Синтаксис:

```
<alternate string literal> ::=  
{ q | Q } <quote> <alternate start char>  
[ { <char> }... ]  
<alternate end char> <quote>
```

Описание

Когда <alternate start char> является одним из символов '(', '{', '<alternate end char>' должен быть использован в паре с соответствующим "партнёром", а именно ')', '}', ']' или '>'. В других случаях <alternate end char> совпадает с <alternate start char>.

Внутри строки, т.е. <char> элементах, одиночные (не экранированные) кавычки могут быть использованы. Каждая кавычка будет частью результирующей строки.

Пример 19. Использование альтернативных апострофов в строковых литералах

```
-- result: abc{def}ghi
SELECT Q'{abc{def}ghi}' FROM rdb$database;

-- result: That's a string
SELECT Q'!That's a string!' FROM rdb$database;
```

Пример 20. Динамическая сборка запроса использующего строковые литералы.

```
EXECUTE BLOCK
RETURNS (
    RDB$TRIGGER_NAME CHAR(64)
)
AS
DECLARE VARIABLE S VARCHAR(8191);
BEGIN
    S = 'SELECT RDB$TRIGGER_NAME FROM RDB$TRIGGERS WHERE RDB$RELATION_NAME IN ';
    S = S || Q'! ('SALES_ORDER', 'SALES_ORDER_LINE')!';
    FOR
        EXECUTE STATEMENT :S
        INTO :RDB$TRIGGER_NAME
    DO
        SUSPEND;
    END
```

3.6. Логический тип данных

В Firebird 3.0 был введён полноценный логический тип данных.

3.6.1. BOOLEAN

SQL-2008 совместимый тип данных BOOLEAN (8 бит) включает различные значения истинности TRUE и FALSE. Если не установлено ограничение NOT NULL, то тип данных BOOLEAN поддерживает также значение истинности UNKNOWN как NULL значение. Спецификация не делает различия между значением NULL этого типа и значением истинности UNKNOWN, которое является результатом SQL предиката, поискового условия или выражения логического типа. Эти значения взаимозаменяемы и обозначают одно и то же.

Как и в других языках программирования, значения типа BOOLEAN могут быть проверены в неявных значениях истинности. Например, field1 OR field2 или NOT field1 являются допустимыми выражениями.

Предикаты могут использовать оператор **IS [NOT]** для проверки соответствия. Например, field1 IS FALSE или field1 IS NOT TRUE.

Оператор IS

Предикаты могут использовать оператор **Логический IS [NOT]** для сопоставления. Например, `field1 IS FALSE`, или `field1 IS NOT TRUE`.



- Операторы эквивалентности (“`=`”, “`!=`”, “`<>`” и др.) допустимы во всех сравнениях.

Примеры BOOLEAN

INSERT u SELECT

```
CREATE TABLE TBOOL (ID INT, BVAL BOOLEAN);
COMMIT;
```

```
INSERT INTO TBOOL VALUES (1, TRUE);
INSERT INTO TBOOL VALUES (2, 2 = 4);
INSERT INTO TBOOL VALUES (3, NULL = 1);
COMMIT;
```

```
SELECT * FROM TBOOL
```

ID	BVAL
1	<true>
2	<false>
3	<null>

Проверка TRUE значения

```
SELECT * FROM TBOOL WHERE BVAL
```

ID	BVAL
1	<true>

Проверка FALSE значения

```
SELECT * FROM TBOOL WHERE BVAL IS FALSE
```

ID	BVAL
2	<false>

Проверка UNKNOWN значения

```
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN
```

ID	BVAL
3	<null>

Логические выражения в SELECT списке

```
SELECT ID, BVAL, BVAL AND ID < 2
FROM TBOOL
```

ID	BVAL
1	<true> <true>
2	<false> <false>
3	<null> <false>

PSQL объявления с начальным значением

```
DECLARE VARIABLE VAR1 BOOLEAN = TRUE;
```

Сравнения с UNKNOWN

```
-- Допустимый синтаксис, но как и сравнение
-- с NULL, никогда не вернёт ни одной записи
SELECT * FROM TBOOL WHERE BVAL = UNKNOWN
SELECT * FROM TBOOL WHERE BVAL <> UNKNOWN
```

Использование Boolean с другими типами данных

Хотя BOOLEAN по своей сути не может быть преобразован в какой-либо другой тип данных, начиная с версии 3.0.1 строки 'true' и 'false' (без учета регистра) будут неявно приводиться к BOOLEAN в выражениях значений, например

```
if (true > 'false') then ...
```

'false' преобразуется в BOOLEAN. Любая попытка использовать логические операторы AND, NOT, OR и IS потерпят неудачу. Например, NOT 'False' приведёт к ошибке.

A BOOLEAN может быть явно преобразован в строку и из нее с помощью CAST. Значение UNKNOWN не доступен при преобразовании к строке.

Другие замечания

- Тип данных BOOLEAN представлен в API типом FB_BOOLEAN и константами FB_TRUE и FB_FALSE.
- Значение TRUE больше чем значение FALSE.

3.7. Бинарные типы данных

3.7.1. BINARY

BINARY является типом данных с фиксированной длиной для хранения бинарных данных. Если переданное количество байт меньше объявленной длины, то значение будет дополнено нулями. В случае если не указана длина, то считается, что она равна единице.

Синтаксис:

```
BINARY [(<length>)]
```



Этот тип является псевдонимом типа CHAR [(<length>)] CHARACTER SET OCTETS и обратно совместим с ним.



Данный тип хорошо подходит для хранения уникального идентификатора полученного с помощью функции [GEN_UUID](#).

См. также:

[CHAR](#), [CHARACTER SET OCTETS](#).

3.7.2. VARBINARY

VARBINARY является типом для хранения бинарных данных переменной длины. Реальный размер хранимой структуры равен фактическому размеру данных плюс 2 байта, в которых задана длина поля.

Полное название BINARY VARYING.

Синтаксис:

```
{VARBINARY | BINARY VARYING} (<length>)
```



Этот тип является псевдонимом типа VARCHAR (<length>) CHARACTER SET OCTETS и обратно совместим с ним.

Пример 21. Использование типов BINARY и VARBINARY в PSQL

```
DECLARE VARIABLE VAR1 VARBINARY(10);
```

Пример 22. Использование типов BINARY и VARBINARY при определении таблицы

```
CREATE TABLE INFO (
    GUID BINARY(16),
    ENCRYPT_KEY VARBINARY(100),
    ICON BINARY VARYING(32000));
```

См. также:

VARCHAR, CHARACTER SET OCTETS.

3.7.3. BLOB

BLOB (Binary Large Objects, большие двоичные объекты) представляют собой сложные структуры, предназначенные для хранения текстовых и двоичных данных неопределённой длины, зачастую очень большого объёма.

Синтаксис:

```
BLOB [SUB_TYPE <subtype>]
[SEGMENT SIZE <seg_length>]
[CHARACTER SET <charset>]
[COLLATE <collation name>]
```

Сокращённый синтаксис:

```
BLOB (<seg_length>)
BLOB (<seg_length>, <subtype>)
BLOB (, <subtype>)
```

Размер сегмента:

Указание размера сегмента BLOB является некоторым атавизмом, оно идёт с тех времён, когда приложения для работы с данными BLOB писались на C (Embedded SQL) при помощи GPRE. В настоящий момент размер сегмента при работе с данными BLOB определяется клиентской частью, причём размер сегмента может превышать размер страницы данных.

Подтипы BLOB

Подтип BLOB отражает природу данных, записанную в столбце. Firebird предоставляет два предопределённых подтипа для сохранения пользовательских данных:

Подтип 0 (BINARY)

Если подтип не указан, то данные считаются нетипизированными и значение подтипа принимается равным 0. Псевдоним подтипа 0 — BINARY. Этот подтип указывает, что данные имеют форму бинарного файла или потока (изображение, звук, видео, файлы текстового процессора, PDF и т.д.).

Подтип 1 (TEXT)

Подтип 1 имеет псевдоним TEXT, который может быть использован вместо указания номера подтипа. Например, BLOB SUBTYPE TEXT. Это специализированный подтип, который используется для хранения текстовых данных большого объёма. Для текстового подтипа BLOB может быть указан набор символов и порядок сортировки COLLATE, аналогично символьному полю.

Пользовательские подтипы

Кроме того, существует возможность добавления пользовательских подтипов данных, для них зарезервирован интервал от -1 до -32768. Пользовательские подтипы с положительными числами не поддерживаются, поскольку Firebird использует числа больше 2 для внутренних подтипов метаданных.

Особенности BLOB

Размер

Максимальный размер поля BLOB ограничен 4Гб и не зависит от варианта сервера, 32 битный или 64 битный (во внутренних структурах, связанных с BLOB присутствуют 4-х байтные счётчики). Для размера страницы 4096 максимальный размер BLOB поля несколько ниже 2 Гб.

Операторы и выражения

Текстовые BLOB любой длины и с любым набором символов (включая multi-byte) могут быть использованы практически с любыми встроенными функциями и операторами.

Полностью поддерживаются следующие операторы:

=	(присвоение)
=, <>, <, <=, >, >=	(сравнение)
	(конкатенация)
BETWEEN,	IS [NOT] DISTINCT FROM,
IN,	ANY SOME,
ALL	

Частично поддерживаются следующие операторы:

- возникает ошибка, в случае если второй аргумент больше или равен 32 Кб

STARTING [WITH], LIKE,
CONTAINING

- Предложения агрегирования работают не с содержимым самого поля, а с идентификатором BLOB ID. Помимо этого, есть некоторые странности:

SELECT	ошибочно выдаёт несколько значений NULL, если они присутствуют
DISTINCT	—
ORDER BY	—
GROUP BY	объединяет одинаковые строки, если они находятся рядом, но не делает этого, если они располагаются вдали друг от друга

Хранение BLOB

- По умолчанию, для каждого BLOB создаётся обычная запись, хранящаяся на какой-то выделенной для этого странице данных (data page). Если весь BLOB на эту страницу поместится, его называют BLOB уровня 0. Номер этой специальной записи хранится в записи таблицы и занимает 8 байт.
- Если BLOB не помещается на одну страницу данных (data page), то его содержимое размещается на отдельных страницах, целиком выделенных для него (blob page), а в записи о BLOB помещают номера этих страниц. Это BLOB уровня 1.
- Если массив номеров страниц с данными BLOB не помещается на страницу данных (data page), то его (массив) размещают на отдельных страницах (blob page), а в запись о BLOB помещают уже номера этих страниц. Это BLOB уровня 2.
- Уровни выше 2 не поддерживаются.

См. также:

[FILTER, DECLARE FILTER](#).

3.7.4. Массивы

Поддержка массивов в СУБД Firebird является расширением традиционной реляционной модели. Поддержка в СУБД такого инструмента позволяет проще решать некоторые задачи по обработке однотипных данных. Массивы в Firebird реализованы на базе полей типа BLOB. Массивы могут быть одномерными и многомерными.

```
CREATE TABLE SAMPLE_ARR (
    ID INTEGER NOT NULL PRIMARY KEY,
    ARR_INT INTEGER [4]);
```

Так будет создана таблица с полем типа массива из четырёх целых. Индексы данного массива от 1 до 4.

Указание явных границ для измерений

По умолчанию размеры начинаются с 1. Для определения верхней и нижней границы значений индекса следует воспользоваться следующим синтаксисом:

[<lower>:<upper>]

Добавление дополнительных измерений

Добавление новой размерности в синтаксисе идёт через запятую. Пример создания таблицы с массивом размерности два, в котором нижняя граница значений начинается с нуля:

```
CREATE TABLE SAMPLE_ARR2 (
    ID INTEGER NOT NULL PRIMARY KEY,
    ARR_INT INTEGER [0:3, 0:3]);
```

Использование массивов

СУБД не предоставляет большого набора инструментов для работы с содержимым массивов. База данных *employee.fdb*, которая находится в дистрибутиве Firebird, содержит пример хранимой процедуры, показывающей возможности работы с массивами. Ниже приведён её текст:

```
CREATE OR ALTER PROCEDURE SHOW_LANGS (
    CODE VARCHAR(5),
    GRADE SMALLINT,
    CTY VARCHAR(15))
RETURNS (
    LANGUAGES VARCHAR(15))
AS
    DECLARE VARIABLE I INTEGER;
BEGIN
    I = 1;
    WHILE (I <= 5) DO
        BEGIN
            SELECT LANGUAGE_REQ[:I]
            FROM JOB
            WHERE (JOB_CODE = :CODE)
                AND (JOB_GRADE = :GRADE)
                AND (JOB_COUNTRY = :CTY)
                AND (LANGUAGE_REQ IS NOT NULL))
            INTO :LANGUAGES;

            IF (:LANGUAGES = '') THEN
                /* PRINTS 'NULL' INSTEAD OF BLANKS */
                LANGUAGES = 'NULL';
            I = I +1;
            SUSPEND;
        END
    END
```

Если приведённых выше возможностей достаточно для ваших задач, то вы вполне можете применять массивы для своих проектов. В настоящее время совершенствования механизмов обработки массивов средствами СУБД не производится.

3.8. Специальные типы данных

3.8.1. Тип данных SQL_NULL

Данный тип данных содержит не данные, а только состояние: NULL или NOT NULL. Также, этот тип данных не может быть использован при объявлении полей таблицы, переменных PSQL, использован в описании параметров. Этот тип данных добавлен для улучшения поддержки нетипизированных параметров в предикате IS NULL. Такая проблема возникает при использовании “отключаемых фильтров” при написании запросов следующего типа:

```
WHERE col1 = :param1 OR :param1 IS NULL
```

после обработки, на уровне API запрос будет выглядеть как

```
WHERE col1 = ? OR ? IS NULL
```

В данном случае получается ситуация, когда разработчик при написании SQL запрос рассматривает :param1 как одну переменную, которую использует два раза, а на уровне API запрос содержит два отдельных и независимых параметра. Вдобавок к этому, сервер не может определить тип второго параметра, поскольку он идёт в паре с IS NULL.

Именно для решения проблемы “? IS NULL” и был добавлен этот специальный тип данных SQL_NULL.

После введения данного специального типа данных при передаче запроса и его параметров на сервер будет работать такая схема: приложение передаёт параметризованные запросы на сервер в виде “?”. Это делает невозможным слияние пары “одинаковых” параметров в один. Так, например, для двух фильтров (двух именованных параметров) необходимо передать четыре позиционных параметра (далее предполагается, что читатель имеет некоторое знакомство с Firebird API):

```
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): установка *sqldata в переданное значение и *sqlind в 0 (для NOT NULL);
- Второй параметр (проверка на NULL): установка *sqldata в null (указатель null, а не SQL NULL) и *sqlind в 0 (для NOT NULL).

Пользователь оставил поле пустым

- Оба параметра (проверка на NULL): установка *sqldata в null (указатель null, а 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.

3.9. Преобразование типов данных

При написании выражения или при задании, например, условий сравнения, нужно стараться использовать совместимые типы данных. В случае необходимости использования смешанных данных различных типов, желательно первоначально выполнить преобразования типов, а уже потом выполнять операции.

При рассмотрении вопроса преобразования типов в Firebird большое внимание стоит уделить тому, в каком диалекте база данных.

3.9.1. Явное преобразование типов данных

В тех случаях, когда требуется выполнить явное преобразование одного типа в другой, используют функцию CAST.

Синтаксис

```
CAST (<expression> | NULL AS <data_type>)

<data_type> ::=  
    sql_datatype  
  | [TYPE OF] domain  
  | TYPE OF COLUMN relname.colname

<datatype> ::=  
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов
```

Преобразование к домену

При преобразовании к домену учитываются объявленные для него ограничения, например, NOT NULL или описанные в CHECK и если <expression> не пройдёт проверку, то преобразование не удастся. В случае если дополнительно указывается TYPE OF (преобразование к базовому типу), при преобразовании игнорируются любые ограничения домена. При использовании TYPE OF с типом [VAR]CHAR набор символов и сортировка сохраняются.

Преобразование к типу столбца

При преобразовании к типу столбца допускается использовать указание столбца таблицы или представления. Используется только сам тип столбца; в случае строковых типов это также включает набор символов, но не сортировку. Ограничения и значения по умолчанию исходного столбца не применяются.

```
CREATE TABLE TTT (
  S VARCHAR (40)
  CHARACTER SET UTF8 COLLATE UNICODE_CI_AI);
COMMIT;

/* У меня много друзей (шведский)*/
SELECT
  CAST ('Jag har många vänner' AS TYPE OF COLUMN TTT.S)
FROM RDB$DATABASE;
```

Допустимые преобразования для функции CAST

Таблица 9. Допустимые преобразования для функции CAST

Из типа	В тип
Числовые типы	Числовые типы, [VAR]CHAR, BLOB
[VAR]CHAR, BLOB	[VAR]CHAR, BLOB, BOOLEAN, Числовые типы, DATE, TIME, TIMESTAMP
DATE, TIME	[VAR]CHAR, BLOB, TIMESTAMP
TIMESTAMP	[VAR]CHAR, BLOB, TIME, DATE
BOOLEAN	[VAR]CHAR, BLOB

Для преобразования строковых типов данных в тип BOOLEAN необходимо чтобы строковый аргумент был одним из предопределённых литералов логического типа ('true' или 'false').



При преобразовании типов следует помнить о возможной частичной потери данных, например, при преобразовании типа данных TIMESTAMP в DATE.

Преобразование строк в дату и время

Для преобразования строковых типов данных в типы DATE, TIME или TIMESTAMP необходимо чтобы строковый аргумент был либо одним из предопределённых литералов даты и времени, либо строковое представление даты в одном из разрешённых форматов.

```

<date_literal> ::=  
[YYYY<p>]MM<p>DD |  
MM<p>DD[<p>YYYY] |  
DD<p>MM[<p>YYYY] |  
MM<p>DD[<p>YY] |  
DD<p>MM[<p>YY]  
  
<time_literal> := HH[:mm[:SS[.NNNN]]]  
  
<datetime_literal> ::= <date_literal> <time_literal>  
  
<time zone> ::=  
  <time zone region> |  
  [+/-] <hour displacement> [: <minute displacement>]  
  
<p> ::= whitespace | . | , | - | /

```

Таблица 10. Описание формата даты и времени

Аргумент	Описание
datetime_literal	Литерал даты-времени.
date_literal	Литерал даты.
time_literal	Литерал времени.
YYYY	Год из четырёх цифр.
YY	Последние две цифры года (00-99).
MM	Месяц. Может содержать 1 или 2 цифры (1-12 или 01-12). В качестве месяца допустимо также указывать трёхбуквенное сокращение или полное наименование месяца на английском языке, регистр не имеет значение.
DD	День. Может содержать 1 или 2 цифры (1-31 или 01-31).
HH	Час. Может содержать 1 или 2 цифры (0-23 или 00-23).
mm	Минуты. Может содержать 1 или 2 цифры (0-59 или 00-59).

Аргумент	Описание
SS	Секунды. Может содержать 1 или 2 цифры (0-59 или 00-59).
NNNN	Десятитысячные доли секунды. Может содержать от 1 до 4 цифр (0-9999).
p	Разделитель, любой из разрешённых символов, лидирующие и завершающие пробелы игнорируются
time zone region	Один из часовых поясов связанных с регионом
hour displacement	Смещение времени для часов относительно GMT
minute displacement	Смещение времени для минут относительно GMT

Таблица 11. Литералы с предопределёнными значениями даты и времени

Литерал	Значение	Тип данных для диалекта 1	Тип данных для диалекта 3
'NOW'	Текущая дата и время	TIMESTAMP	TIMESTAMP
'TODAY'	Текущая дата	TIMESTAMP (с нулевым временем)	DATE (только дата)
'TOMORROW'	Завтрашняя дата	TIMESTAMP (с нулевым временем)	DATE (только дата)
'YESTERDAY'	Вчерашняя дата	TIMESTAMP (с нулевым временем)	DATE (только дата)

Правила:

- В формате Год-Месяц-День, год обязательно должен содержать 4 цифры;
- Для дат в формате с завершающим годом, если в качестве разделителя дат используется точка “.”, то дата интерпретируется в форме День-Месяц-Год, для остальных разделителей она интерпретируется в форме Месяц-День-Год;
- Если год не указан, то в качестве года берётся текущий год;
- Если указаны только две цифры года, то для получения столетия Firebird использует алгоритм скользящего окна. Задача заключается в интерпретации двухсимвольного значения года как ближайшего к текущему году в интервале предшествующих и последующих 50 лет;
- Если не указан один из элементов времени, то оно принимается равным 0.

При использовании преобразования строковых литералов в тип даты/времени с помощью функции CAST() вычисление значения всегда происходит в момент выполнения.

При преобразовании строковых литералов с предопределёнными значениями даты и времени в тип TIMESTAMP точность составляет 3 знака после запятой (миллисекунды).



Настоятельно рекомендуем в литералах дат использовать только формы с полным указанием года в виде 4 цифр во избежание путаницы.

Пример 23. Преобразование строк в дату и время:

```

SELECT
  CAST('04.12.2014' AS DATE) AS d1, -- DD.MM.YYYY
  CAST('12-04-2014' AS DATE) AS d2, -- MM-DD-YYYY
  CAST('12/04/2014' AS DATE) AS d3, -- MM/DD/YYYY
  CAST('04.12.14' AS DATE) AS d4, -- DD.MM.YY
  -- DD.MM в качестве года берётся текущий
  CAST('04.12' AS DATE) AS d5,
  -- MM/DD в качестве года берётся текущий
  CAST('12/4' AS DATE) AS d6,
  CAST('2014/12/04' AS DATE) AS d7, -- YYYY/MM/DD
  CAST('2014.12.04' AS DATE) AS d8, -- YYYY.MM.DD
  CAST('2014-12-04' AS DATE) AS d9, -- YYYY-MM-DD
  CAST('11:37' AS TIME) AS t1, -- HH:mm
  CAST('11:37:12' AS TIME) AS t2, -- HH:mm:ss
  CAST('11:31:12.1234' AS TIME) AS t3, -- HH:mm:ss.nnnn
  -- HH:mm:ss.nnnn +hh
  CAST('11:31:12.1234 +03' AS TIME WITH TIME ZONE) AS t4,
  -- HH:mm:ss.nnnn +hh:mm
  CAST('11:31:12.1234 +03:30' AS TIME WITH TIME ZONE) AS t5,
  -- HH:mm:ss.nnnn tz
  CAST('11:31:12.1234 Europe/Moscow' AS TIME WITH TIME ZONE) AS t5,
  -- HH:mm tz
  CAST('11:31 Europe/Moscow' AS TIME WITH TIME ZONE) AS t6,
  -- DD.MM.YYYY HH:mm
  CAST('04.12.2014 11:37' AS TIMESTAMP) AS dt1,
  -- MM/DD/YYYY HH:mm:ss
  CAST('12/04/2014 11:37:12' AS TIMESTAMP) AS dt2,
  -- DD.MM.YYYY HH:mm:ss.nnnn
  CAST('04.12.2014 11:31:12.1234' AS TIMESTAMP) AS dt3,
  -- YYYY-MM-DD HH:mm:ss.nnnn +hh:mm
  CAST('2014-12-04 11:31:12.1234 +03:00' AS TIMESTAMP WITH TIME ZONE) AS dt4,
  -- DD.MM.YYYY HH:mm:ss.nnnn tz
  CAST('04.12.2014 11:31:12.1234 Europe/Moscow' AS TIMESTAMP WITH TIME ZONE) AS
dt5,
  CAST('now' AS DATE) AS d_now,
  CAST('now' AS TIMESTAMP) AS ts_now,
  CAST('now' AS TIMESTAMP WITH TIME ZONE) AS ts_now_tz,
  CAST('today' AS DATE) AS d_today,
  CAST('today' AS TIMESTAMP) AS ts_today,
  CAST('today' AS TIMESTAMP WITH TIME ZONE) AS ts_today_tz,
  CAST('tomorrow' AS DATE) AS d_tomorrow,
  CAST('tomorrow' AS TIMESTAMP) AS ts_tomorrow,
  CAST('tomorrow' AS TIMESTAMP WITH TIME ZONE) AS ts_tomorrow_tz,
  CAST('yesterday' AS DATE) AS d_yesterday,
  CAST('yesterday' AS TIMESTAMP) AS ts_yesterday,
  CAST('yesterday' AS TIMESTAMP WITH TIME ZONE) AS ts_yesterday_tz
FROM rdb$database

```



Поскольку `CAST('NOW' AS TIMESTAMP)` всегда возвращает актуальные значения даты и времени, то она может использоваться для измерения временных интервалов и скорости выполнения кода в процедурах, триггерах и блоках кода PSQL.

Пример 24. Использование `CAST('NOW' AS TIMESTAMP)` измерения длительности выполнения кода

```
EXECUTE BLOCK
RETURNS (ms BIGINT)
AS
DECLARE VARIABLE t1 TIME;
DECLARE VARIABLE n BIGINT;
BEGIN
    t1 = CAST('now' AS TIMESTAMP);
    /* Долгая операция */
    SELECT COUNT(*) FROM rdb$types, rdb$types, rdb$types INTO n;
    /*=====*/
    ms = DATEDIFF(MILLISECOND FROM t1 TO CAST('now' AS TIMESTAMP));
    SUSPEND;
END
```

См. также:

[Литералы даты и времени, `CAST`.](#)

3.9.2. Неявное преобразование типов данных

В 3-м диалекте невозможно неявное преобразование данных, здесь требуется указывать функцию `CAST` для явной трансляции одного типа в другой. Однако это не относится к операции конкатенации, при которой все другие типы данных будут неявно преобразованы к символьному.

При использовании 1-го диалекта во многих выражениях выполняется неявное преобразование одних типов в другой без применение функции `CAST`. Например, в выражении отбора в диалекте 1 можно записать:

```
WHERE DOC_DATE < '31.08.2014'
```

и преобразование строки в дату произойдёт неявно.

В 1-м диалекте можно смешивать целые данные и числовые строки, строки неявно преобразуются в целое, если это будет возможно, например:

```
2 + '1'
```

корректно выполнится.

В 3-м диалекте подобное выражение вызовет ошибку, в нем потребуется запись следующего вида:

```
2 + CAST('1' AS SMALLINT)
```

Неявное преобразование типов при конкатенации

При конкатенации множества элементов разных типов, все не строковые данные будут неявно преобразованы к строке, если это возможно.

Пример 25. Неявное преобразование типов при конкатенации

```
SELECT 30||' days hath September, April, June and November' CONCAT$  
FROM RDB$DATABASE
```

CONCAT\$

30 days hath September, April, June and November

3.10. Пользовательские типы данных — домены

Домены в СУБД Firebird реализуют широко известный по многим языкам программирования инструмент “типы данных, определённые пользователем”. Когда несколько таблиц в базе данных содержат поля с характеристиками одинаковыми или практически одинаковыми, то есть целесообразность сделать домен, в котором описать набор свойств поля и использовать такой набор свойств, описанный один раз, в нескольких объектах базы данных. Домены могут использоваться помимо описания полей таблиц и представлений (VIEW) и при объявлении входных и выходных параметров, а также при объявлении переменных в коде PSQL.

3.10.1. Атрибуты домена

Определение домена содержит обязательные и необязательные атрибуты. К обязательному атрибуту относится тип данных. К необязательным относятся:

- значение по умолчанию;
- возможности использования NULL для домена;
- ограничения CHECK для данных домена;
- набор символов (для символьных типов данных и BLOB полей);
- порядок сортировки (для символьных типов данных).

Пример 26. Создание домена

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

См. также:

[Явное преобразование типов данных](#), где описаны отличия работы механизма преобразования данных при указании доменов для опций TYPE OF и TYPE OF COLUMN.

3.10.2. Переопределение свойств доменов

При описании таблиц базы данных некоторые свойства столбцов, базирующихся на доменах, могут быть переопределены. Возможности переопределения атрибутов столбцов на базе доменов приведены в таблице.

Таблица 12. Возможности переопределения атрибутов столбцов на базе доменов

Атрибут	Переопределяется?	Примечания
тип данных	нет	
значение по умолчанию	да	
текстовый набор символов	да	также может использоваться, чтобы восстановить для столбца значения по умолчанию для базы данных
текстовый порядок сортировки	да	
условия проверки CHECK	нет	для добавления в проверку новых условий, можно использовать в операторах CREATE и ALTER на уровне таблицы соответствующие предложения CHECK.
NOT NULL	нет	во многих случаях лучше оставить при описании домена возможность значения NULL, а контроль его допустимости осуществлять в описании полей на уровне таблицы.

3.10.3. Создание доменов

Создание домена производится оператором CREATE DOMAIN.

Краткий синтаксис:

```
CREATE DOMAIN <name> [AS] <type>
[DEFAULT {<const> | <literal> | NULL | <context_var>}]
[NOT NULL] [CHECK (<condition>)]
[COLLATE collation];
```

См. также:

[CREATE DOMAIN](#).

3.10.4. Изменение доменов

Для редактирования свойств домена используют оператор `ALTER DOMAIN` языка определения данных (DDL).

*При редактировании домена можно:** переименовать домен;

- изменить тип данных;
- удалить текущее значение по умолчанию;
- установить новое значение по умолчанию;
- установить ограничение NOT NULL;
- удалить ограничение NOT NULL;
- удалить текущее ограничение CHECK;
- добавить новое ограничение CHECK.

Краткий синтаксис:

```
ALTER DOMAIN name
[ {TO new_name} ]
[ {SET DEFAULT {literal | NULL | <context_var>} |  
DROP DEFAULT} ]
[ {SET | DROP} NOT NULL ]
[ {ADD [CONSTRAINT] CHECK (<dom_condition>) |  
DROP CONSTRAINT} ]
[ {TYPE <datatype>} ];
```

Пример 27. Изменение значения по умолчанию для домена

```
ALTER DOMAIN STORE_GRP SET DEFAULT -1;
```

При изменении доменов следует учитывать и его зависимости: имеются ли столбцы таблиц; находятся ли в коде PSQL объявления переменных, входных и/или выходных параметров с типом этого домена. При поспешном редактировании без внимательной проверки можно сделать данный код неработоспособным!

! При смене в домене типа данных не допустимы преобразования, которые могут привести к потере данных. Также, например, при преобразовании VARCHAR в INTEGER проверьте, все ли данные, что используют данных домен, смогут пройти преобразование.

См. также:

[ALTER DOMAIN](#).

3.10.5. Удаление доменов

Оператор DROP DOMAIN удаляет из базы данных домена при условии, что он не используется в каком либо из объектов базы данных.

Синтаксис:

```
DROP DOMAIN name;
```

Пример 28. Удаление домена

```
DROP DOMAIN Test_Domain;
```

См. также:

[DROP DOMAIN](#).

3.11. Синтаксис объявления типа данных

В этом разделе описывается синтаксис объявления типов данных. Объявление типа данных чаще всего встречается в [операторах DDL](#), но также в [CAST](#) и [EXECUTE BLOCK](#).

На приведенный ниже синтаксис есть ссылки из других частей этого руководства.

3.11.1. Синтаксис скалярных типов данных

Скалярные типы данных — это простые типы данных, которые содержат одно значение. Синтаксис типов BLOB рассматривается отдельно в секции [Синтаксисе типов данных BLOB](#).

Синтаксис:

```

<domain_or_non_array_type> ::=

  <scalar_datatype>
  | <blob_datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col

<scalar_datatype> ::=
  {SMALLINT | INT[TEGER] | BIGINT | INT128}
  | BOOLEAN
  | {FLOAT | REAL | DOUBLE PRECISION}
  | [LONG] FLOAT [binary precision]
  | DECFLOAT[({16 | 34})]
  | DATE
  | {TIME | TIMESTAMP} [{WITH | WITHOUT} TIME ZONE]
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {VARCHAR | {CHAR | CHARACTER} VARYING} (length)
    [CHARACTER SET charset]
  | {CHAR | CHARACTER} [(length)] [CHARACTER SET charset]
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} VARYING (length)
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} [(length)]

```

Таблица 13. Параметры декларации скалярных типов

Параметр	Описание
domain	Имя домена (только не домены типа массив).
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления (только столбцы не типа массив).
binary precision	Двоичная точность. От 1 до 64 бит.
precision	Десятичная точность. От 1 до 38 десятичных цифр.
scale	Масштаб или количество знаков после запятой. От 0 до 38. Оно должно быть меньше или равно точности.
length	Максимальная длина строки в символах.
charset	Набор символов.
domain_or_non_array_type	Типы, не являющиеся массивами, которые можно использовать в коде PSQL и операторе CAST.

Использование доменов в объявлениях

Имя домена может быть указано как тип параметра PSQL или локальной переменной. Параметр или переменная наследует все атрибуты домена. Если для параметра или

переменной указано значение по умолчанию, оно переопределяет значение по умолчанию, указанное в определении домена.

Если предложение `TYPE OF` добавлено перед именем домена, то используется только тип данных домена: любые другие атрибуты домена — ограничение `NOT NULL`, ограничение `CHECK`, значение по умолчанию — не проверяются и не используются. Однако, если домен имеет текстовый тип, всегда используются его набор символов и последовательность сортировки.

Использование `TYPE OF COLUMN` в объявлении

Входные и выходные параметры или локальные переменные также могут быть объявлены с использованием типа данных столбцов в существующих таблицах и представлениях. Для этого используется предложение `TYPE OF COLUMN`, в котором в качестве аргумента указывается `[replaceable]relationname.[replaceable]columnname``.

Когда используется `TYPE OF COLUMN`, параметр или переменная наследует только тип данных и — для строковых типов — набор символов и последовательность сортировки. Ограничения и значение столбца по умолчанию игнорируются.

3.11.2. Синтаксис типов данных BLOB

Типы данных `BLOB` содержат данные в двоичном, символьном или пользовательском формате неопределенного размера. Для получения дополнительной информации см. [BLOB](#).

Синтаксис:

```
<blob_datatype> ::=  
    BLOB [SUB_TYPE {subtype_num | subtype_name}]  
        [SEGMENT SIZE seglen] [CHARACTER SET charset]  
    | BLOB [(seglen [, subtype_num])]
```

Таблица 14. Параметры декларации типа `BLOB`

Параметр	Описание
<code>charset</code>	Набор символов (игнорируется для всех подтипов кроме 1 (<code>TEXT</code>)).
<code>subtype_num</code>	Номер подтипа <code>BLOB</code> .
<code>subtype_name</code>	Мнемоническое имя подтипа <code>BLOB</code> ; это может быть <code>TEXT</code> , <code>BINARY</code> или одно из (других) стандартных или настраиваемых имен, определенных в <code>RDB\$TYPES</code> для <code>RDB\$FIELD_NAME = 'RDB\$FIELD_SUB_TYPE'</code> .
<code>seglen</code>	Размер сегмента не может быть больше 65535, по умолчанию — 80, если не указан. Размер сегмента может быть может быть переопределён клиентом и в большинстве случаев не учитывается.

3.11.3. Синтаксис массивов

Тип данных массив содержит несколько скалярных значений в одном или многомерном массиве. Для получения дополнительной информации см. [Тип массив](#).

Синтаксис:

```

<array_datatype> ::= 
    {SMALLINT | INT[ENTER] | BIGINT | INT128} <array_dim>
    | BOOLEAN <array_dim>
    | {FLOAT | REAL | DOUBLE PRECISION} <array_dim>
    | [LONG] FLOAT [binary precision] <array_dim>
    | DECFLOAT[({16 | 34})] <array_dim>
    | DATE <array_dim>
    | {TIME | TIMESTAMP} [{WITH | WITHOUT} TIME ZONE] <array_dim>
    | {DECIMAL | NUMERIC} [(precision [, scale])] <array_dim>
    | {VARCHAR | {CHAR | CHARACTER} VARYING} (length) <array_dim>
        [CHARACTER SET charset]
    | {CHAR | CHARACTER} [(length)] <array_dim> [CHARACTER SET charset]
    | {NCHAR | NATIONAL {CHARACTER | CHAR}} VARYING (length) <array_dim>
    | {NCHAR | NATIONAL {CHARACTER | CHAR}} [(length)] <array_dim>

<array_dim> ::= '[' [:]n [,:n ...] ']'

```

Таблица 15. Параметры декларации массивов

Параметр	Описание
binary precision	Двоичная точность. От 1 до 64 бит.
precision	Десятичная точность. От 1 до 38 десятичных цифр.
scale	Масштаб или количество знаков после запятой. От 0 до 38. Оно должно быть меньше или равно точности.
length	Максимальная длина строки в символах.
charset	Набор символов.
m, n	Целые числа, определяющие диапазон индекса измерения массива.

Глава 4. Общие элементы языка

В этой главе рассматриваются элементы, которые являются общими для всех реализаций языка SQL — выражения, которые используются для извлечения и работают на утверждениях о данных, и предикатов, которые проверяют истинность этих утверждений.

4.1. Выражения

Выражения SQL представляют формальные методы для вычисления, преобразования и сравнения значений. Выражения SQL могут включать в себя столбцы таблиц, переменные, константы, литералы, различные операторы и предикаты, а также другие выражения. Полный список допустимых символов (tokens) в выражениях описан ниже.

Описание элементов языка

Имя столбца

Идентификаторы столбцов из указанных таблиц, используемые в вычислениях, или сравнениях, или в качестве условия поиска. Столбец типа массив не может быть элементом выражения, если только он не проверяется на IS [NOT] NULL.

Элементы массива

В выражении может содержаться ссылка на элемент массива, т.е. `<array_name>[s]`, где `s` — индекс элемента в массиве.

Арифметические операторы

Символы `+`, `-`, `*`, `/` используемые для вычисления значений.

Оператор конкатенации

Оператор `||` (“две вертикальные линии”) используется для соединения символьных строк.

Логические операторы

Зарезервированные слова NOT, AND и OR используются при комбинировании простых условий поиска для создания сложных утверждений.

Операторы сравнения

Символы `=`, `<>`, `!=`, `~`, `^=`, `<`, `<=`, `>`, `>=`, `!<`, `~<`, `^<`, `!>`, `~>` и `^>`

Предикаты сравнения

LIKE, STARTING WITH, CONTAINING, SIMILAR TO, BETWEEN, IS [NOT] NULL, IS [NOT] {TRUE | FALSE | UNKNOWN}, и IS [NOT] DISTINCT FROM

Предикаты существования

Предикаты, используемые для проверки существования значений в наборе. Предикат IN может быть использован как с наборами констант, так и со скалярными подзапросами. Предикаты EXISTS, SINGULAR, ALL ANY, SOME могут быть использованы только с подзапросами.

Константы и литералы

Числа, заключённые в апострофы строковые литералы, логические значения TRUE, FALSE и UNKNOWN, псевдозначение NULL.

Литералы дат и времени

Выражения, подобные строковым литералам, заключённые в апострофах, которые могут быть интерпретированы как значения даты, времени или даты-времени. Литералами дат могут быть строки из символов и чисел, такие как TIMESTAMP '25.12.2016 15:30:35', которые могут быть преобразованы в дату, время или дату с временем.

Мнемоники дат и времени

Строковый литерал с описанием желаемого значения даты и/или времени, которое можно привести к типу даты и/или времени. Для примера 'NOW', 'TODAY'.

Контекстные переменные

Встроенные [контекстные переменные](#).

Локальные переменные

Локальные переменные, входные или выходные параметры PSQL модулей (хранимых процедур, триггеров, анонимных блоков PSQL).

Позиционные параметры

В DSQL в качестве параметров запроса могут быть использованы только позиционные параметры. Позиционные параметры представляют собой знаки вопроса (?) внутри DSQL оператора. Доступ к таким параметрам осуществляется по его номеру (позиции в запросе относительно предыдущего позиционного параметра) поэтому они называются позиционными. Обычно компоненты доступа позволяют работать с именованными параметрами, которые они сами преобразовывают в позиционные.

Подзапросы

Оператор SELECT заключённый в круглые скобки, который возвращает одно единственное (скалярное) значение или множество значений (при использовании в предикатах существования).

Идентификаторы функций

Идентификаторы встроенных или внешних функций в функциональных выражениях.

Приведения типа

Выражение явного преобразования одного типа данных в другой с использованием CAST как CAST(<value> AS <datatype>).

Условные выражения

Выражение CASE и встроенные функции COALESCE, NULLIF.

Круглые скобки

Пара скобок (...) используются для группировки выражений. Операции внутри скобок выполняются перед операциями вне скобок. При использовании вложенных скобок, сначала вычисляются значения самых внутренних выражений, а затем вычисления

перемещаются наверх по уровням вложенности.

Предложение COLLATE

Предложение применяется к типам CHAR и VARCHAR, чтобы в указанной кодировке установить параметры сортировки, используемые при сравнении.

NEXT VALUE FOR sequence

Конструкция NEXT VALUE FOR позволяет получить следующее значение последовательности, то же самое делает встроенная функция GEN_ID().

Выражение AT

Выражение для изменения часового пояса даты и времени.

4.1.1. Литералы (константы)

Литерал или константа — это значение, подставляемое непосредственно в SQL оператор, которое не получено из выражения, параметра, ссылки на столбец или переменной.

Строковые литералы (константы)

Строковый литерал это последовательность символов, заключенных между парой апострофов (“одинарных кавычек”). Максимальная длина строковой константы составляет 65535 байт; максимальное количество символов будет определяться количеством байт, используемых для кодирования каждого символа.

- Двойные кавычки *не должны* (допускаются 1 диалектом) использоваться для квотирования строк. В SQL они предусмотрены для других целей.
- Если литерал апострофа требуется в строковой константе, то он может быть “экранирован” другим предшествующим апострофом. Например,

'Mother O''Reilly's home-made hooch'



- Необходимо быть осторожным с длиной строки, если значение должно быть записано в столбец типа VARCHAR. Максимальная длина строки для типа VARCHAR составляет 32765 байт (32767 для типа CHAR). Если значение должно быть записано в столбец типа BLOB, то максимальная длина строкового литерала составляет 65535 байт.

Предполагается, что набор символов строковой константы совпадает с набором символов столбца предназначенного для её сохранения.

Строковые константы в шестнадцатеричной нотации

Начиная с Firebird 2.5 строковые константы могут быть записаны в шестнадцатеричной нотации, так называемые “двоичные строки”. Каждая пара шестнадцатеричных цифр определяет один байт в строке. Строки введённые таким образом будут иметь кодировку OCTETS по умолчанию, но [вводный синтаксис \(introducer syntax\)](#) может быть использован для

принудительной интерпретации строки в другом наборе символов.

Синтаксис:

```
{x|X}'<hexstring>'  
  
<hexstring> ::= an even number of <hexdigit>  
  
<hexdigit> ::= 0..9 | A..F | a..f
```

Пример 29. Примеры:

```
SELECT x'4E657276656E' FROM rdb$database  
-- returns 4E657276656E, a 6-byte 'binary' string  
  
SELECT _ascii x'4E657276656E' FROM rdb$database  
-- returns 'Nerven' (same string, now interpreted as ASCII text)  
  
SELECT _iso8859_1 x'53E46765' FROM rdb$database  
-- returns 'Säge' (4 chars, 4 bytes)  
  
SELECT _utf8 x'53C3A46765' FROM rdb$database  
-- returns 'Säge' (4 chars, 5 bytes)
```

Как будут отображены двоичные строки зависит от интерфейса клиента. Например, утилита `isql` использует заглавные буквы A-F, в то время как `FlameRobin` буквы в нижнем регистре. Другие могут использовать другие правила конвертирования, например отображать пробелы между парами байт: '4E 65 72 76 65 6E'.



Шестнадцатеричная нотация позволяет вставить любой байт (включая 00) в любой позиции в строке.

Альтернативы для апострофов в строковых литералах

Вместо двойного (экранированного) апострофа вы можете использовать другой символ или пару символов.

Ключевое слово `q` или `Q` предшествующее строке в кавычках сообщает парсеру, что некоторые левые и правые пары одинаковых символов являются разделителями для встроенного строкового литерала.

Синтаксис:

```
<alternate string literal> ::=  
{ q | Q } <quote> <alternate start char>  
[ { <char> }... ]  
<alternate end char> <quote>
```

Правила использования

Когда `<alternate start char>` является одним из символов '(', '{', '`<alternate end char>`' должен быть использован в паре с соответствующим "партнёром", а именно ')', '}', ']' или '>'. В других случаях `<alternate end char>` совпадает с `<alternate start char>`.



Внутри строки, т.е. `<char>` элементах, одиночные (не экранированные) кавычки могут быть использованы. Каждая кавычка будет частью результирующей строки.

Пример 30. Использование альтернативных апострофов в строковых литералах

```
-- result: abc{def}ghi  
SELECT Q'{abc{def}ghi}' FROM rdb$database;  
  
-- result: That's a string  
SELECT Q'!That's a string!' FROM rdb$database;
```

Пример 31. Динамическая сборка запроса использующего строковые литералы.

```
EXECUTE BLOCK  
RETURNS (  
    RDB$TRIGGER_NAME CHAR(64)  
)  
AS  
    DECLARE VARIABLE S VARCHAR(8191);  
BEGIN  
    S = 'SELECT RDB$TRIGGER_NAME FROM RDB$TRIGGERS WHERE RDB$RELATION_NAME IN ';  
    S = S || Q'! ('SALES_ORDER', 'SALES_ORDER_LINE')!';  
    FOR  
        EXECUTE STATEMENT :S  
        INTO :RDB$TRIGGER_NAME  
    DO  
        SUSPEND;  
END
```

Вводный синтаксис для строковых литералов

При необходимости, строковому литералу может предшествовать имя набор символов, который начинается с префикса подчеркивания “_”. Это известно как вводный синтаксис (Introducer syntax). Его цель заключается в информировании Firebird о том, как интерпретировать и хранить входящую строку.

Пример 32. Вводный синтаксис для строковых литералов

```
-- обратите внимание на префикс '_'
INSERT INTO People
VALUES (_ISO8859_1 'Hans-Jörg Schäfer');
```

Числовые константы

Числовая константа — это любое правильное число в одной из поддерживаемых нотаций:

- В SQL, для чисел в стандартной десятичной записи, десятичная точка всегда представлена символом точки и тысячи не разделены. Включение запятых, пробелов, и т.д. вызовет ошибки.
- Экспоненциальная запись, например число 0.0000234 может быть записано как 2.34e-5.
- Шестнадцатеричная запись (см. ниже) чисел поддерживается начиная с Firebird 2.5.

Далее показаны форматы числовых литералов и их типы. Где <d> - десятичная цифра, <h> - шестнадцатеричная цифра.

Таблица 16. Формат числовых констант

Формат	Тип
<d>[<d> ...]	INTEGER или BIGINT.
0{x X} <h><h>[<h><h> ...]	INTEGER для 1-8 <h><h> пар или BIGINT для 9-16 пар.
<d>[<d> ...].[<d> ...]	NUMERIC(18, n) где n зависит от количества цифр после десятичной точки.
<d>[<d> ...][. [<d> ...]] E <d>[<d> ...]	DOUBLE PRECISION.

Шестнадцатеричная нотация чисел

Начиная с Firebird 2.5 целочисленные значения могут быть записаны в шестнадцатеричной системе счисления. Числа состоящие из 1-8 шестнадцатеричных цифр будут интерпретированы как INTEGER, состоящие из 9-16 цифр — как BIGINT.

Синтаксис:

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

Пример 33. Шестнадцатиричные константы

```
SELECT 0x6FAA0D3 FROM rdb$database -- returns 117088467
SELECT 0x4F9 FROM rdb$database -- returns 1273
SELECT 0x6E44F9A8 FROM rdb$database -- returns 1850014120
SELECT 0x9E44F9A8 FROM rdb$database -- returns -1639646808 (an INTEGER)
SELECT 0x09E44F9A8 FROM rdb$database -- returns 2655320488 (a BIGINT)
SELECT 0x28ED678A4C987 FROM rdb$database -- returns 720001751632263
SELECT 0xFFFFFFFFFFFFFF FROM rdb$database -- returns -1
```

Диапазон значений шестнадцатиричных чисел

- Шестнадцатиричные числа в диапазоне 0 .. 7FFF FFFF являются положительными INTEGER числа со значениями 0 .. 2147483647. Для того чтобы интерпретировать константу как BIGINT число, необходимо дописать необходимое количество нулей слева. Это изменит тип, но не значение.
- Числа в диапазоне 8000 0000 .. FFFF FFFF требуют особого внимания:
 - При записи восемью шестнадцатиричными числами, такие как 0x9E44F9A8, интерпретируются как 32-битное целое. Поскольку крайний левый (знаковый) бит установлен, то такие числа будут находиться в отрицательном диапазоне -2147483648 .. -1.
 - Числа предварённые одним или несколькими нулями, такие как 0x09E44F9A8, будут интерпретированы как 64-разрядный BIGINT в диапазоне значений 0000 0000 8000 0000 .. 0000 0000 FFFF FFFF. В этом случае знаковый бит не установлен, поэтому они отображаются в положительном диапазоне 2147483648 .. 4294967295 десятичных чисел.

Таким образом, только в этом диапазоне числа, предварённые совершенно незначимым нулём, имеют кардинально разные значения. Это необходимо знать.

- Шестнадцатиричные числа в диапазоне 1 0000 0000 .. 7FFF FFFF FFFF FFFF являются положительными BIGINT числами.
- Шестнадцатиричные числа в диапазоне 8000 0000 0000 0000 .. FFFF FFFF FFFF FFFF являются отрицательными BIGINT числами.
- Числа с типом SMALLINT не могут быть записаны в шестнадцатиричном виде, строго говоря, так как даже 0x1 оценивается как INTEGER. Тем не менее, если вы записываете положительное целое число в пределах 16-разрядного диапазона от 0x0000 (десятичный

ноль) до `0xFFFF` (десятичное 32767), то оно будет преобразовано в `SMALLINT` прозрачно.

Вы можете записать отрицательное `SMALLINT` число в шестнадцатеричном виде используя 4-байтное шестнадцатеричное число в диапазоне от `0xFFFF8000` (десятичное -32768) до `0xFFFFFFFF` (десятичное -1).

Логические литералы

Логический литерал может быть одним из следующих значений: `TRUE`, `FALSE` или `UNKNOWN`.

4.1.2. Операторы SQL

SQL операторы включают в себя операторы для сравнения, вычисления, оценки и конкатенации значений.

Приоритет операторов

Приоритет определяет порядок, в котором операторы и получаемые с помощью них значения вычисляются в выражении.

Все операторы разбиты на 4 типа. Каждый тип оператора имеет свой приоритет. Чем выше приоритет типа оператора, тем раньше он будет вычислен. Внутри одного типа операторы имеют собственный приоритет, который также определяет порядок их вычисления в выражении. Операторы с одинаковым приоритетом вычисляются слева направо. Для изменения порядка вычислений операции могут быть сгруппированы с помощью круглых скобок.

Таблица 17. Приоритеты типов операторов

Тип оператора	Приоритет	Пояснение
Конкатенация	1	Строки объединяются до выполнения любых других операций.
Арифметический	2	Арифметические операции выполняются после конкатенации строк, но перед выполнением операторов сравнения и логических операций.
Сравнение	3	Операции сравнения вычисляются после конкатенации строк и выполнения арифметических операций, но до логических операций.
Логический	4	Логические операторы выполняются после всех других типов операторов.

Оператор конкатенации

Оператор конкатенации `||` соединяет две символьные строки и создаёт одну строку. Символьные строки могут быть константами или значениями, полученными из столбцов или других выражений.

Пример 34. Оператор конкатенации

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME
FROM EMPLOYEE
```

Арифметические операторы

Таблица 18. Приоритет арифметических операторов

Оператор	Назначение	Приоритет
+signed_number	Унарный плюс	1
-signed_number	Унарный минус	1
*	Умножение	2
/	Деление	2
+	Сложение	3
-	Вычитание	3

Пример 35. Арифметические операторы

```
UPDATE T
SET A = 4 + 1/(B-C)*D
```

Операторы сравнения

Таблица 19. Операторы сравнения

Оператор	Назначение	Приоритет
IS	Проверяет, что выражение в левой части является псевдо значением NULL или соответствует логическому значению в правой части.	1
=	Равно, идентично	2
<>, !=, ~=, ^=	Не равно	2
>	Больше	2
<	Меньше	2
>=	Больше или равно	2
<=	Меньше или равно	2
!>, ~>, ^>	Не больше	2
!<, ~<, ^<	Не меньше	2

В эту же группу входят предикаты сравнения **IS DISTINCT FROM**, **BETWEEN**, **IN**, **LIKE**, **CONTAINING**, **SIMILAR TO** и другие.

Пример 36. Использование оператора сравнения

```
IF (SALARY > 1400) THEN
  ...
```

См. также:

[Другие предикаты сравнения](#).

Логические операторы

Таблица 20. Приоритет логических операторов

Оператор	Назначение	Приоритет
NOT	Отрицание условия поиска.	1
AND	Объединяет два предиката и более, каждый из которых должен быть истинным, чтобы истинным был и весь предикат.	2
OR	Объединяет два предиката и более, из которых должен быть истинным хотя бы один предикат, чтобы истинным был и весь предикат.	3

Пример 37. Использование логических операторов

```
IF (A > B OR (A > C AND A > D) AND NOT (C = D)) THEN
  ...
```

4.1.3. AT

Доступно в

DSQL, PSQL.

Синтаксис

```

<expr> AT {TIME ZONE <time zone string> | LOCAL}

<time zone string> ::= 
    '<time zone>'

<time zone> ::= 
    <time zone region> |
    [+/-] <hour displacement> [: <minute displacement>]

```

Преобразует время или временную метку в указанный часовой пояс. Если используется ключевое слово LOCAL, то преобразование происходит в часовой пояс сессии.

Пример 38. Использование функции AT

```

select time '12:00 GMT' at time zone '-03'
  from rdb$database;

select current_timestamp at time zone 'America/Sao_Paulo'
  from rdb$database;

select timestamp '2018-01-01 12:00 GMT' at local
  from rdb$database;

```

4.1.4. NEXT VALUE FOR

Доступно в DSQL, PSQL.

Синтаксис

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

Возвращает следующее значение в последовательности (SEQUENCE). SEQUENCE является SQL совместимым термином генератора в InterBase и Firebird. Оператор NEXT VALUE FOR полностью эквивалентен функции GEN_ID (sequence-name, 1) и является рекомендуемым синтаксисом.

 NEXT VALUE FOR не поддерживает значение приращения, отличное от того, что было указано при создании последовательности в предложении INCREMENT [BY]. Если требуется другое значение шага, то используйте старую функцию GEN_ID.

Пример 39. Использование NEXT VALUE FOR

```
NEW.CUST_ID = NEXT VALUE FOR CUSTSEQ;
```

См. также:

[SEQUENCE \(GENERATOR\), GEN_ID.](#)

4.1.5. Условные выражения

Условное выражение — это выражение, которое возвращает различные значения в зависимости от истинности некоторого условия или условий. В данном разделе описано лишь одно условное выражение CASE. Остальные условные выражения являются производными встроенными функциями и описаны в разделе [Условные функции](#).

CASE

Доступно в

DSQL, ESQL.

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 первым *expr*, затем вторым *expr* и так далее, 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, то возвращается значение NULL.

Совпадение эквивалентно оператору “=”, то есть если *test-expr* имеет значение NULL, то он не соответствует ни одному из *expr*, даже тем, которые имеют значение NULL.

Результаты необязательно должны быть литеральными значениями, они также могут быть именами полей, переменными, сложными выражениями или NULL.

Пример 40. Использование простого CASE

```

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

Сокращённый вид простого оператора CASE используется в функции DECODE.

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 обязаны быть литеральными значениями: они могут быть полями или именами переменных, сложными выражениями, или NULL.

Пример 41. Использование поискового CASE

```

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

4.1.6. NULL в выражениях

NULL не является значением — это состояние, указывающее, что значение элемента неизвестно или не существует. Это не ноль, не пустота, не “пустая строка”, и оно не ведёт себя как какое-то из этих значений.

При использовании `NULL` в числовых, строковых выражениях или в выражениях, содержащих дату/время, в результате вы всегда получите `NULL`. При использовании `NULL` в логических (булевых) выражениях результат будет зависеть от типа операции и других вовлечённых значений. При сравнении значения с `NULL` результат будет неопределённым (`UNKNOWN`).



Неопределённый логический результат `UNKNOWN` тоже представлен псевдо-значением `NULL`.

Выражения возвращающие `NULL`

Выражения в этом списке всегда возвратят `NULL`:

```
1 + 2 + 3 + NULL
'Home' || 'sweet' || NULL
MyField = NULL
MyField <> NULL
NULL = NULL
not (NULL)
```

Если вам трудно понять, почему, вспомните, что `NULL` — значит “неизвестно”.

NULL в логических выражениях

Мы уже рассмотрели, что `not (NULL)` даёт в результате `NULL`. Для операторов `AND` и `OR` взаимодействие несколько сложнее:

```
NULL or false → NULL
NULL or true → true
NULL or NULL → NULL
NULL and false → false
NULL and true → NULL
NULL and NULL → NULL
```

Пример 42. NULL в логических выражениях

```
(1 = NULL) or (1 <> 1)      -- returns NULL
(1 = NULL) or FALSE          -- returns NULL
(1 = NULL) or (1 = 1)          -- returns TRUE
(1 = NULL) or TRUE            -- returns TRUE
(1 = NULL) or (1 = NULL)       -- returns NULL
(1 = NULL) or UNKNOWN         -- returns NULL
(1 = NULL) and (1 <> 1)        -- returns FALSE
(1 = NULL) and FALSE          -- returns FALSE
(1 = NULL) and (1 = 1)          -- returns NULL
(1 = NULL) and TRUE            -- returns NULL
(1 = NULL) and (1 = NULL)       -- returns NULL
(1 = NULL) and UNKNOWN         -- returns NULL
```

4.2. Подзапросы

Подзапрос — это специальный вид выражения, которое фактически является запросом, встроенным в другой запрос. Подзапросы пишутся как обычные SELECT запросы, но должны быть заключены в круглые скобки. Выражения подзапроса используется следующими способами:

- Для задания выходного столбца в списке выбора SELECT;
- Для получения значений или условий для предикатов поиска (предложения WHERE, HAVING);
- Для создания набора данных, из которого включающий запрос может выбирать, как будто это обычная таблица или представление. Подобные подзапросы появляются в предложении FROM (производные таблицы) или в общем табличном выражении (CTE).

4.2.1. Коррелированные подзапросы

Подзапрос может быть коррелированным (соотнесённым). Запрос называется коррелированным, когда подзапрос и основной запрос взаимозависимы. Это означает, что для обработки каждой записи подзапроса, должна быть получена также запись из основного запроса, т.е. подзапрос всецело зависит от основного запроса.

Пример 43. Коррелированный подзапрос

```
SELECT *
FROM Customers C
WHERE EXISTS
  (SELECT *
   FROM Orders O
   WHERE C.cnum = O.cnum
   AND O.adate = DATE '10.03.1990');
```

При использовании подзапросов для получения значений выходного столбца в списке выбора SELECT, подзапрос должен возвращать скалярный результат.

4.2.2. Подзапросы возвращающие скалярный результат

Подзапросы, используемые в предикатах поиска, кроме предикатов существования и количественных предикатов, должны возвращать скалярный результат, то есть не более чем один столбец из одной отобранный строки или одно агрегированное значение, в противном случае, произойдёт ошибки времени выполнения (“Multiple rows in a singleton select...”).



Несмотря на то, что Firebird сообщает о подлинной ошибке, сообщение может немного вводить в заблуждение. “singleton SELECT” — это запрос, который не должен возвращать более одной строки. Однако “singleton” и “scalar” не являются синонимами: не все одноэлементные SELECTS должны быть скалярными; а выборка по одному столбцу может возвращать несколько строк для предикатов существования и количественных предикатов.

Пример 44. Подзапрос в качестве выходного столбца в списке выбора

```
SELECT
  e.first_name,
  e.last_name,
  (SELECT
    sh.new_salary
   FROM
    salary_history sh
   WHERE
    sh.emp_no = e.emp_no
   ORDER BY sh.change_date DESC ROWS 1) AS last_salary
FROM
  employee e
```

Пример 45. Подзапрос в предложении WHERE для получения значения максимальной зарплаты сотрудника и фильтрации по нему

```

SELECT
    e.first_name,
    e.last_name,
    e.salary
FROM
    employee e
WHERE
    e.salary = (SELECT
                  MAX(ie.salary)
              FROM
                  employee ie)

```

4.3. Предикаты

Предикат — это простое выражение, утверждающее некоторый факт, назовем его P . Если P разрешается как TRUE, он успешен. Если он принимает значение FALSE или NULL (UNKNOWN), он терпит неудачу. Однако здесь кроется ловушка: предположим, что предикат P возвращает FALSE. В этом случае NOT (P) вернет TRUE. С другой стороны, если P возвращает NULL (неизвестно), то NOT (P) также возвращает NULL.

В SQL предикаты проверяют в ограничении CHECK, предложении WHERE, выражении CASE, условии соединения во фразе ON для предложений JOIN, а также в предложении HAVING. В PSQL операторы управления потоком выполнения проверяют предикаты в предложениях IF, WHILE и WHEN. Поскольку начиная с Firebird 3.0 введена поддержка логического типа, то предикат может встречаться в любом правильном выражении.

4.3.1. Утверждения

Проверяемые условия не всегда являются простыми предикатами. Они могут быть группой предикатов, каждый из которых при вычислении делает вклад в вычислении общей истинности. Такие сложные условия называются утверждениями. Утверждения могут состоять из одного или нескольких предикатов, связанных логическими операторами AND, OR и NOT. Для группировки предикатов и управления порядком вычислений можно использовать скобки.

Каждый из предикатов может содержать вложенные предикаты. Результат вычисления истинности утверждения получается в результате вычисления всех предикатов по направлению от внутренних к внешним. Каждый “уровень” вычисляется в порядке приоритета до тех пор, пока не будет получено значение истинности окончательного утверждения.

4.3.2. Предикаты сравнения

Предикат сравнения представляет собой два выражения, соединяемых оператором сравнения. Имеется шесть традиционных операторов сравнения:

=, >, <, >=, <=, <>

(Полный список операторов сравнения см. [Операторы сравнения](#)).

Если в одной из частей (левой или правой) предиката сравнения встречается NULL, то значение предиката будет неопределённым (UNKNOWN).

Пример 46. Предикаты сравнения

Получить информацию о компьютерах, имеющих частоту процессора не менее 500 МГц и цену ниже \$800

```
SELECT *
FROM Pc
WHERE speed >= 500 AND price < 800;
```

Получить информацию обо всех принтерах, которые являются матричными и стоят меньше \$300

```
SELECT *
FROM Printer
WHERE type = 'matrix' AND price < 300;
```

Следующий запрос не вернёт ни одной записи, поскольку сравнение происходит с псевдо-значением NULL, даже если существуют принтеры с неуказанным типом.

```
SELECT *
FROM Printer
WHERE type = NULL AND price < 300;
```

Замечание о сравнении строк



При сравнении на равенство полей типов CHAR и VARCHAR завершающий пробелы игнорируются во всех случаях.

4.3.3. Другие предикаты сравнения

Другие предикаты сравнения состоят из ключевых слов.

BETWEEN

Доступно в
DSQL, PSQL, ESQL.

Синтаксис

```
<value> [NOT] BETWEEN <value_1> AND <value_2>
```

Предикат BETWEEN проверяет, попадает (или не попадает при использовании NOT) ли значение во включающий диапазон значений.

Операнды для предиката BETWEEN — это два аргумента совместимых типов. В отличие от некоторых других СУБД в Firebird предикат BETWEEN не является симметричным. Меньшее значение должно быть первым аргументом, иначе предикат BETWEEN всегда будет ложным. Поиск является включающим. Таким образом, предикат BETWEEN можно переписать следующим образом:

```
<value> >= <value_1> AND <value> <= <value_2>
```

При использовании предиката BETWEEN в поисковых условиях DML запросов, оптимизатор Firebird может использовать индекс по искомому столбцу, если таковой доступен.

Пример 47. Использование предиката BETWEEN

```
SELECT *
FROM EMPLOYEE
WHERE HIRE_DATE BETWEEN date '01.01.1992' AND CURRENT_DATE
```

LIKE

Доступно в
DSQL, PSQL, ESQL.

Синтаксис

```
<match value> [NOT] LIKE <pattern>
[ESCAPE <escape character>]

<match value> ::= выражение символьного типа
<pattern> ::= шаблон поиска
<escape character> ::= символ экранирования
```

Предикат LIKE сравнивает выражение символьного типа с шаблоном, определённым во втором выражении. Чувствительность к регистру или диакритическим знакам при сравнении определяется используемым параметром сортировки (COLLATION).

При использовании оператора LIKE во внимание принимаются все символы строки-шаблона. Это касается так же начальных и конечных пробелов. Если операция сравнения в запросе должна вернуть все строки, содержащие строки LIKE 'абв' (с символом пробела на конце), то строка, содержащая 'абв' (без пробела), не будет возвращена.

Трафаретные символы

В шаблоне, разрешается использование двух трафаретных символов:

- символ процента (%) заменяет последовательность любых символов (число символов в последовательности может быть от 0 и более) в проверяемом значении;
- символ подчёркивания (_), который можно применять вместо любого единичного символа в проверяемом значении.

Если проверяемое значение соответствует образцу с учётом трафаретных символов, то предикат истинен.

Использование управляющего символа в предложении ESCAPE

Если искомая строка содержит трафаретный символ, то следует задать управляющий символ в предложении ESCAPE. Этот управляющий символ должен использоваться в образце перед трафаретным символом, сообщая о том, что последний следует трактовать как обычный символ.

Примеры использования предиката LIKE

Пример 48. Поиск строк начинающихся с заданной подстроки с использованием предиката LIKE

Поиск номеров отделов, названия которых начинаются со слова “Software”

```
SELECT DEPT_NO
  FROM DEPT
 WHERE DEPT_NAME LIKE 'Software%';
```

В данном запросе может быть использован индекс, если он построен на поле DEPT_NAME.

Оптимизация LIKE

В общем случае предикат LIKE не использует индекс. Однако если предикат принимает вид LIKE 'string%', то он будет преобразован в предикат STARTING WITH, который будет использовать индекс. Если вам необходимо выполнить поиск с начала строки, то вместо предиката LIKE рекомендуется использовать предикат STARTING WITH.



Пример 49. Использование трафаретного символа “_” в предикате LIKE

Поиск сотрудников, имена которых состоят из 5 букв, начинающихся с букв “Sm” и заканчивающихся на “th”. В данном случае предикат будет истинен для имен “Smith” и “Smyth”.

```
SELECT
    first_name
FROM
    employee
WHERE first_name LIKE 'Sm_th'
```

Пример 50. Поиск внутри строки с использованием предиката LIKE

Поиск всех заказчиков, в адресе которых содержится строка “Ростов”.

```
SELECT *
FROM CUSTOMER
WHERE ADDRESS LIKE '%Ростов%'
```



Если вам необходимо выполнить поиск внутри строки, то вместо предиката LIKE рекомендуется использовать предикат [CONTAINING](#).

Пример 51. Использование управляющего символа в предложении ESCAPE с предикатом LIKE

Поиск таблиц, содержащих в имени знак подчёркивания. В данном случае в качестве управляющего символа задан символ “#”.

```
SELECT
    RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME LIKE '%#_%' ESCAPE '#'
```

См. также:

[STARTING WITH](#), [CONTAINING](#), [SIMILAR TO](#).

STARTING WITH

Доступно в

DSQL, PSQL, ESQL.

Синтаксис

```
<value> [NOT] STARTING WITH <start-value>
```

Предикат `STARTING WITH` ищет строку, которая начинается с символов в его аргументе `start-value`. Чувствительность к регистру и ударению в `STARTING WITH` зависит от сортировки (COLLATION) первого аргумента `value`.

При использовании предиката `STARTING WITH` в поисковых условиях DML запросов, оптимизатор Firebird может использовать индекс по искомому столбцу, если он определён.

Пример 52. Использование предиката `STARTING WITH`

Поиск сотрудников, фамилия которых начинается с “Jo”.

```
SELECT LAST_NAME, FIRST_NAME
  FROM EMPLOYEE
 WHERE LAST_NAME STARTING WITH 'Jo'
```

См. также:

[LIKE](#).

CONTAINING

Доступно в

DSQL, PSQL, ESQL.

Синтаксис

```
<value> [NOT] CONTAINING <substring>
```

Оператор `CONTAINING` ищет строку или тип, подобный строке, отыскивая последовательность символов, которая соответствует его аргументу. Он может быть использован для алфавитно-цифрового (подобного строковому) поиска в числах и датах. Поиск `CONTAINING` не чувствителен к регистру. Тем не менее, если используется сортировка чувствительная к акцентам, то поиск будет чувствителен к акцентам.

При использовании оператора `CONTAINING` во внимание принимаются все символы строки. Это касается так же начальных и конечных пробелов. Если операция сравнения в запросе должна вернуть все строки, содержащие строки `CONTAINING 'абв '` (с символом пробела на конце), то строка, содержащая `'абв'` (без пробела), не будет возвращена.

При использовании предиката `CONTAINING` в поисковых условиях DML запросов, оптимизатор Firebird не может использовать индекс по искомому столбцу.

Пример 53. Поиск подстроки с использованием предиката CONTAINING

Поиск проектов в именах, которых присутствует подстрока “Map”:

```
SELECT *
FROM PROJECT
WHERE PROJ_NAME CONTAINING 'map';
```

В данном случае будет возвращены две строки с именами “AutoMap” и “MapBrowser port”.

Пример 54. Поиск внутри даты с использованием предиката CONTAINING

Поиск записей об изменении зарплат с датой содержащей число 84 (в данном случае изменения, которые произошли в 1984 году):

```
SELECT *
FROM SALARY_HISTORY
WHERE CHANGE_DATE CONTAINING 84;
```

См. также:

[LIKE](#).

SIMILAR TO

Доступно в

DSQL, PSQL.

Синтаксис

```
string-expression [NOT] SIMILAR TO <pattern> [ESCAPE <escape-char>]
```

<pattern> ::= регулярное выражение SQL

<escape-char> ::= символ экранирования

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 недостаточно. Если один из операндов имеет значение NULL, то и результат будет NULL. В противном случае результат является TRUE или FALSE.

Синтаксис регулярных выражений SQL

Следующий синтаксис определяет формат регулярного выражения SQL. Это полное и корректное его определение. Он является весьма формальным и довольно длинным и,

вероятно, озадачивает тех, кто не имеет опыта работы с регулярными выражениями. 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> ::= целые положительные числа, если присутствуют оба числа, то <m> <= <n>

<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](#)

В этом разделе представлены элементы и правила построения регулярных выражений SQL.

Символы

В регулярных выражениях большинство символов представляет сами себя, за исключением специальных символов (special character):

```
[ ] ( ) | ^ - + * % _ ? { }
```

... и управляющих символов (escaped character), 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_гне' -- TRUE
'Birne' SIMILAR TO 'B_нe' -- FALSE
'Birne' SIMILAR TO 'B%нe' -- TRUE
'Birne' SIMILAR TO 'Bir%нe%' -- TRUE
'Birne' SIMILAR TO 'Birг%нe' -- 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[арю]oen' -- TRUE
'Citroen' SIMILAR TO 'Ci[тr]oen' -- FALSE
'Citroen' SIMILAR TO 'Ci[тr][тr]oen' -- TRUE
```

Как видно из второй строки классу только соответствует единственный символ, а не их последовательность.

Два символа, соединённые 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
```

Предопределённые классы символов

Следующие предопределенные классы символов также могут использоваться в определении класса:

[:ALPHA:]

Латинские буквы a...z и A...Z. Этот класс также включает символы с диакритическими знаками при нечувствительных к акцентам сортировках.

[:DIGIT:]

Десятичные цифры 0...9.

[:ALNUM:]

Объединение [:ALPHA:] и [:DIGIT:].

[:UPPER:]

Прописные (в верхнем регистре) латинские буквы A...Z. Также включает в себя символы в нижнем регистре при нечувствительных к регистру сортировках и символы с диакритическими знаками при нечувствительных к акцентам сортировках.

[:LOWER:]

Строчные (в нижнем регистре) латинские буквы a...z. También включает в себя символы в верхнем регистре при нечувствительных к регистру сортировках и символы с диакритическими знаками при нечувствительных к акцентам сортировках.

[:SPACE:]

Символ пробела (ASCII 32).

[:WHITE SPACE:]

Горизонтальная табуляция (ASCII 9), перевод строки (ASCII 10), вертикальная табуляция (ASCII 11), разрыв страницы (ASCII 12), возврат каретки (ASCII 13) и пробел (ASCII 32).

Включение в оператор SIMILAR TO предопределённого класса имеет тот же эффект, как и включение всех его элементов. Использование предопределённых классов допускается только в пределах определения класса. Если вам нужно сопоставление только с предопределённым 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 все, 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, соответствующим любому единственному символу.

Кванторы

Вопросительный знак (?) сразу после символа или класса указывает на то, что для соответствия предыдущий элемент должен встретиться 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 должен встретиться 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 должен встретиться 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:]+ujebe' -- FALSE
```

Если символ или класс сопровождаются числом, заключённым в фигурные скобки ('{' и '}'), то для соответствия необходимо повторение элемента точно 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 необходимо повторение элемента как минимум 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 n 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 '+' являются сокращением 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 ('(' и ')'):

```
'Orange' SIMILAR TO '0(га|гi|го)nge' -- TRUE
'Orange' SIMILAR TO '0(г[a-e])+nge' -- TRUE
'Orange' SIMILAR TO '0(га){2,4}nge' -- FALSE
'Orange' SIMILAR TO '0(г(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
'Rega [Pear]' SIMILAR TO 'P[^ ]+ #[\P[^ ]+\#]' ESCAPE '#' -- TRUE
'Paron-Appledryck' SIMILAR TO 'P%$-A%' ESCAPE '$' -- TRUE
'Parondryck' SIMILAR TO 'P%--A%' ESCAPE '-' -- FALSE
```

IS DISTINCT FROM

Доступно в

DSQL, PSQL.

Синтаксис

```
<operand1> IS [NOT] DISTINCT FROM <operand2>
```

Два операнда считают *DISTINCT* (различными), если они имеют различные значения, или если одно из них — NULL, а другое нет. Они считаются *NOT DISTINCT* (равными), если имеют одинаковые значения или оба имеют значение NULL.

IS [NOT] DISTINCT FROM всегда возвращает TRUE или FALSE и никогда UNKNOWN (NULL) (неизвестное значение). Операторы '=' и '<>', наоборот, вернут UNKNOWN (NULL), если один или оба операнда имеют значение NULL.

Таблица 21. Результаты выполнения различных операторов сравнения

Характеристики операнда	Результаты различных операторов			
	=	IS NOT DISTINCT FROM	<>	IS DISTINCT FROM
Однаковые значения	TRUE	TRUE	FALSE	FALSE
Различные значения	FALSE	FALSE	TRUE	TRUE
Оба NULL	UNKNOWN	TRUE	UNKNOWN	FALSE
Одно NULL	UNKNOWN	FALSE	UNKNOWN	TRUE

Пример 55. Использование предиката IS [NOT] DISTINCT FROM

```
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';
```

См. также:

Логический IS [NOT], IS [NOT] NULL.

Логический IS [NOT]

Доступно в
DSQL, PSQL.

Синтаксис

```
<value> IS [NOT] {TRUE | FALSE | UNKNOWN}
```

Оператор IS проверяет, что выражение в левой части соответствует логическому значению в правой части. Выражение в левой части должно быть логического типа, иначе будет выдана ошибка.

Для логического типа данных предикат IS [NOT] UNKNOWN эквивалентен IS [NOT] NULL.

Замечание:

 В правой части предиката могут быть использованы только литералы TRUE, FALSE, UNKNOWN, но не выражения.

Пример 56. Использование оператора IS с логическим типом данных

```
-- Проверка FALSE значения
```

```
SELECT * FROM TBOOL WHERE BVAL IS FALSE
```

ID	BVAL
2	<false>

```
-- Проверка UNKNOWN значения
```

```
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN
```

ID	BVAL
3	<null>

IS [NOT] NULL

Доступно в

DSQL, PSQL.

Синтаксис

```
<value> IS [NOT] NULL
```

Поскольку NULL не является значением, эти операторы не являются операторами сравнения. Оператор IS [NOT] NULL проверяет, что выражение слева имеет значение (IS NOT NULL) или не имеет значения (IS NULL)

Пример 57. Использование предиката IS [NOT] NULL

Поиск записей о продажах, для которых не установлена дата отгрузки:

```
SELECT *
FROM SALES
WHERE SHIP_DATE IS NULL;
```

4.3.4. Предикаты существования

В эту группу предикатов включены предикаты, которые используют подзапросы и передают значения для всех видов утверждений в условиях поиска. Предикаты

существования называются так потому, что они различными способами проверяют существование или отсутствие результатов подзапросов.

EXISTS

Доступно в

DSQL, PSQL, ESQL.

Синтаксис

```
[NOT] EXISTS (<select_stmt>)
```

Предикат EXISTS использует подзапрос в качестве аргумента. Если результат подзапроса будет содержать хотя бы одну запись, то предикат оценивается как истинный (TRUE), в противном случае предикат оценивается как ложный (FALSE).

Результат подзапроса может содержать несколько столбцов, поскольку значения не проверяются, а просто фиксируется факт наличия строк результата. Данный предикат может принимать только два значения: истина (TRUE) и ложь (FALSE).

Предикат NOT EXISTS возвращает FALSE, если результат подзапроса будет содержать хотя бы одну запись, в противном случае предикат вернёт TRUE.

Пример 58. Предикат EXISTS

Найти тех сотрудников, у которых есть проекты.

```
SELECT *
FROM employee
WHERE EXISTS (SELECT *
               FROM
                   employee_project ep
               WHERE
                   ep.emp_no = employee.emp_no)
```

Пример 59. Предикат NOT EXISTS

Найти тех сотрудников, у которых нет проектов.

```
SELECT *
FROM employee
WHERE NOT EXISTS (SELECT *
                     FROM
                         employee_project ep
                     WHERE
                         ep.emp_no = employee.emp_no)
```

IN

Доступно в

DSQL, PSQL, ESQL.

Синтаксис

```
<value> [NOT] IN (<select_stmt> | <value_list>)
```

```
<value_list> ::= <value_1> [, <value_2> ...]
```

Предикат IN проверяет, присутствует ли значение выражения слева в указанном справа наборе значений. Набор значений не может превышать 1500 элементов. Предикат IN может быть переписан в следующей эквивалентной форме:

```
(<value> = <value_1> [OR <value> = <value_2> ...])
```

При использовании предиката IN в поисковых условиях DML запросов, оптимизатор Firebird может использовать индекс по искомому столбцу, если он определён.

Во второй форме предикат IN проверяет, присутствует (или отсутствует, при использовании NOT IN) ли значение выражения слева в результате выполнения подзапроса справа. Результат подзапроса может содержать только один столбец, иначе будет выдана ошибка “count of column list and variable list do not match”.

Запросы с использованием предиката IN с подзапросом, можно переписать на аналогичный запрос с использованием предиката EXISTS. Например, следующий запрос:

```
SELECT
    model, speed, hd
FROM PC
WHERE
    model IN (SELECT model
               FROM product
               WHERE maker = 'A');
```

Можно переписать на аналогичный запрос с использованием предиката EXISTS:

```
SELECT
    model, speed, hd
FROM PC
WHERE
    EXISTS (SELECT *
               FROM product
               WHERE maker = 'A'
               AND product.model = PC.model);
```

Однако, запрос с использованием NOT IN не всегда даст тот же результат, что запрос NOT EXISTS. Причина заключается в том, что предикат EXISTS всегда возвращает TRUE или FALSE, тогда как предикат IN может вернуть NULL в следующих случаях:

- Когда проверяемое значение равно NULL и список в IN не пуст.
- Когда проверяемое значение не имеет совпадений в списке IN и одно из значений является NULL.

В этих двух случаях предикат IN вернёт NULL, в то время как соответствующий предикат EXISTS вернёт FALSE. В поисковых условиях или операторе IF оба результата обозначают “провал” и обрабатываются одинаково.

Однако на тех же данных NOT IN вернёт NULL, в то время как EXISTS вернёт TRUE, что приведёт к противоположному результату.

Это можно продемонстрировать следующим примером.

Предположим у вас есть такой запрос:

```
-- Ищем людей, которые не родились в тот же день, что
-- известные жители Нью-Йорка
SELECT P1.name AS NAME
FROM Personnel P1
WHERE P1.birthday NOT IN (SELECT C1.birthday
                           FROM Celebrities C1
                           WHERE C1.birthcity = 'New York');
```

Можно предположить, что аналогичный результат даст запрос с использованием предиката NOT EXISTS:

```
-- Ищем людей, которые не родились в тот же день, что
-- известные жители Нью-Йорка
SELECT P1.name AS NAME
FROM Personnel P1
WHERE NOT EXISTS (SELECT *
                   FROM Celebrities C1
                   WHERE C1.birthcity = 'New York'
                   AND C1.birthday = P1.birthday);
```

Допустим, что в Нью-Йорке всего один известный житель, и его дата рождения неизвестна. При использовании предиката EXISTS подзапрос внутри него не выдаст результатов, так как при сравнении дат рождения с NULL результатом будет UNKNOWN. Это приведёт к тому, что результат предиката NOT EXISTS будет истинен для каждой строки основного запроса. В то время как результатом предиката NOT IN будет UNKNOWN и ни одна строка не будет выведена.

Пример 60. Предикат IN

Найти сотрудников с именами “Pete”, “Ann” и “Roger”:

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME IN ('Pete', 'Ann', 'Roger');
```

Пример 61. Поисковый предикат IN

Найти все компьютеры, для которых существуют модели с производителем начинающимися на букву “A”:

```
SELECT
    model, speed, hd
FROM PC
WHERE
    model IN (SELECT model
        FROM product
        WHERE maker STARTING WITH 'A');
```

См. также:

EXISTS.

SINGULAR

Доступно в

DSQL, PSQL, ESQL.

Синтаксис

```
[NOT] SINGULAR (<select_stmt>)
```

Предикат SINGULAR использует подзапрос в качестве аргумента и оценивает его как истинный, если подзапрос возвращает одну и только одну строку результата, в противном случае предикат оценивается как ложный. Результат подзапроса может содержать несколько столбцов, поскольку значения не проверяются. Данный предикат может принимать только два значения: истина (TRUE) и ложь (FALSE).

Пример 62. Предикат SINGULAR

Найти тех сотрудников, у которых есть только один проект.

```
SELECT *
FROM employee
WHERE SINGULAR (SELECT *
                  FROM
                    employee_project ep
                 WHERE
                   ep.emp_no = employee.emp_no)
```

4.3.5. Количественные предикаты подзапросов

Квантором называется логический оператор, задающий количество объектов, для которых данное утверждение истинно. Это логическое количество, а не числовое; оно связывает утверждение с полным множеством возможных объектов. Такие предикаты основаны на формальных логических квантификаторах общности и существования, которые распознаются формальной логикой.

В выражениях подзапросов количественные предикаты позволяют сравнивать отдельные значения с результатами подзапросов; их общая форма:

```
<value expression> <comparison operator> <quantifier> <subquery>
```

ALL

Доступно в

DSQL, PSQL.

Синтаксис

```
<value> <op> ALL (<select_stmt>)
```

При использовании квантора ALL, предикат является истинным, если каждое значение выбранное подзапросом удовлетворяет условию в предикате внешнего запроса. Если подзапрос не возвращает ни одной строки, то предикат автоматически считается верным.

Пример 63. Квантор ALL

Вывести только тех заказчиков, чьи оценки выше, чем у каждого заказчика в Париже.

```
SELECT *
FROM Customers
WHERE rating > ALL
  (SELECT rating
   FROM Customers
   WHERE city = 'Paris')
```

Если подзапрос возвращает пустое множество, то предикат будет истинен для каждого левостороннего значения, независимо от оператора. Это может показаться странным и противоречивым, потому что в этом случае каждое левостороннее значение рассматривается как одновременно больше, меньше, равное и неравное любому значению из правого потока.



Тем не менее это нормально согласуется с формальной логикой: если множество пусто, то предикат верен 0 раз, т.е. для каждой строки в множестве.

ANY и SOME

Доступно в
DSQL, PSQL.

Синтаксис

```
<value> <op> {ANY | SOME} (<select_stmt>)
```

Эти два квантора идентичны по поведению. Очевидно, оба представлены в стандарте SQL для взаимозаменяемого использования с целью улучшения читаемости операторов. При использовании квантора ANY или SOME, предикат является истинным, если любое из значений выбранное подзапросом удовлетворяет условию в предикате внешнего запроса. Если подзапрос не возвращает ни одной строки, то предикат автоматически считается ложным.

Пример 64. Квантор ANY

Вывести только тех заказчиков, чьи оценки выше, чем у какого-либо заказчика в Риме.

```
SELECT *
FROM Customers
WHERE rating > ANY
  (SELECT rating
   FROM Customers
   WHERE city = 'Rome')
```

Глава 5. Операторы определения данных (DDL)

Data Definition Language (DDL) — язык описания данных. С помощью этого подмножества языка создаются, модифицируются и удаляются объекты базы данных (т.е. Метаданные).

5.1. DATABASE

В данном разделе описываются вопросы создания базы данных, подключения к существующей базе данных, изменения структуры файлов, перевод базы данных в состояние, необходимое для безопасного резервного копирования, и обратно и удаления базы данных.

5.1.1. CREATE DATABASE

Назначение

Создание новой базы данных.

Доступно в

DSQL, ESQL

Синтаксис

```

CREATE {DATABASE | SCHEMA} <filespec>
[<db_initial_option> [<db_initial_option> ...]]
[<db_config_option> [<db_config_option> ...]]

<db_initial_option> ::= 
    USER username
  | PASSWORD 'password'
  | ROLE rolename
  | PAGE_SIZE [=] size
  | LENGTH [=] num [PAGE[S]]
  | SET NAMES 'charset'

<db_config_option> ::= 
    DEFAULT CHARACTER SET default_charset
      [COLLATION collation]
  | <sec_file>
  | DIFFERENCE FILE 'diff_file'

<filespec> ::= '<server_spec>{filepath | db_alias}'

<server_spec> ::= 
    host[/:{port | service}]:
  | \\host\
  | <protocol>://[host[:{port | service}]/]

<protocol> ::= inet | inet4 | inet6 | wnet | xnet

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

```



Каждый *db_initial_option* и *db_config_option* может встречаться не более одного раза, за исключением *sec_file*, который может встречаться ноль или более раз.

Таблица 22. Параметры оператора CREATE DATABASE

Параметр	Описание
filespec	Спецификация первичного файла базы данных.
server_spec	Спецификация удалённого сервера. Включает в себя имя сервера и протокол. Необходима, если база данных создаётся на удалённом сервере.

Параметр	Описание
filepath	Полный путь и имя файла, включая расширение. Имя файла должно быть задано в соответствии со спецификой используемой платформы.
db_alias	Псевдоним (alias) базы данных, присутствующий в файле <i>databases.conf</i>
host	Имя сервера или IP адрес, на котором создаётся база данных.
port	Номер порта, который слушает удалённый сервер (параметр <i>RemoteServicePort</i> файла <i>firebird.conf</i>).
service	Имя сервиса. Должно совпадать со значением параметра <i>RemoteServiceName</i> файла <i>firebird.conf</i> .
protocol	Наименование протокола.
username	Имя пользователя-владельца базы данных. Может быть заключено в одинарные или двойные кавычки. Если имя пользователя заключено в двойные кавычки, то оно чувствительно к регистру.
password	Пароль пользователя-владельца базы данных. Чувствительно к регистру.
role	Имя роли, права которой могут учитываться при создании базы данных. Может быть заключено в одинарные или двойные кавычки. Если имя роли заключено в двойные кавычки, то оно чувствительно к регистру.
size	Размер страницы для базы данных. Допустимые значения 4096, 8192, 16384, 32768. Размер страницы по умолчанию 8192.
num	Максимальный размер первичного или вторичного файла в страницах.
charset	Задаёт набор символов подключения, доступного после успешного создания базы данных.
default_charset	Задаёт набор символов по умолчанию для строковых типов данных.
collation	Сортировка для набора символов по умолчанию.
sec_file	Спецификация вторичного файла.
pagenum	Номер страницы, с которой начинается вторичный файл базы данных.
diff_file	Путь и имя дельта файла.

Оператор `CREATE DATABASE` создаёт новую базу данных. Вы можете использовать `CREATE DATABASE` или `CREATE SCHEMA`. Это синонимы.

База данных может состоять из одного или нескольких файлов. Первый, основной, файл называется первичным, остальные файлы — вторичными.

 В настоящее время многофайловые базы данных являются атавизмом. Многофайловые базы данных имеет смысл использовать на старых файловых системах, в которых существует ограничение на размер любого файла. Например, в FAT32 нельзя создать файл больше 4х гигабайт.

Спецификация первичного файла — имя файла базы данных и его расширение с указанием к нему полного пути в соответствии с правилами используемой операционной системы. При создании базы данных файл базы данных должен отсутствовать. В противном случае будет выдано сообщение об ошибке и база данных не будет создана. Если полный путь к базе данных не указан, то база данных будет создана в одном из системных каталогов. В каком именно зависит от операционной системы.

Использование псевдонимов БД

Вместо полного пути к первичному файлу базы можно использовать псевдонимы (aliases). Псевдонимы описываются в файле *databases.conf* в формате:

```
alias = filepath
```

 Помимо указания псевдонимов для базы данных в этом файле можно задать параметры уровня базы данных для каждой описываемой базы данных. Эти параметры задаются в фигурных скобках сразу после объявления псевдонима.

Создание БД на удалённом сервере

При создании базы данных на удалённом сервере необходимо указать спецификацию удалённого сервера. Спецификация удалённого сервера зависит от используемого протокола.

Если вы при создании базы данных используете протокол TCP/IP, то спецификация первичного файла должна выглядеть следующим образом:

```
host[{port | service}]:{filepath | db_alias}
```

Если вы при создании базы данных используете протокол под названием именованные каналы (Name Pipes), то спецификация первичного файла должна выглядеть следующим образом.

```
\\\host\{filepath | db_alias}
```

Существует также унифицированный URL-подобный синтаксис спецификации удалённого сервера. В этом синтаксисе первым параметром указывается наименование протокола, далее указывается имя сервера или IP адрес, номер порта и путь к первичному файлу базы данных или псевдоним. В качестве протокола можно указать следующие значения:

INET

TCP/IP (сначала пробует подключиться по протоколу TCP/IP v6, если не получилось, то TCP/IP v4);

INET4

TCP/IP v4;

INET6

TCP/IP v6;

WNET

протокол именованных каналов (Named Pipes);

XNET

локальный протокол.

```
<protocol>://[host[:{port | service}]/]{filepath | db_alias}
```

Необязательные параметры CREATE DATABASE**USER и PASSWORD**

Необязательные предложения USER и PASSWORD задают, соответственно, имя и пароль пользователя присутствующего в базе данных безопасности (*security4.fdb* или той, что указана в параметре *SecurityDatabase*). Пользователя и пароль можно не указывать, если установлены переменные окружения *ISC_USER* и *ISC_PASSWORD*. Пользователь, указанный при создании базы данных, будет её владельцем.

ROLE

Необязательное предложение ROLE задаёт имя роли (обычно это *RDB\$ADMIN*), права которой будут учитываться при создании базы данных. Роль должна быть назначена пользователю в соответствующей базе данных безопасности.

PAGE_SIZE

Необязательное предложение PAGE_SIZE задаёт размер страницы базы данных. Этот размер будет установлен для первичного файла и всех вторичных файлов базы данных. При вводе размера страницы БД меньшего, чем 4096, он будет автоматически изменён на 4096. Другие числа (не равные 4096, 8192, 16384 или 32768) будут изменены на ближайшее меньшее из поддерживаемых значений. Если размер страницы базы данных не указан, то по умолчанию принимается значение 8192.

Больше не значит лучше

Большие размеры страницы могут вместить больше записей на одной странице, иметь более широкие индексы и больше индексов, но они также будут тратить больше места для BLOB (сравните потраченное впустую пространство BLOB размером 3 КБ на странице размером 4096 и такого же BLOB на 32768: +/- 1 КБ против +/- 29 КБ). Кроме того, при большом размере страницы увеличивается конкуренция за одну и ту же страницу данных, поскольку на неё вмещается больше записей, который могли бы располагаться на разных страницах.

LENGTH

Необязательное предложение LENGTH задаёт максимальный размер первичного или вторичного файла базы данных в страницах. При создании базы данных её первичный или вторичный файл будут занимать минимально необходимое количество страниц для хранения системных данных, не зависимо от величины, установленной в предложении LENGTH. Для единственного или последнего (в многофайловой базе данных) файла значение LENGTH никак не влияет на его размер. Файл будет автоматически увеличивать свой размер по мере необходимости.

SET NAMES

Необязательное предложение SET NAMES задаёт набор символов подключения, доступного после успешного создания базы данных. По умолчанию используется набор символов NONE.

DEFAULT CHARACTER SET

Необязательное предложение DEFAULT CHARACTER SET задаёт набор символов по умолчанию для строковых типов данных. Наборы символов применяются для типов CHAR, VARCHAR и BLOB. По умолчанию используется набор символов NONE. Для набора символов по умолчанию можно также указать сортировку по умолчанию (COLLATION). В этом случае сортировка станет умалчивающей для набора символов по умолчанию (т.е. для всей БД за исключением случаев использования других наборов символов).

STARTING AT

Предложение STARTING AT задаёт номер страницы базы данных, с которой должен начинаться следующий файл базы данных. Когда предыдущий файл будет полностью заполнен данными в соответствии с заданным номером страницы, система начнёт помещать вновь добавляемые данные в следующий файл базы данных.

DIFFERENCE FILE

Необязательное предложение DIFFERENCE FILE 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. Полное описание данного параметра см. в [ALTER DATABASE](#).

Диалект базы данных

По умолчанию база данных создаётся в 3 диалекте. Для того чтобы база данных была

создана в нужном вам диалекте SQL, следует перед выполнением оператора создания базы данных задать нужный диалект, выполнив оператор SET SQL DIALECT.

Кто может создать базу данных?

Выполнить оператор CREATE DATABASE могут:

- Администраторы;
- Пользователи с привилегией CREATE DATABASE.

Примеры

Пример 65. Создание базы данных в операционной системе Windows

Создание базы данных в операционной системе Windows расположенной на диске D с размером страницы 8192. Владельцем базы данных будет пользователь wizard. База данных будет в 1 диалекте, и использовать набор символов по умолчанию WIN1251.

```
SET SQL DIALECT 1;
CREATE DATABASE 'D:\test.fdb'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET WIN1251;
```

Пример 66. Создание базы данных в операционной системе Linux

Создание базы данных в операционной системе Linux с размером страницы 4096. Владельцем базы данных будет пользователь wizard. База данных будет в 3 диалекте, и использовать набор символов по умолчанию UTF8 с умалчиваемой сортировкой UNICODE_CI_AI.

```
CREATE DATABASE '/home/firebird/test.fdb'
USER "wizard" PASSWORD 'player' ROLE 'RDB$ADMIN'
PAGE_SIZE = 4096
DEFAULT CHARACTER SET UTF8 COLLATION UNICODE_CI_AI;
```

В данном случае при создании базы данных будет учитываться регистр символов для имени пользователя, потому что оно указано в двойных кавычках.

Пример 67. Создание базы данных на удалённом сервере

Создание базы данных на удалённом сервере baseserver расположенному по пути, на который ссылается псевдоним test, описанный в файле *databases.conf*. Используется протокол TCP. Владельцем базы данных будет пользователь wizard.

```
CREATE DATABASE 'baseserver:test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

То же самое с использованием унифицированного URL-подобного синтаксиса задания спецификации удалённого сервера.

```
CREATE DATABASE 'inet://baseserver:3050/test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

или

```
CREATE DATABASE 'inet://baseserver:gds_db/test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

При использовании доменных имён может быть полезно указать какой именно из протоколов IP v4 или IP v6 вы хотите использовать.

```
CREATE DATABASE 'inet4://baseserver/test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

или

```
CREATE DATABASE 'inet6://baseserver/test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

Создание базы данных с указанием IP адреса (IPv4) вместо указания имени сервера.

```
CREATE DATABASE '127:0:0:1:test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

Создание базы данных с указанием IP адреса (IPv6) вместо указания имени сервера.

```
CREATE DATABASE '[:1]:test'
USER wizard PASSWORD 'player' ROLE RDB$ADMIN
DEFAULT CHARACTER SET UTF8;
```

Пример 68. Создание многофайловой базы данных

Создание базы данных в З диалекте с набором символов по умолчанию UTF8. Первичный файл будет содержать 10000 страниц с размером страницы 8192. Как только в процессе работы с базой данных первичный файл будет заполнен, СУБД будет помещать новые данные во вторичный файл *test.fdb2*. Аналогичные действия будут происходить и со вторым вторичным файлом. Размер последнего файла будет увеличиваться до тех пор, пока это позволяет используемая операционная система или пока не будет исчерпана память на внешнем носителе.

```
SET SQL DIALECT 3;
CREATE DATABASE 'baseserver:D:\test.fdb'
USER wizard PASSWORD 'player' ROLE 'RDB$ADMIN'
PAGE_SIZE = 8192
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
STARTING AT PAGE 10001
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

Пример 69. Создание многофайловой базы данных 2

Создание базы данных в З диалекте с набором символов по умолчанию UTF8. Первичный файл будет содержать 10000 страниц с размером страницы 8192. Как только в процессе работы с базой данных первичный файл будет заполнен, СУБД будет помещать новые данные во вторичный файл *test.fdb2*. Аналогичные действия будут происходить и со вторым вторичным файлом.

```
SET SQL DIALECT 3;
CREATE DATABASE 'baseserver:D:\test.fdb'
USER wizard PASSWORD 'player' ROLE 'RDB$ADMIN'
PAGE_SIZE = 8192
LENGTH 10000 PAGES
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

См. также:

[ALTER DATABASE, DROP DATABASE.](#)

5.1.2. ALTER DATABASE

Назначение

Изменение структуры файлов базы данных, переключение её в состояние “безопасное для копирования” или изменение некоторых свойств базы данных.

Доступно в

DSQL, ESQL

Синтаксис

```

ALTER {DATABASE | SCHEMA}
  {<add_sec_clause> [<add_sec_clause> ...]}
  | {ADD DIFFERENCE FILE 'diff_file' | DROP DIFFERENCE FILE}
  | {{BEGIN | END} BACKUP}
  | {SET DEFAULT CHARACTER SET charset}
  | {SET DEFAULT SQL SECURITY {DEFINER | INVOKER}}
  | {SET LINGER TO linger_duration | DROP LINGER}
  | {ENCRYPT WITH plugin_name [KEY key_name] | DECRYPT}
  | {ENABLE | DISABLE} PUBLICATION
  | INCLUDE {TABLE <table_list> | ALL} TO PUBLICATION
  | EXCLUDE {TABLE <table_list> | ALL} FROM PUBLICATION

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

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

<table_list> ::= tablename [, tablename ...]
```

Таблица 23. Параметры оператора ALTER DATABASE

Параметр	Описание
add_sec_clause	Инструкция для добавления вторичного файла базы данных.
sec_file	Спецификация вторичного файла.
filepath	Полный путь и имя дельта файла или вторичного файла базы данных.
pagenum	Номер страницы, с которой начинается вторичный файл базы данных.
num	Максимальный размер вторичного файла в страницах.
diff_file	Путь и имя дельта файла.
charset	Новый набор символов по умолчанию для базы данных.
linger_duration	Задержка в секундах.
plugin_name	Имя плагина шифрования.

Параметр	Описание
key_name	Имя ключа шифрования.
table_list	Список таблиц, которые необходим разрешить или запретить для публикации (репликации).
tablename	Имя таблицы.

Оператор ALTER DATABASE изменяет структуру файлов базы данных или переключает её в состояние “безопасное для копирования”.

Добавление вторичного файла

Предложение ADD FILE добавляет к базе данных вторичный файл. Для вторичного файла необходимо указывать полный путь к файлу и имя вторичного файла. Описание вторичного файла аналогично тому, что описано в операторе CREATE DATABASE.

Пример 70. Добавление вторичного файла в базу данных

Как только в предыдущем первичном или вторичных файлах будет заполнено 30000 страниц, СУБД будет помещать данные во вторичный файл *test4.fdb*.

```
ALTER DATABASE
ADD FILE 'D:\test.fdb4'
STARTING PAGE 30001;
```

Изменение пути и имени дельта файла

Предложение ADD DIFFERENCE FILE задаёт путь и имя дельта файла, в который будут записываться изменения, внесённые в базу данных после перевода её в режим “безопасного копирования” (“copy-safe”). Этот оператор в действительности не добавляет файла. Он просто переопределяет умалчивающиеся имя и путь файла дельты. Для изменения существующих установок необходимо сначала удалить ранее указанное описание файла дельты с помощью 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*.



При задании относительного пути или только имени файла дельты он будет создаваться в текущем каталоге сервера. Для операционных систем Windows это системный каталог.

Предложение DROP DIFFERENCE FILE удаляет описание (путь и имя) файла дельты, заданное ранее командой ADD DIFFERENCE FILE. На самом деле при выполнении этого оператора файл не удаляется. Он удаляет путь и имя файла дельты и при последующем переводе БД в режим “безопасного копирования” будут использованы значения по умолчанию (т.е. тот же путь и имя, что и у файла БД, но с расширением *.delta*).

Пример 71. Установка пути и имени файла дельты

```
ALTER DATABASE
ADD DIFFERENCE FILE 'D:\test.diff';
```

Пример 72. Удаление описание файла дельты

```
ALTER DATABASE
DROP DIFFERENCE FILE;
```

Перевод базы данных в режим “безопасного копирования”

Предложение BEGIN BACKUP предназначено для перевода базы данных в режим “безопасного копирования” (“copy-safe”). Этот оператор “замораживает” основной файл базы данных, что позволяет безопасно делать резервную копию 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*).



Оператор BEGIN BACKUP, несмотря на синтаксис, не начинает резервное копирование базы данных, а лишь создаёт условия для его осуществления.

Предложение END BACKUP предназначено для перевода базы данных из режима “безопасного копирования” (“copy-safe”) в режим нормального функционирования. Этот оператор объединяет файл дельты с основным файлом базы данных и восстанавливает нормальное состояние работы, таким образом, закрывая возможность создания безопасных резервных копий средствами файловой системы. (При этом безопасное резервное копирование с помощью утилиты *gbak* остаётся доступным).

Пример 73. Перевод базы данных в режим “безопасногокопирования”

```
ALTER DATABASE
BEGIN BACKUP;
```

Пример 74. Возвращение базы данных в режим нормального функционирования из режима “безопасного копирования”

```
ALTER DATABASE
END BACKUP;
```

Изменение набора символов по умолчанию

Предложение SET DEFAULT CHARACTER SET изменяет набор символов по умолчанию для базы данных. Это изменение не затрагивает существующие данные. Новый набор символов по умолчанию будет использоваться только в последующих DDL командах, кроме того для них будет использоваться сортировка по умолчанию для нового набора символов.

Пример 75. Изменение набора символов по умолчанию для базы данных

```
ALTER DATABASE SET DEFAULT CHARACTER SET WIN1251;
```

Изменение привилегий выполнения по умолчанию

Начиная с Firebird 4.0 появилась возможность указывать объектам метаданных с какими привилегиями они будут выполняться:зывающего или определяющего пользователя. Для этого используется предложение SQL SECURITY, которое можно указать для таблицы, триггера, процедуры, функции или пакета. Если выбрана опция INVOKER, то объект метаданных будет выполняться с привилегиямизывающего пользователя. Если выбрана опция DEFINER, то объект метаданных будет выполняться с привилегиями определяющего пользователя (владельца). Если при создании PSQL модуля или таблицы предложение SQL SECURITY не задано, то по умолчанию используется опция INVOKER.

Предложение SET DEFAULT SQL SECURITY изменяет привилегии выполнения с которым по умолчанию создаются PSQL модули (хранимые процедуры, функции и пакеты).

Пример 76. Изменение привилегий выполнения по умолчанию

После выполнения данного оператора PSQL модули по умолчанию будут создаваться с опцией SQL SECURITY DEFINER

```
ALTER DATABASE SET DEFAULT SQL SECURITY DEFINER;
```

LINGER

Предложение SET LINGER позволяет установить задержку закрытия базы данных. Этот механизм позволяет Firebird в режиме SuperServer, сохранять базу данных в открытом состоянии в течение некоторого времени после того как последнее соединение закрыто, т.е. иметь механизм задержки закрытия базы данных. Это может помочь улучшить производительность и уменьшить издержки в случаях, когда база данных часто открывается и закрывается, сохраняя при этом ресурсы “разогретыми” до следующего открытия.



Такой режим может быть полезен для Web приложений, в которых коннект к базе обычно “живёт” очень короткое время.

Предложение `DROP LINGER` удаляет задержку и возвращает базу данных к нормальному состоянию (без задержки). Эта команда эквивалентна установки задержки в 0.



Удаление `LINGER` не самое лучшее решение для временной необходимости его отключения для некоторых разовых действий, требующих принудительного завершения работы сервера. Утилита `gfix` теперь имеет переключатель `-NoLinger`, который сразу закроет указанную базу данных, после того как последнего соединения не стало, независимо от установок `LINGER` в базе данных. Установка `LINGER` будет сохранена и нормально отработает в следующий раз.

Кроме того, одноразовое переопределение доступно также через сервисы API, с использованием тега `isc_spb_prp_nolinger`, например (в такой строке):

```
fbsvcmgr host:service_mgr user sysdba password xxx
action_properties dbname employee prp_nolinger
```

Пример 77. Установка задержки в 30 секунд

```
ALTER DATABASE SET LINGER 30;
```

Пример 78. Удаление задержки

```
ALTER DATABASE DROP LINGER;
```

или

```
ALTER DATABASE SET LINGER 0;
```

Шифрование базы данных

Оператор `ALTER DATABASE` с предложением `ENCRYPT WITH` шифрует базу данных с помощью указанного плагина шифрования. Шифрование начинается сразу после этого оператора и будет выполняться в фоновом режиме. Нормальная работа с базами данных не нарушается во время шифрования.

Процесс шифрования может быть проконтролирован с помощью поля MON\$CRYPT_PAGE в псевдо-таблице MON\$DATABASE или просмотрен на странице заголовка базы данных с помощью gstat -e.

gstat -h также будет предоставлять ограниченную информацию о состоянии шифрования.



Например, следующий запрос

```
select MON$CRYPT_PAGE * 100 / MON$PAGES from MON$DATABASE
```

будет отображать процент завершения процесса шифрования.

Необязательное предложение KEY позволяет передать имя ключа для плагина шифрования. Что делать с этим именем ключа решает плагин.

Оператор ALTER DATABASE с предложением DECRYPT дешифрует базу данных.

Пример 79. Шифрование базы данных

```
ALTER DATABASE ENCRYPT WITH DbCrypt;
```

Пример 80. Дешифрование базы данных

```
ALTER DATABASE DECRYPT;
```

Управление репликаций

Оператор ALTER DATABASE с предложением ENABLE PUBLICATION включает репликацию базы данных.

```
ALTER DATABASE ENABLE PUBLICATION
```

Для отключения репликации базы данных выполните оператор

```
ALTER DATABASE DISABLE PUBLICATION
```

Изменения будут применены сразу после подтверждения транзакции.

При настройке репликации должен быть определен набор репликации (он же публикация). Он включает в себя таблицы, которые должны быть реплицированы. Это также делается с помощью команды DDL:

```
ALTER DATABASE INCLUDE {TABLE <table_list> | ALL} TO PUBLICATION
```

```
<table_list> ::= tablename [, tablename ...]
```

При использовании ключевого слова ALL в набор репликации будут включены все таблицы, включая те что будут созданы позднее. Команда будет выглядеть следующим образом:

```
ALTER DATABASE INCLUDE ALL TO PUBLICATION
```

Вы можете задать конкретный набор таблиц для репликации. Для этого после ключевого слова TABLE необходимо указать список таблиц через запятую. В следующем примере мы разрешаем репликацию для таблиц t1 и t2:

```
ALTER DATABASE INCLUDE TABLE t1, t2 TO PUBLICATION
```

Для исключения таблиц из набора репликации (публикации) используется следующий оператор:

```
ALTER DATABASE EXCLUDE {TABLE <table_list> | ALL} FROM PUBLICATION
```

```
<table_list> ::= tablename [, tablename ...]
```

При использовании ключевого слова ALL из набора репликации будут исключены все таблицы. Если ранее в публикацию были добавлены все таблицы с использованием ключевого слова ALL, то данный оператор отключит автоматическую публикацию для вновь создаваемых таблиц. Команда будет выглядеть следующим образом:

```
ALTER DATABASE EXCLUDE ALL FROM PUBLICATION
```

Вы можете задать конкретный набор таблиц для исключения из репликации. Для этого после ключевого слова TABLE необходимо указать список таблиц через запятую. В следующем примере мы исключаем таблицы t1 и t2 из набора репликации:

```
ALTER DATABASE EXCLUDE TABLE t1, t2 FROM PUBLICATION
```

Таблицы, включенные для репликации, могут быть дополнительно отфильтрованы с использованием двух параметров в файле конфигурации *replication.conf*: *include_filter* и *exclude_filter*. Это регулярные выражения, которые применяются к именам таблиц и определяют правила для включения таблиц в набор репликации или исключения их из набора репликации.

Кто может выполнить ALTER DATABASE?

Выполнить оператор ALTER DATABASE могут:

- Администраторы;
- Владелец базы данных;
- Пользователи с привилегией ALTER DATABASE.

См. также:

[CREATE DATABASE](#), [DROP DATABASE](#).

5.1.3. DROP DATABASE

Назначение

Удаление текущей базы данных.

Доступно в

DSQL, ESQL

Синтаксис

`DROP DATABASE`

Оператор DROP DATABASE удаляет текущую базу данных. Перед удалением базы данных, к ней необходимо присоединиться. Оператор удаляет первый, все вторичные файлы и все файлы теневых копий.

Кто может удалить базу данных?

Выполнить оператор DROP DATABASE могут:

- Администраторы;
- Владелец базы данных;
- Пользователи с привилегией DROP DATABASE.

Примеры

Пример 81. Удаление базы данных

Удаление базы данных, к которой подключен клиент.

`DROP DATABASE;`

См. также:

[CREATE DATABASE](#), [ALTER DATABASE](#).

5.2. SHADOW

Теневая копия (shadow — дословно тень) является точной страничной копией базы данных. После создания теневой копии все изменения, сделанные в базе данных, сразу же отражаются и в теневой копии. Если по каким либо причинам первичный файл базы данных станет недоступным, то СУБД переключится на теневую копию.

В данном разделе рассматриваются вопросы создания и удаления теневых копий.



Это относится только к текущим операциям с базой данных, но не к новым подключениям. В случае поломки исходной базы данных администратор БД должен восстановить изначальные файлы базы данных, в том числе и с помощью файлов теневых копий. Только после этого будет возможно подключение новых клиентов.

5.2.1. CREATE SHADOW

Назначение

Создание теневой копии.

Синтаксис

```
CREATE SHADOW sh_num [AUTO | MANUAL] [CONDITIONAL]
  'filepath' [LENGTH [=] num [PAGE[S]]]
  [<secondary_file>];

<secondary_file> ::= 
  FILE 'filepath'
  LENGTH [=] num [PAGE[S]] | STARTING [AT [PAGE]] pageum
```

Таблица 24. Параметры оператора CREATE SHADOW

Параметр	Описание
sh_num	Номер теневой копии – положительное число, идентифицирующее набор файлов теневой копии.
filepath	Имя файла и путь к нему в соответствии с требованиями ОС.
num	Максимальный размер теневой копии в страницах.
secondary_file	Спецификация вторичного файла.
page_num	Номер страницы, с которой должен начинаться вторичный файл копии.

Оператор CREATE SHADOW создаёт новую теневую копию. Теневая копия начинает дублировать базу данных сразу в момент создания этой копии.

Теневые копии, как и база данных, могут состоять из нескольких файлов. Количество и размер файлов теневых копий не связано с количеством и размером файлов базы данных.

Для файлов теневой копии размер страницы устанавливается равным размеру страницы базы данных и не может быть изменён.

Если по каким либо причинам файл базы данных становится недоступным, то система преобразует тень в копию базы данных и переключается на неё. Теневая копия становится недоступной. Что будет дальше зависит от выбранного режима.

Режимы AUTO и MANUAL

Когда теневая копия преобразуется в базу данных она становится недоступной. Теневая копия может также стать недоступной если будет удалён её файл, или закончится место на диске, где она расположена, или если этот диск повреждён.

- Если выбран режим **AUTO** (значение по умолчанию), то в случае, когда теневая копия становится недоступной, автоматически прекращается использование этой копии и из базы данных удаляются все ссылки на нее. Работа с базой данных продолжается обычным образом без осуществления копирования в данную теневую копию.

Если указано ключевое слово **CONDITIONAL**, то система будет пытаться создать новую теневую копию, чтобы заменить потерянную. Это не всегда возможно, тогда вам потребуется создать новую тень вручную.

- Если выбран режим **MANUAL**, то в случае, когда теневая копия становится недоступной, все попытки соединения с базой данных и обращения к ней будут вызывать сообщение об ошибке до тех пор, пока теневая копия не станет доступной или пока не будет удалена администратором БД с помощью оператора **DROP SHADOW**.

Необязательные параметры CREATE SHADOW

LENGTH

Необязательное предложение **LENGTH** задаёт максимальный размер первичного или вторичного файла теневой копии в страницах. Для единственного или последнего файла теневой копии значение **LENGTH** никак не влияет на его размер. Файл будет автоматически увеличивать свой размер по мере необходимости.

STARTING AT

Предложение **STARTING AT** задаёт номер страницы теневой копии, с которой должен начинаться следующий файл теневой копии. Когда предыдущий файл будет полностью заполнен данными в соответствии с заданным номером страницы, система начнёт помещать вновь добавляемые данные в следующий файл теневой копии.

Кто может создать теневую копию?

Выполнить оператор **CREATE SHADOW** могут:

- Администраторы;
- Владелец базы данных;
- Пользователи с привилегией **ALTER DATABASE**.

Примеры

Пример 82. Создание теневую копию базы данных с номером 1

```
CREATE SHADOW 1 'g:\data\test.shd';
```

Пример 83. Создание многофайловой теневой копии

```
CREATE SHADOW 2 'g:\data\test.sh1'
LENGTH 8000 PAGES
FILE 'g:\data\test.sh2';
```

См. также:

[CREATE DATABASE](#), [ALTER DATABASE](#), [DROP SHADOW](#).

5.2.2. DROP SHADOW

Назначение

Удаление теневой копии.

Доступно в

DSQL, ESQL

Синтаксис

```
DROP SHADOW sh_num
[ {PRESERVE | DELETE} FILE ]
```

Таблица 25. Параметры оператора DROP SHADOW

Параметр	Описание
sh_num	Номер теневой копии — положительное число, идентифицирующее набор файлов теневой копии.

Оператор `DROP SHADOW` удаляет указанную теневую копию из базы данных, с которой установлено текущее соединение. При удалении теневой копии прекращается процесс дублирования данных в эту копию. Если указана опция `DELETE FILE`, то будут также удалены и все связанные файлы с этой теневой копией. Если указана опция `PRESERVE FILE`, то файлы останутся не тронутыми. Это может быть полезно, если вы делаете резервную копию с теневого файла. Умолчательной является опция `DELETE FILE`.

Кто может удалить теневую копию?

Выполнить оператор `DROP SHADOW` могут:

- Администраторы;
- Владелец базы данных;
- Пользователи с привилегией ALTER DATABASE.

Примеры

Пример 84. Удаление теневой копии с номером 1

```
DROP SHADOW 1;
```

См. также:

[CREATE SHADOW](#).

5.3. DOMAIN

Домен (Domain) — один из объектов реляционной базы данных, при создании которого можно задать некоторые характеристики, а затем использовать ссылку на домен при определении столбцов таблиц, объявлении локальных переменных, входных и выходных аргументов в модулях PSQL.

В данном разделе рассматриваются синтаксис операторов создания, модификации и удаления доменов. Подробное описание доменов и их использования можно прочесть в главе [Пользовательские типы данных — домены](#).

5.3.1. CREATE DOMAIN

Назначение

Создание нового домена.

Доступно в

DSQL, ESQL.

Синтаксис

```
CREATE DOMAIN name [AS] <datatype>
[DEFAULT {<literal> | NULL | <context_var>}]
[NOT NULL] [CHECK (<dom_condition>)]
[COLLATE collation_name];

<datatype> ::= 
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB
```

`<array_datatype> ::= См. Синтаксис массивов`

```

<dom_condition> ::=
    <val> <operator> <val>
    | <val> [NOT] BETWEEN <val> AND <val>
    | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
    | <val> IS [NOT] NULL
    | <val> IS [NOT] DISTINCT <val>
    | <val> IS [NOT] {TRUE | FALSE | UNKNOWN}
    | <val> [NOT] CONTAINING <val>
    | <val> [NOT] STARTING [WITH] <val>
    | <val> [NOT] LIKE <val> [ESCAPE <val>]
    | <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
    | <val> <operator> {ALL | SOME | ANY} (<select_list>)
    | [NOT] EXISTS (<select_expr>)
    | [NOT] SINGULAR (<select_expr>)
    | (<dom_condition>)
    | NOT <dom_condition>
    | <dom_condition> OR <dom_condition>
    | <dom_condition> AND <dom_condition>
```

```

<operator> ::=
    <> | != | ^= | ~= | = | < | > | <= | >=
    | !< | ^< | ~< | !> | ^> | ~>
```

```

<val> ::= {
    VALUE
    | <literal>
    | <context_var>
    | <expression>
    | NULL
    | NEXT VALUE FOR genname
    | GEN_ID(genname, <val>)
    | CAST(<val> AS <cast_type>)
    | (<select_one>)
    | func(<val> [, <val> ...])
}
```

```

<cast_type> ::=
    <datatype>
    | [TYPE OF] domain
    | TYPE OF COLUMN rel.col
```

Таблица 26. Параметры оператора CREATE DOMAIN

Параметр	Описание
name	Имя домена. Может содержать до 63 символов.
datatype	Тип данных SQL.

Параметр	Описание
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом данных домена.
dom_condition	Условие домена.
collation_name	Порядок сортировки.
charset	Набор символов.
select_one	Оператор SELECT выбирающий один столбец и возвращающий только одну строку.
select_list	Оператор SELECT выбирающий один столбец и возвращающий ноль и более строк.
select_expr	Оператор SELECT выбирающий несколько столбцов и возвращающий ноль и более строк.
expression	Выражение.
genname	Имя последовательности (генератора).
func	Скалярная функция.

Оператор `CREATE DOMAIN` создаёт новый домен.

В качестве базового типа домена можно указать любой тип данных SQL.

Детали для конкретного типа

Массивы

- Если домен должен быть массивом, базовым типом может быть любой тип данных SQL, кроме BLOB и массива.
- Размеры массива указаны в квадратных скобках. (В синтаксисе эти скобки заключены в кавычки, чтобы отличать их от квадратных скобок, обозначающих необязательные элементы синтаксиса).
- Для каждого измерения массива указывается одно или два целых числа, которые определяют нижнюю и верхнюю границы диапазона индекса:
 - По умолчанию массивы начинаются с 1. Нижняя граница является неявной, и необходимо указать только верхнюю границу. Когда указано только одно число меньше 1, то это определяет диапазон `пим .. 1`, а число больше 1 определяет диапазон `1 .. пим`.
 - Когда указано два числа, разделенных двоеточием (':') и необязательный пробел, то если второе большее, чем первое, это явно определяет диапазон индексов. Одна или обе границы могут быть меньше нуля, если верхняя граница больше нижней.
- Если массив имеет несколько измерений, определения диапазонов для каждого измерения должны быть разделены запятыми и необязательными пробелами.
- Индексы проверяются только, если значение массива действительно существует.

- Это означает, что сообщения об ошибках относительно недопустимых индексов не будут возвращаться, если выбор конкретного элемента массива ничего не вернет или если поле массива имеет значение NULL.

Строковые типы

Для типов CHAR, VARCHAR и BLOB с подтипов text можно указать набор символов в предложении CHARACTER SET. Если набор символов не указан, то по умолчанию принимается тот набор символов, который был указан при создании базы данных.



Если же при создании базы данных не был указан набор символов, то при создании домена по умолчанию принимается набор символов NONE. В этом случае данные хранятся и извлекаются, так как они были поданы. В столбец, основанный на таком домене, можно загружать данные в любой кодировке, но невозможно загрузить эти данные в столбец с другой кодировкой. Транслитерация не выполняется между исходными и конечными кодировками, что может приводить к ошибкам.

Предложение DEFAULT

Необязательное предложение DEFAULT позволяет указать значение по умолчанию для домена. Это значение будет помещено в столбец таблицы, который ссылает на данный домен, при выполнении оператора INSERT, если значение не будет указано для этого столбца. Локальные переменные и аргументы PSQL модулей, которые ссылаются на этот домен, будут инициализированы значением по умолчанию. В качестве значения по умолчанию может быть литерал совместимый по типу, неизвестное значение NULL и контекстная переменная, тип которой совместим с типом домена.

Ограничение NOT NULL

Предложение NOT NULL запрещает столбцам и переменным, основанным на домене, присваивать значение NULL.

Ограничение CHECK

Необязательное предложение CHECK задаёт ограничение домена. Ограничение домена задаёт условия, которому должны удовлетворять значения столбцов таблицы или переменных, которые ссылаются на данный домен. Условие должно быть помещено в круглые скобки. Условие — это логическое выражение, называемое также предикат, которое может возвращать значения TRUE (истина), FALSE (ложь) и UNKNOWN (неизвестно). Условие считается выполненным, если предикат возвращает значение TRUE или UNKNOWN (эквивалент NULL). Если предикат возвращает FALSE, то значение не будет принято.

Ключевое слово VALUE

Ключевое слово VALUE в ограничении домена является заменителем столбца таблицы, который основан на данном домене, или переменной PSQL модуля. Оно содержит значение, присваиваемое переменной или столбцу таблицы. Ключевое слово VALUE может быть использовано в любом месте ограничения CHECK, но обычно его используют в левой части условия.

COLLATE

Необязательное предложение `COLLATE` позволяет задать порядок сортировки, если домен основан на одном из строковых типов данных (за исключением `BLOB`). Если порядок сортировки не указан, то по умолчанию принимается порядок сортировки умалчиваемый для указанного набора сортировки при создании домена.

Кто может создать домен?

Выполнить оператор `CREATE DOMAIN` могут:

- Администраторы
- Пользователи с привилегией `CREATE DOMAIN`.

Пользователь, создавший домен, становится его владельцем.

Примеры

Пример 85. Создание домена, который может принимать значения больше 1000.

```
CREATE DOMAIN CUSTNO AS
INTEGER DEFAULT 10000
CHECK (VALUE > 1000);
```

Пример 86. Создание домена, который может принимать значения 'Да' и 'Нет'.

```
CREATE DOMAIN D_BOOLEAN AS
CHAR(3) CHECK (VALUE IN ('Да', 'Нет'));
```

Пример 87. Создание домена с набором символов UTF8 и порядком сортировки UNICODE_CI_AI.

```
CREATE DOMAIN FIRSTNAME AS
VARCHAR(30) CHARACTER SET UTF8
COLLATE UNICODE_CI_AI;
```

Пример 88. Создание домена со значением по умолчанию.

```
CREATE DOMAIN D_DATE AS
DATE DEFAULT CURRENT_DATE
NOT NULL;
```

Пример 89. Создание домена, определённого как массив из 2 элементов.

Создание домена, определённого как массив из 2 элементов типа NUMERIC(18, 3), нумерация элементов начинается с 1.

```
CREATE DOMAIN D_POINT AS
NUMERIC(18, 3) [2];
```



Вы можете использовать домены определённые как массив только для определения столбцов таблиц. Вы не можете использовать такие домены для определения локальных переменных и аргументов PostgreSQL модулей.

См. также:

[ALTER DOMAIN, DROP DOMAIN.](#)

5.3.2. ALTER DOMAIN

Назначение

Изменение текущих характеристик домена или его переименование.

Доступно в

DSQL, ESQL.

Синтаксис

```
ALTER DOMAIN domain_name
[TO new_name]
[TYPE <datatype>]
[SET DEFAULT {<literal> | NULL | <context_var>} | DROP DEFAULT]
[SET | DROP] NOT NULL
[ADD [CONSTRAINT] CHECK (<dom_condition>)} | DROP CONSTRAINT]
```

<datatype> ::=
 <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. [Синтаксис скалярных типов данных](#)

<blob_datatype> ::= См. [Синтаксис типа данных BLOB](#)

<array_datatype> ::= См. [Синтаксис массивов](#)

<dom_condition> ::=
 <val> <operator> <val>
 | <val> [NOT] BETWEEN <val> AND <val>
 | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
 | <val> IS [NOT] NULL
 | <val> IS [NOT] DISTINCT <val>

```

| <val> IS [NOT] {TRUE | FALSE | UNKNOWN}
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
| <val> <operator> {ALL | SOME | ANY} (<select_list>)
| [NOT] EXISTS (<select_expr>)
| [NOT] SINGULAR (<select_expr>)
| (<dom_condition>)
| NOT <dom_condition>
| <dom_condition> OR <dom_condition>
| <dom_condition> AND <dom_condition>

```

```

<operator> ::=

  <> | != | ^= | ~= | = | < | > | <= | >=
  | !< | ^< | ~< | !> | ^> | ~>

```

```

<val> ::=

  VALUE
  | <literal>
  | <context_var>
  | <expression>
  | NULL
  | NEXT VALUE FOR genname
  | GEN_ID(genname, <val>)
  | CAST(<val> AS <cast_type>)
  | (<select_one>)
  | func(<val> [, <val> ...])

```

```

<cast_type> ::=

  <datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col

```

Таблица 27. Параметры оператора ALTER DOMAIN

Параметр	Описание
domain_name	Имя домена.
new_name	Новое имя домена. Может содержать до 63 символов.
datatype	Тип данных SQL.
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом данных домена.
dom_condition	Условие домена.
collation	Порядок сортировки.

Параметр	Описание
select_one	Оператор SELECT выбирающий один столбец и возвращающий только одну строку.
select_list	Оператор SELECT выбирающий один столбец и возвращающий ноль и более строк.
select_expr	Оператор SELECT выбирающий несколько столбцов и возвращающий ноль и более строк.
expression	Выражение.
genname	Имя последовательности (генератора).
func	Скалярная функция.

Оператор ALTER DOMAIN изменяет текущие характеристики домена, в том числе и его имя. В одном операторе ALTER DOMAIN можно выполнить любое количество изменений домена.

TO name

Предложение TO позволяет переименовать домен. Имя домена можно изменить, если не существует зависимостей от этого домена, т.е. столбцов таблиц, локальных переменных и аргументов процедур, ссылающихся на данный домен.

SET DEFAULT

Предложение SET DEFAULT позволяет установить новое значение по умолчанию. Если домен уже содержал значение по умолчанию, то установка нового значения по умолчанию не требует предварительного удаления старого.

DROP DEFAULT

Предложение DROP DEFAULT удаляет ранее установленное для домена значение по умолчанию. В этом случае значением по умолчанию становится значение NULL.

ADD CONSTRAINT CHECK

Предложение ADD [CONSTRAINT] CHECK добавляет условие ограничения домена. Если домен уже содержал ограничение CHECK, то его предварительно необходимо удалить с помощью предложения DROP CONSTRAINT.

TYPE

Предложение TYPE позволяет изменить тип домена на другой допустимый тип. Не допустимы любые изменения типа, которые могут привести к потере данных. Например, количество символов в новом типе для домена не может быть меньше, чем было установлено ранее.



Изменение типа не поддерживается для типа BLOB и массивов.

SET NOT NULL

Предложение SET NOT NULL устанавливает ограничение NOT NULL для домена. В этом случае для переменных и столбцах базирующихся на домене значение NULL не допускается.



Успешная установка ограничения NOT NULL для домена происходит только после полной проверки данных таблиц, столбцы которых базируются на домене. Это может занять довольно длительное время.



При изменении описания домена, существующий PSQL код, может стать некорректным. Информация о том, как это обнаружить, находится в приложении [Поле RDB\\$VALID_BLR](#).

DROP NOT NULL

Предложение DROP NOT NULL удаляет ограничение NOT NULL для домена.

Что не может изменить ALTER DOMAIN

- Если домен был объявлен как массив, то изменить ни его тип, ни размерность нельзя. Также нет возможности изменить любой другой тип на тип массив.
- Не существует способа изменить сортировку по умолчанию. В этом случае необходимо удалить домен и пересоздать его с новыми атрибутами.

Кто может изменить домен?

Выполнить оператор ALTER DOMAIN могут:

- Администраторы
- Владелец домена;
- Пользователи с привилегией ALTER ANY DOMAIN.

Примеры

Пример 90. Изменение значения по умолчанию для домена.

```
ALTER DOMAIN CUSTNO
INTEGER DEFAULT 2000;
```

Пример 91. Переименование домена.

```
ALTER DOMAIN D_BOOLEAN TO D_BOOL;
```

Пример 92. Удаление значения по умолчанию и добавления ограничения для домена.

```
ALTER DOMAIN D_DATE
DROP DEFAULT
ADD CONSTRAINT CHECK (VALUE >= date '01.01.2000');
```

Пример 93. Изменение ограничения домена.

```
ALTER DOMAIN D_DATE
DROP CONSTRAINT;

ALTER DOMAIN D_DATE
ADD CONSTRAINT CHECK
(VALUE BETWEEN date '01.01.1900' AND date '31.12.2100');
```

Пример 94. Изменение типа домена.

```
ALTER DOMAIN FIRSTNAME
TYPE VARCHAR(50) CHARACTER SET UTF8;
```

Пример 95. Добавление ограничения NOT NULL для домена.

```
ALTER DOMAIN FIRSTNAME SET NOT NULL;
```

См. также:

[CREATE DOMAIN](#), [DROP DOMAIN](#).

5.3.3. DROP DOMAIN

Назначение

Удаление существующего домена.

Доступно в

DSQL, ESQL.

Синтаксис

```
DROP DOMAIN domain_name
```

Таблица 28. Параметры оператора DROP DOMAIN

Параметр	Описание
domain_name	Имя домена.

Оператор `DROP DOMAIN` удаляет домен, существующий в базе данных. Невозможно удалить домен, на который ссылаются столбцы таблиц базы данных или если он был задействован в одном из PSQL модулей. Чтобы удалить такой домен, необходимо удалить из таблиц все столбцы, ссылающиеся на домен и удалить все ссылки на домен из PSQL модулей.

Кто может удалить домен?

Выполнить оператор `DROP DOMAIN` могут:

- Администраторы
- Владелец домена;
- Пользователи с привилегией `DROP ANY DOMAIN`.

Примеры

Пример 96. Удаление домена

```
DROP DOMAIN COUNTRYNAME;
```

См. также:

[CREATE DOMAIN](#), [ALTER DOMAIN](#).

5.4. TABLE

Firebird — это реляционная СУБД. Данные в таких базах хранятся в таблицах. Таблица — это плоская двухмерная структура, содержащая произвольное количество строк (row). Строки таблицы часто называют записями (record). Все строки таблицы имеют одинаковую структуру и состоят из столбцов (column). Столбцы таблицы часто называют полями (fields). Таблица должна иметь хотя бы один столбец. С каждым столбцом связан определённый тип данных SQL.

В данном разделе рассматриваются вопросы создания, модификации и удаления таблиц базы данных.

5.4.1. CREATE TABLE

Назначение

Создание новой таблицы.

Доступно в

DSQL, ESQL

Синтаксис

```

CREATE [GLOBAL TEMPORARY] TABLE tablename
[EXTERNAL [FILE] 'filespec']
(<col_def> [, <col_def> | <tconstraint> ...])
[ON COMMIT {DELETE | PRESERVE} ROWS]
[SQL SECURITY {DEFINER | INVOKER}]
[ENABLE | DISABLE] PUBLICATION

<col_def> ::= 
    <regular_col_def>
  | <computed_col_def>
  | <identity_col_def>

<regular_col_def> ::= 
    colname { <datatype> | domain_name }
    [DEFAULT {<literal> | NULL | <context_var>}]
    [NOT NULL]
    [<col_constraint>]
    [COLLATE collation_name]

<computed_col_def> ::= 
    colname [{ <datatype> | domain_name }]
    {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::= 
    colname [<datatype>]
    GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY [(<identity column options>)]
    [<col_constraint>]

<identity column options> ::= 
    <identity column option> [<identity column option>]

<identity column option> ::= 
    START WITH startvalue
  | INCREMENT [BY] incrementvalue

<datatype> ::= 
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов

<col_constraint> ::= [CONSTRAINT constr_name]
{   UNIQUE [<using_index>]
  | PRIMARY KEY [<using_index>]

```

```

| REFERENCES other_table [(other_col)]
  [ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
  [ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
  [<using_index>]
| CHECK (<check_condition>
}

<tconstraint> ::= [CONSTRAINT constr_name]
{   UNIQUE (<col_list>) [<using_index>]
| PRIMARY KEY (<col_list>) [<using_index>]
| FOREIGN KEY (<col_list>
  REFERENCES other_table [(<col_list>)]
  [ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
  [ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
  [<using_index>]
| CHECK (<check_condition>
}

<col_list> ::= colname [, colname ...]

<using_index> ::= USING [ASC[ENDING] | DESC[ENDING]] INDEX indexname

<check_condition> ::=
  <val> <operator> <val>
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> IS [NOT] DISTINCT <val>
| <val> IS [NOT] {TRUE | FALSE | UNKNOWN}
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
| <val> <operator> {ALL | SOME | ANY} (<select_list>)
| [NOT] EXISTS (<select_expr>)
| [NOT] SINGULAR (<select_expr>)
| (<check_condition>)
| NOT <check_condition>
| <check_condition> OR <check_condition>
| <check_condition> AND <check_condition>

<operator> ::=
  <> | != | ^= | ~= | = | < | > | <= | >=
| !< | ^< | ~< | !> | ^> | ~>

<val> ::=
  colname ['['<array_idx> [, <array_idx> ...]']']
| <literal>
| <context_var>
| <expression>

```

```

| NULL
| NEXT VALUE FOR genname
| GEN_ID(genname, <val>)
| CAST(<val> AS <cast_type>)
| (<select_one>)
| func(<val> [, <val> ...])

```

```

<cast_type> ::=

  <datatype>
  | [TYPE OF] domain_name
  | TYPE OF COLUMN rel.colname

```

Таблица 29. Параметры оператора CREATE TABLE

Параметр	Описание
tablename	Имя таблицы, может содержать до 63 символов.
filespec	Спецификация файла (только для внешних таблиц).
colname	Имя столбца таблицы, может содержать до 63 символов.
datatype	Тип данных SQL.
domain_name	Имя домена.
startvalue	Начальное значение столбца идентификации.
identityvalue	Приращение столбца идентификации. Не может быть равно 0.
col_constraint	Ограничение столбца.
tconstraint	Ограничение таблицы.
constr_name	Имя ограничения, может содержать до 63 символов.
other_table	Имя таблицы, на которую ссылается внешний ключ.
other_col	Столбец таблицы, на которую ссылается внешний ключ.
using_index	Позволяет задать имя автоматически создаваемого индекса для ограничения, и дополнительно определить, какой это будет индекс — по возрастанию (по умолчанию) или по убыванию.
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом данных столбца.
check_condition	Условие проверки ограничения. Выполняется, если оценивается как TRUE или NULL/UNKNOWN.
collation_name	Порядок сортировки. Необходимо указывать если вы хотите чтобы порядок сортировки для столбца отличался от порядка сортировки для набора символов по умолчанию этого столбца.
select_one	Оператор SELECT выбирающий один столбец и возвращающий только одну строку.

Параметр	Описание
select_list	Оператор SELECT выбирающий один столбец и возвращающий ноль и более строк.
select_expr	Оператор SELECT выбирающий несколько столбцов и возвращающий ноль и более строк.
expression	Выражение.
genname	Имя последовательности (генератора).
func	Скалярная функция.

Оператор CREATE TABLE создаёт новую таблицу. Имя таблицы должно быть уникальным среди имён всех таблиц, представлений (VIEWs) и хранимых процедур базы данных.

Таблица может содержать, по меньшей мере, один столбец и произвольное количество ограничений таблицы.

Имя столбца должно быть уникальным для создаваемой таблицы. Для столбца обязательно должен быть указан либо тип данных, либо имя домена, характеристики которого будут скопированы для столбца, либо должно быть указано, что столбец является вычисляемым.

В качестве типа столбца можно использовать любой тип данных SQL.

Символьные столбцы

Для типов CHAR, VARCHAR и BLOB с подтипов TEXT можно указать набор символов в предложении CHARACTER SET. Если набор символов не указан, то по умолчанию принимается тот набор символов, что был указан при создании базы данных. Если же при создании базы данных не был указан набор символов, то по умолчанию принимается набор символов NONE. В этом случае данные хранятся и извлекаются, так как они были поданы. В столбец можно загружать данные в любой кодировке, но невозможно загрузить эти данные в столбец с другой кодировкой. Транслитерация между исходными и конечными кодировками не выполняется, что может приводить к ошибкам.

Необязательное предложение COLLATE позволяет задать порядок сортировки для строковых типов данных (за исключением BLOB). Если порядок сортировки не указан, то по умолчанию принимается порядок сортировки по умолчанию для указанного набора сортировки.

Ограничение NOT NULL

По умолчанию столбец может принимать значение NULL.

Необязательное предложение NOT NULL указывает, что столбцу не может быть присвоено значение NULL.

Значение по умолчанию

Необязательное предложение DEFAULT позволяет указать значение по умолчанию для столбца таблицы. Это значение будет помещено в столбец таблицы при выполнении оператора INSERT, если значение не будет указано для этого столбца. В качестве значения по

умолчанию может быть литерал совместимый по типу, неизвестное значение NULL или контекстная переменная, тип которой совместим с типом столбца. Если значение по умолчанию явно не устанавливается, то подразумевается пустое значение, NULL. Использование выражений в значении по умолчанию недопустимо.

Столбцы основанные на домене

Для определения столбца, можно воспользоваться ранее описанным доменом. Если определение столбца основано на домене, оно может включать новое значение по умолчанию, дополнительные ограничения CHECK, предложение COLLATE, которые перекрывают значения указанные при определении домена. Определение такого столбца может включать дополнительные ограничения столбца, например NOT NULL, если домен его ещё не содержит.

Следует обратить внимание на то, что если в определении домена было указано NOT NULL, на уровне столбца невозможно определить допустимость использования в нем значения NULL. Если вы хотите чтобы на основе домена можно было определять столбцы допускающие псевдозначение NULL и не допускающее его, то хорошей практикой является создание домена допускающего NULL и указание ограничения NOT NULL у столбцов таблицы там где это необходимо.



Столбцы идентификации (автоинкремент)

Столбец идентификации представляет собой столбец, связанный с внутренним генератором последовательностей. Столбцы идентификации могут быть определены либо с помощью предложения GENERATED BY DEFAULT AS IDENTITY, либо предложения GENERATED ALWAYS AS IDENTITY.

Если столбец идентификации задан как GENERATED BY DEFAULT, то его значение будет увеличиваться и использовано как значение по умолчанию при каждой вставке, только в том случае, если значение этого столбца не задано явно.

Чтобы использовать сгенерированное по умолчанию значение, необходимо либо указать ключевое слово DEFAULT при вставке в столбец идентификации, или просто не упоминать столбец идентификации в списке столбцов для вставки. В противном случае будет использовано указанное вами значение.

Пример 97. Столбец определённый как GENERATED BY DEFAULT AS IDENTITY

```
CREATE TABLE greetings (
    id INT GENERATED BY DEFAULT AS IDENTITY,
    name CHAR(50));

-- specify value "1":
INSERT INTO greetings VALUES (1, 'hi');

-- use generated default
INSERT INTO greetings VALUES (DEFAULT, 'salut');

-- use generated default
INSERT INTO greetings(ch) VALUES ('bonjour');
```



Это поведение может быть изменено в операторе INSERT если указана директива `OVERRIDING USER VALUE`. Подробнее см. [Директива OVERRIDING](#).

Если столбец идентификации задан как `GENERATED ALWAYS`, то его значение будет увеличиваться при каждой вставке. При попытке явно присвоить значение столбца идентификации в операторе `INSERT`, будет выдано сообщение об ошибке. В операторе `INSERT` вы можете указать ключевое слово `DEFAULT` вместо значения для столбца идентификации.

```
create table greetings (
    id INT GENERATED ALWAYS AS IDENTITY,
    name CHAR(50));

INSERT INTO greetings VALUES (DEFAULT, 'hello');

INSERT INTO greetings(ch) VALUES ('bonjour');
```



Это поведение может быть изменено в операторе `INSERT` если указана директива `OVERRIDING SYSTEM VALUE`. Подробнее см. [Директива OVERRIDING](#).

Необязательное предложение `START WITH` позволяет указать начальное значение отличное от нуля. Предложение `INCREMENT [BY]` устанавливает значение приращения. Значение приращения должно быть отлично от 0. По умолчанию значение приращения равно 1.

Правила

- Тип данных столбца идентификации должен быть целым числом с нулевым масштабом. Допустимыми типами являются `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC(x,0)` и `DECIMAL(x,0)`;
- Идентификационный столбец не может иметь `DEFAULT` и `COMPUTED` значений.

- Идентификационный столбец может быть изменён, чтобы стать обычным столбцом. Обычный столбец не может быть изменён, чтобы стать идентификационным.
- Идентификационные столбцы неявно являются NOT NULL столбцами.
- Уникальность не обеспечивается автоматически. Ограничения UNIQUE или PRIMARY KEY требуются для гарантии уникальности.

См. также:

[Директива OVERRIDING](#).

Вычисляемые поля

Вычисляемые поля могут быть определены с помощью предложения COMPUTED [BY] или GENERATED ALWAYS AS (согласно стандарту SQL-2003). Они эквивалентны по смыслу. Для вычисляемых полей не требуется описывать тип данных (но допустимо), СУБД вычисляет подходящий тип в результате анализа выражения. В выражении требуется указать корректную операцию для типов данных столбцов, входящих в его состав. При явном указании типа столбца для вычисляемого поля результат вычисления приводится к указанному типу, то есть, например, результат числового выражения можно вывести как строку. Вычисление выражения происходит для каждой строки выбранных данных, если в операторе выборки данных SELECT, присутствует такой столбец.



Вместо использования вычисляемого столбца в ряде случаев имеет смысл использовать обычный столбец, значение которого рассчитывается в триггерах на добавление и обновление данных. Это может снизить производительность вставки/модификации записей, но повысит производительность выборки данных.

Столбцы типа массив

Для любого типа данных кроме BLOB можно указать размерность массива, если столбец должен быть массивом. Размерность массива указывается в квадратных скобках. Чтобы не перепутать их с символами, означающими необязательные элементы, они выделены жирным шрифтом. При указании размерности массива указываются два числа через двоеточие. Первое число означает начальный номер элемента массива, второе — конечный. Если указано только одно число, то оно означает последний номер в элементе массива, а первым номером считается 1. Для многомерного массива размерности массива перечисляются через запятую.

Ограничения

Существуют четыре вида ограничений:

- первичный ключ (PRIMARY KEY);
- уникальный ключ (UNIQUE);
- внешний ключ (REFERENCES или FOREIGN KEY);

- проверочное ограничение (CHECK).

Ограничения могут быть указаны на уровне столбца (“ограничения столбцов”) или на уровне таблицы (“табличные ограничения”). Ограничения уровня таблицы необходимы, когда ключи (ограничение уникальности, первичный ключ или внешний ключ) должны быть сформированы по нескольким столбцам, или, когда ограничение CHECK включает несколько столбцов, т.е. действует на уровне записи. Синтаксис для некоторых типов ограничений может незначительно отличаться в зависимости от того определяется ограничение на уровне столбца или на уровне таблицы.

- Ограничение на уровне столбца указывается после определения других характеристик столбца. Оно может включать только столбец указанный в этом определении.
- Ограничения на уровне таблицы указываются после определений всех столбцов. Ограничения таблицы являются более универсальным способом записи ограничений, поскольку позволяют ограничение более чем для одного столбца таблицы.
- Вы можете смешивать ограничения столбцов и ограничения таблиц в одном операторе CREATE TABLE.

Системой автоматически создаётся индекс для первичного ключа (PRIMARY KEY), уникального ключа (UNIQUE KEY) и внешнего ключа (REFERENCES для ограничения уровня столбца, и FOREIGN KEY REFERENCES для ограничения уровня таблицы).

Имена для ограничений и их индексов

Если имя ограничения не задано, то оно автоматически будет генерировано системой.

Ограничения уровня столбца и их индексы автоматически именуются следующим образом:

- Имена ограничений имеют следующий вид INTG_[*replaceable*]n` , где [*replaceable*]n` представлено одним или несколькими числами;
- Имена индексов имеют вид RDB\$PRIMARYn (для индекса первичного ключа), RDB\$FOREIGNn (для индекса внешнего ключа) или RDB\$n (для индекса уникального ключа), где n представлено одним или несколькими числами;

Схемы автоматического формирования имён для ограничений уровня таблицы и их индексов одинаковы.

Именованные ограничения

Имя ограничения можно задать явно, если указать его в необязательном предложении CONSTRAINT. По умолчанию имя индекса ограничения будет тем же самым, что и самого ограничения. Если для индекса необходимо задать другое имя, то его можно указать в предложении USING.

Предложение USING

Предложение USING позволяет задать определённое пользователем имя автоматически создаваемого индекса для ограничения, и опционально определить, какой это будет индекс — по возрастанию (по умолчанию) или по убыванию.

Первичный ключ (PRIMARY KEY)

Ограничение первичного ключа PRIMARY KEY строится на поле с заданным ограничением NOT NULL и требует уникальности значений столбца. Таблица может иметь только один первичный ключ.

- Первичный ключ по единственному столбцу может быть определён как на уровне столбца, так и на уровне таблицы.
- Первичный ключ по нескольким столбцам может быть определён только на уровне таблицы.

Ограничение уникальности (UNIQUE)

Ограничение уникального ключа UNIQUE задаёт для значений столбца требование уникальности содержимого. Таблица может содержать любое количество уникальных ключей.

Как и первичный ключ, ограничение уникальности может быть определено на нескольких столбцах. В этом случае вы должны определять его как ограничение уровня таблицы.

NULL в уникальных ключах

Согласно стандарту SQL-99 Firebird допускает одно или более значений NULL в столбце на который наложено ограничение UNIQUE. Это позволяет определить ограничение UNIQUE на столбцах, которые не имеют ограничения NOT NULL.

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

- Разрешено множество записей со значением NULL во всех столбцах ключа;
- Разрешено множество записей с различными комбинациями null и not-null значений в ключах;
- Разрешено множество записей, в которых в одном из столбцов уникального ключа содержится значение NULL, а остальные столбцы заполнены значениями и эти значения различны хотя бы в одном из них;
- Разрешено множество записей, в которых в одном из столбцов уникального ключа содержится значение NULL, а остальные столбцы заполнены значениями, и эти значения имеют совпадения хотя бы в одном из них.

Это можно резюмировать следующим примером:

```
RECREATE TABLE t( x int, y int, z int, unique(x,y,z));
INSERT INTO t values( NULL, 1, 1 );
INSERT INTO t values( NULL, NULL, 1 );
INSERT INTO t values( NULL, NULL, NULL );
INSERT INTO t values( NULL, NULL, NULL ); -- Разрешено
INSERT INTO t values( NULL, NULL, 1 ); -- Запрещено
```

Внешний ключ (FOREIGN KEY)

Ограничение внешнего ключа гарантирует, что столбец (столбцы) участник может содержать только те значения, которые существуют в указанном столбце (столбцах) главной таблицы. Эти ссылочные столбцы часто называют столбцами назначения. Они должны быть первичным ключом или уникальным ключом в целевой таблице. Они могут не иметь ограничения NOT NULL, если они входят в ограничение уникального ключа.

Столбцы внешнего ключа не требуют ограничения NOT NULL.

На уровне столбца ограничение внешнего ключа определяется с использованием ключевого слова REFERENCES.

```
...  
ARTIFACT_ID INTEGER REFERENCES COLLECTION (ARTIFACT_ID),
```

В этом примере столбец ARTIFACT_ID ссылается на столбец с тем же именем в таблице COLLECTION.

На уровне таблицы могут быть определены внешний ключ над одним или несколькими столбцами. Внешние ключи над несколькими столбцами можно определить только на уровне таблицы.

Синтаксис определения внешнего ключа на уровне таблицы несколько отличается. После определения всех столбцов, с их ограничения уровня столбца, вы можете определить именованное ограничение внешнего ключа уровня таблицы, используя ключевые слова FOREIGN KEY и имён столбцов для которых оно применяется:

```
...  
CONSTRAINT FK_ARTSOURCE FOREIGN KEY(DEALER_ID, COUNTRY)  
REFERENCES DEALER (DEALER_ID, COUNTRY),
```

Обратите внимание на то, что имена столбцов в целевой (master) таблице могут отличаться от тех что указаны во внешнем ключе.



Если целевые столбцы не указаны, то внешний ключ автоматически ссылается на столбцы первичного ключа целевой таблицы.

Действия внешнего ключа

Для обеспечения дополнительной целостности данных можно указать необязательные опции, которые обеспечат согласованность данных между родительскими и дочерними таблицами по заданным правилам:

- Предложение ON UPDATE определяет, что произойдёт с записями подчинённой таблицы при изменении значения первичного/уникального ключа в строке главной таблицы.
- Предложение ON DELETE определяет, что произойдёт с записями подчинённой таблицы

при удалении соответствующей строки главной таблицы.

Для обеспечения ссылочной целостности внешнего ключа, когда изменяется или удаляется значение связанного первичного или уникального ключа, могут быть выполнены следующие действия:

- NO ACTION (по умолчанию) — не будет выполнено никаких действий;
- CASCADE — при изменении или удалении значения первичного ключа над значением внешнего ключа будут произведены те же действия. При выполнении удаления строки в главной таблице в подчинённой таблице должны быть удалены все записи, имеющие те же значения внешнего ключа, что и значение первичного (уникального) ключа удалённой строки главной таблицы. При выполнении обновления записи главной таблицы в подчинённой таблице должны быть изменены все значения внешнего ключа, имеющие те же значения, что и значение первичного (уникального) ключа изменяемой строки главной таблицы;
- SET DEFAULT — значения внешнего ключа всех соответствующих строк в подчинённой таблице устанавливаются в значение по умолчанию, заданное в предложении DEFAULT для этого столбца;
- SET NULL — значения внешнего ключа всех соответствующих строк в подчинённой таблице устанавливаются в пустое значение NULL.

Пример 98. Внешний ключ с каскадным обновлением и установкой NULL при удалении

```
CONSTRAINT FK_ORDERS_CUST
FOREIGN KEY (CUSTOMER) REFERENCES CUSTOMERS (ID)
ON UPDATE CASCADE ON DELETE SET NULL
```

Ограничение CHECK

Ограничение CHECK задаёт условие, которому должны удовлетворять значения, помещаемые в данный столбец. Условие — это логическое выражение, называемое также предикатом, которое может возвращать значения TRUE (истина), FALSE (ложь) и UNKNOWN (неизвестно). Условие считается выполненным, если предикат возвращает значение TRUE или UNKNOWN (эквивалент NULL). Если предикат возвращает FALSE, то значение не будет принято. Это условие используется при добавлении в таблицу новой строки (оператор INSERT) и при изменении существующего значения столбца таблицы (оператор UPDATE), а также операторов, в которых может произойти одно из этих действий (UPDATE OR INSERT, MERGE).

 При использовании предложения CHECK для столбца, базирующегося на домене, следует помнить, что выражение в CHECK лишь дополняет условие проверки, которое может уже быть определено в домене.

На уровне столбца или таблицы выражение в предложении CHECK ссылается на входящее значение с помощью идентификаторов столбцов, в отличие от доменов, где в ограничении CHECK для этих целей используется ключевое слово VALUE.

Пример 99. CHECK ограничения уровня столбца и уровня таблицы

```
CREATE TABLE PLACES (
    ...
    LAT DECIMAL(9, 6) CHECK (ABS(LAT) <= 90),
    LON DECIMAL(9, 6) CHECK (ABS(LON) <= 180),
    ...
    CONSTRAINT CHK_POLES CHECK (ABS(LAT) < 90 OR LON = 0)
);
```

Привилегии выполнения

Необязательное предложение SQL SECURITY в спецификации таблицы позволяет задать с какими привилегиями вычисляются вычисляемые столбцы. Если выбрана опция INVOKER, то вычисляемые столбцы вычисляются с привилегиями вызывающего пользователя. Если выбрана опция DEFINER, то вычисляемые столбцы вычисляются с привилегиями определяющего пользователя (владельца). По умолчанию вычисляемые столбцы вычисляются с привилегиями вызывающего пользователя. Кроме триггеры наследуют привилегии выполнения таблицы, если они не переопределены у самих триггеров.

Привилегии выполнения по умолчанию для вновь создаваемых объектов метаданных можно изменить с помощью оператора



```
ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

Управление репликаций

Необязательное предложение ENABLE PUBLICATION включает таблицу в набор репликации (публикацию). Если ранее был выполнен оператор ALTER DATABASE ADD ALL TO PUBLICATION, то таблица будет включена в публикацию даже если предложение ENABLE PUBLICATION не указано.

Необязательное предложение DISABLE PUBLICATION исключает таблицу из набора репликации (публикации). Это предложение имеет смысл указывать только если ранее был выполнен оператор ALTER DATABASE ADD ALL TO PUBLICATION, который автоматически добавляет вновь созданные таблицы в публикацию.

Кто может создать таблицу?

Выполнить оператор CREATE TABLE могут:

- Администраторы
- Пользователи с привилегией CREATE TABLE.

Пользователь, создавший таблицу, становится её владельцем.

Примеры

Пример 100. Создание таблицы

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

Пример 101. Создание таблицы с заданием именованного первичного и уникального ключей

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

Пример 102. Создание таблицы с добавлением её в набор репликации

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

Пример 103. Таблица с полем массивом

```
CREATE TABLE JOB (
    JOB_CODE          JOBCODE NOT NULL,
    JOB_GRADE         JOBGRADE NOT NULL,
    JOB_COUNTRY       COUNTRYNAME,
    JOB_TITLE         VARCHAR(25) NOT NULL,
    MIN_SALARY        NUMERIC(18, 2) DEFAULT 0 NOT NULL,
    MAX_SALARY        NUMERIC(18, 2) NOT NULL,
    JOB_REQUIREMENT   BLOB SUB_TYPE 1,
    LANGUAGE_REQ      VARCHAR(15) [1:5],
    PRIMARY KEY (JOB_CODE, JOB_GRADE, JOB_COUNTRY),
    FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY)
        ON UPDATE CASCADE
        ON DELETE SET NULL,
    CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY)
);
```

Пример 104. Создание таблицы с ограничением первичного, внешнего и уникального ключа для которых заданы пользовательские имена индексов

```
CREATE TABLE PROJECT (
    PROJ_ID      PROJNO NOT NULL,
    PROJ_NAME    VARCHAR(20) NOT NULL UNIQUE
        USING DESC INDEX IDX_PROJNAME,
    PROJ_DESC     BLOB SUB_TYPE 1,
    TEAM_LEADER   EMPNO,
    PRODUCT       PRODTYPE,
    CONSTRAINT PK_PROJECT PRIMARY KEY (PROJ_ID)
        USING INDEX IDX_PROJ_ID,
    FOREIGN KEY (TEAM_LEADER) REFERENCES EMPLOYEE (EMP_NO)
        USING INDEX IDX_LEADER
);
```

Пример 105. Создание таблицы со столбцом идентификации BY DEFAULT

```
CREATE TABLE objects (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name VARCHAR(15)
);

INSERT INTO objects (name) VALUES ('Table');
INSERT INTO objects (name) VALUES ('Book');
INSERT INTO objects (id, name) VALUES (10, 'Computer');

SELECT * FROM objects;
```

ID	NAME
1	Table
2	Book
10	Computer

Пример 106. Создание таблицы со столбцом идентификации ALWAYS

```

CREATE TABLE objects (
    id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name VARCHAR(15)
);

INSERT INTO objects (name) VALUES ('Table');
INSERT INTO objects (name) VALUES ('Book');
INSERT INTO objects (id, name) VALUES (DEFAULT, 'Computer');

SELECT * FROM objects;

```

ID	NAME
1	Table
2	Book
3	Computer

Пример 107. Создание таблицы со столбцом идентификации с начальным значением равным 10 и приращением равным 2

```

CREATE TABLE objects (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 10 INCREMENT BY 2)
PRIMARY KEY,
    name VARCHAR(15)
);

INSERT INTO objects (name) VALUES ('Table');
INSERT INTO objects (name) VALUES ('Book');

```

ID	NAME
12	Table
14	Book

Пример 108. Создание таблицы с вычисляемыми полями

```
CREATE TABLE SALARY_HISTORY (
    EMP_NO      EMPNO NOT NULL,
    CHANGE_DATE TIMESTAMP DEFAULT 'NOW' NOT NULL,
    UPDATER_ID  VARCHAR(20) NOT NULL,
    OLD_SALARY   SALARY NOT NULL,
    PERCENT_CHANGE DOUBLE PRECISION DEFAULT 0 NOT NULL,
    SALARY_CHANGE GENERATED ALWAYS AS
        (OLD_SALARY * PERCENT_CHANGE / 100),
    NEW_SALARY   COMPUTED BY
        (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100)
);
```

Поле SALARY_CHANGE объявлено согласно стандарту SQL::2003, поле NEW_SALARY в классическом стиле объявления вычисляемых полей в Firebird.

```
CREATE TABLE SALARY_HISTORY
(
    EMP_NO      EMPNO NOT NULL,
    CHANGE_DATE TIMESTAMP DEFAULT 'NOW' NOT NULL,
    UPDATER_ID  VARCHAR(20) NOT NULL,
    OLD_SALARY   SALARY NOT NULL,
    PERCENT_CHANGE DOUBLE PRECISION DEFAULT 0 NOT NULL,
    SALARY_CHANGE GENERATED ALWAYS AS
        (OLD_SALARY * PERCENT_CHANGE / 100),
    NEW_SALARY   COMPUTED BY
        (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100)
)
SQL SECURITY DEFINER;
```

То же самое, но вычисляемые столбцы вычисляются с правами определяющего пользователя (владельца таблицы). Кроме триггеров наследуют привилегии выполнения таблицы, если они не переопределены у самих триггеров.

Глобальные временные таблицы (GTT)

Глобальные временные таблицы (в дальнейшем сокращённо “GTT”) так же, как и обычные таблицы, являются постоянными метаданными, но данные в них ограничены 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 завершении транзакции или отключении от БД. Метаданные GTT могут быть изменены или удалены с помощью инструкций ALTER TABLE и DROP TABLE.

Синтаксис

```
CREATE GLOBAL TEMPORARY TABLE name
  (<column_def> [, {<column_def> | <table_constraint>} ...])
  [ON COMMIT {DELETE | PRESERVE} ROWS]
  [SQL SECURITY {DEFINER | INVOKER}]
```

Если в операторе создания глобальной временной таблицы указано необязательное предложение ON COMMIT DELETE ROWS, то будет создана GTT транзакционного уровня (по умолчанию). При указании предложения ON COMMIT PRESERVE ROWS — будет создана GTT уровня соединения с базой данных.

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

 Операторы COMMIT RETAINING и ROLLBACK RETAINING сохраняют данные в глобальных временных таблицах объявленных как ON COMMIT DELETE ROWS. В Firebird 2.x была ошибка: COMMIT RETAINING и ROLLBACK RETAINING делали записи не видимыми для текущей транзакции. Для возврата поведения 2.x установить параметр ClearGTTAtRetaining равным 1 в *firebird.conf*. Этот параметр может быть удалён в Firebird 5.0.

Ограничения GTT

GTT обладают всеми атрибутами обычных таблиц (ключи, внешние ключи, индексы и триггеры), но имеют ряд 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”);
- Ограничения домена не могут ссылаться на любую GTT;
- Уничтожения экземпляра GTT в конце своего жизненного цикла не вызывает срабатывания триггеров до/после удаления.

В существующей базе данных не всегда легко отличить обычную таблицу от GTT, или GTT транзакционного уровня от GTT уровня соединения. Используйте следующий запрос для определения типа таблицы:

```
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'
```



Для просмотра информации о типах всех таблиц используйте запрос:

```
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
```

Поле RDB\$TYPE_NAME будет отображать PERSISTENT для обычной таблицы, VIEW для представления, GLOBAL_TEMPORARY_PRESERVE для GTT уровня соединения, и GLOBAL_TEMPORARY_DELETE для GTT уровня транзакции.

Примеры

Пример 109. Создание глобальной временной таблицы уровня соединения

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

Пример 110. Создание глобальной временной таблицы уровня транзакции ссылающейся внешним ключом на глобальную временную таблицу уровня соединения.

```
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);
```

Внешние таблицы

Необязательное предложение EXTERNAL [FILE] указывает, что таблица хранится вне базы данных во внешнем текстовом файле. Столбцы таблицы, хранящейся во внешнем файле, могут быть любого типа за исключением BLOB и массивов с любым типом данных.

Над таблицей, хранящейся во внешнем файле, допустимы только операции добавления новых строк (INSERT) и выборки (SELECT) данных. Операции же изменения существующих данных (UPDATE) или удаления строк такой таблицы (DELETE) не могут быть выполнены.

Внешняя таблица не может содержать ограничений первичного, внешнего и уникального ключа. Для полей такой таблицы невозможно создать индексы.

Файл с внешней таблицей должен располагаться на устройстве хранения, физически расположенному на сервере, на котором расположена СУБД. Если параметр ExternalFileAccess в файле конфигурации *firebird.conf* содержит *Restrict*, то файл внешней таблицы должен находится в одном из каталогов, указанных в качестве аргумента *Restrict*. Если при обращении к таблице Firebird не находит файла, то он создаёт его при первом обращении.

Возможность использования для таблиц внешних файлов зависит от установки значения параметра ExternalFileAccess в файле конфигурации *firebird.conf*:

- Если он установлен в значение *None*, то запрещён любой доступ к внешнему файлу.
- Значение *Restrict* рекомендуется для ограничения доступа к внешним файлам только каталогами, созданными специально для этой цели администратором сервера. Например:
 - ExternalFileAccess = *Restrict externalfiles* ограничит доступ директорией *externalfiles* корневого каталога Firebird.
 - ExternalFileAccess = *Restrict d:\databases\outfiles; e:\infiles* ограничит доступ только двумя директориями Windows. Обратите внимание, что любые пути являющиеся отображением сетевых путей не будут работать. Также не будут работать пути заключённые в одинарные или двойные кавычки.
- Значение *Full* позволяет доступ к внешним файлам в любом месте файловой системы хоста. Это создаёт уязвимость и не рекомендуется к использованию.

Формат внешних файлов

Внешняя таблица имеет формат “строк” с фиксированной длинной. Нет никаких разделителей полей: границы полей и строк определяются максимальными размерами в байтах в определении каждого поля. Это необходимо помнить и при определении структуры внешней таблицы, и при проектировании входного файла для внешней таблицы, в которую должны импортироваться данные из другого приложения. Например, широко распространённый формат “.csv”, не может быть использован в качестве входного файла, и не может быть получен непосредственно как внешний файл.

Самым полезным типом данных для столбцов внешних таблиц является тип CHAR с фиксированной длинной, длина должна подходить под данные с которыми необходимо работать. Числовые типы и даты легко преобразуются в них, строки получаются как есть, в то время как, если данные не читаются другой базой данных Firebird, то родные типы могут быть нераспознаваемыми для внешних приложений и являться для них “абракадаброй”.

Конечно, существуют способы манипулирования типами данных так, чтобы создавать выходные файлы из Firebird, которые могут быть непосредственно прочитаны как входные файлы в других приложениях, используя хранимые процедуры с использованием внешних таблиц или без них. Описания этих методов выходит за рамки данного руководства. Здесь мы приведём лишь некоторые рекомендации и советы для создания и работы с простыми текстовыми файлами, поскольку внешняя таблица часто используется как простой способ для создания или чтения транзакционно-независимого журнала. Эти файлы могут быть прочитаны в оффлайн режиме текстовым редактором или приложением аудита.

Разделитель строк

Как правило, внешние файлы более удобны если строки разделены разделителем, в виде последовательности "новой строки", которая может быть распознана приложением на предназначеннной платформе. Для Windows — это двухбайтная 'CRLF' последовательность, возврат каретки (ASCII код 13) и перевод строки (ASCII код 10). Для POSIX — LF обычно самодостаточен, в некоторых MacOS X приложениях она может быть LFCR. Существуют различные способы для автоматического заполнения столбца разделителя. В нашем примере это сделано с помощью BEFORE INSERT триггера и встроенной функции ASCII_CHAR.

Примеры использования внешних таблиц

В нашем примере мы будем определять внешнюю таблицу журнала, которая может быть использована в обработчике исключений внутри хранимой процедуры или триггера. Внешняя таблица выбрана потому, что сообщения из любых обрабатываемых исключений будут сохранены в журнале, даже если транзакция, в которой был запущен процесс, будет откачена из-за другого необработанного исключения. В целях демонстрации наша таблица содержит всего два столбца: метку времени и текстовое сообщение. Третий столбец хранит разделитель строки:

```
CREATE TABLE ext_log
EXTERNAL FILE 'd:\externals\log_me.txt' (
    stamp  CHAR(24),
    message CHAR(100),
    crlf    CHAR(2) -- Для Windows
);
COMMIT;
```

Теперь создадим триггер, для автоматического сохранения метки времени и разделителя строки, каждый раз когда сообщение записывается в таблицу:

```

SET TERM ^;
CREATE TRIGGER bi_ext_log FOR ext_log
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (NEW.stamp IS NULL) THEN
    NEW.stamp = CAST (CURRENT_TIMESTAMP AS CHAR(24));
  NEW.crlf = ASCII_CHAR(13) || ASCII_CHAR(10);
END ^
COMMIT ^
SET TERM ;^

```

Вставка некоторых записей (это может быть сделано в обработчике исключения):

```

INSERT INTO ext_log (message)
VALUES('Shall I compare thee to a summer''s day?');
INSERT INTO ext_log (message)
VALUES('Thou art more lovely and more temperate');

```

Содержимое внешнего файла:

```

2015-10-07 15:19:03.4110Shall I compare thee to a summer's day?
2015-10-07 15:19:58.7600Thou art more lovely and more temperate

```

См. также:

[ALTER TABLE](#), [DROP TABLE](#), [CREATE DOMAIN](#).

5.4.2. ALTER TABLE

Назначение

Изменение структуры таблицы.

Доступно в

DSQL, ESQL.

Синтаксис

```

ALTER TABLE tablename
<operation> [, <operation>];

<operation> ::= 
  ADD <col_def>
  | ADD <tconstraint>
  | DROP colname
  | DROP CONSTRAINT constr_name
  | DROP SQL SECURITY

```

```

| ALTER [COLUMN] colname <col_mod>
| ALTER SQL SECURITY {DEFINER | INVOKER}
| {ENABLE | DISABLE} PUBLICATION

<col_def> ::= 
    <regular_col_def>
| <computed_col_def>
| <identity_col_def>

<regular_col_def> ::= 
    colname { <datatype> | domainname }
    [DEFAULT {literal | NULL | <context_var>}]
    [NOT NULL]
    [<col_constraint>]
    [COLLATE collation_name]

<computed_col_def> ::= 
    colname [<datatype>]
    {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::= 
    colname [<datatype>] {ALWAYS | GENERATED BY} DEFAULT AS IDENTITY
    [(START WITH startvalue)] [<col_constraint>]

<col_mod> ::= 
    TO newname
| POSITION newpos
| <regular_col_mod>
| <computed_col_mod>
| <identity_col_mod>

<regular_col_mod> ::= 
    | TYPE { <datatype> | domain_name }
    | SET DEFAULT {literal | NULL | <context_var>}
    | DROP DEFAULT
    | SET NOT NULL
    | DROP NOT NULL

<computed_col_mod> ::= 
    [TYPE <datatype>] {GENERATED ALWAYS AS | COMPUTED [BY]} (<expression>)

<identity_col_mod> ::= 
    <alter identity column option> ...
| SET GENERATED { ALWAYS | BY DEFAULT } [<alter identity column option> ...]
| DROP INDENITY

<alter identity column option> ::= 
    RESTART [ WITH startvalue ]
| SET INCREMENT [BY] incrementvalue

```

```

<datatype> ::= 
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов

<col_constraint> ::= 
    [CONSTRAINT constr_name]
    {   UNIQUE [<using_index>]
        | PRIMARY KEY [<using_index>]
        | REFERENCES other_table [(other_col)]
            [ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
            [ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
            [<using_index>]
        | CHECK (<check_condition>)
    }

<tconstraint> ::= 
    [CONSTRAINT constr_name]
    {   UNIQUE (<col_list>) [<using_index>]
        | PRIMARY KEY (<col_list>) [<using_index>]
        | FOREIGN KEY (<col_list>)
            REFERENCES other_table [(<col_list>)]
            [ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
            [ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
            [<using_index>]
        | CHECK (<check_condition>)
    }

<col_list> ::= colname [, colname ...]

<using_index> ::= USING [ASC[ENDING] | DESC[ENDING]] INDEX indexname

<check_condition> ::= 
    <val> <operator> <val>
    | <val> [NOT] BETWEEN <val> AND <val>
    | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
    | <val> IS [NOT] NULL
    | <val> IS [NOT] DISTINCT <val>
    | <val> IS [NOT] {TRUE | FALSE | UNKNOWN}
    | <val> [NOT] CONTAINING <val>
    | <val> [NOT] STARTING [WITH] <val>
    | <val> [NOT] LIKE <val> [ESCAPE <val>]
    | <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
    | <val> <operator> {ALL | SOME | ANY} (<select_list>)
    | [NOT] EXISTS (<select_expr>)
    | [NOT] SINGULAR (<select_expr>)
    | <check_condition>

```

```

| NOT <check_condition>
| <check_condition> OR <check_condition>
| <check_condition> AND <check_condition>

<operator> ::= 
    <> | != | ^= | ~= | = | < | > | <= | >=
    | !< | ^< | ~< | !> | ^> | ~>

<val> ::= 
    colname [[<array_idx> [, <array_idx> ...]]]
    | literal
    | <context_var>
    | <expression>
    | NULL
    | NEXT VALUE FOR genname
    | GEN_ID(genname, <val>)
    | CAST(<val> AS <datatype>)
    | (<select_one>)
    | func(<val> [, <val> ...])

```



```

<cast_type> ::= 
    <datatype>
    | [TYPE OF] domain_name
    | TYPE OF COLUMN rel.colname

```

Таблица 30. Параметры оператора ALTER TABLE

Параметр	Описание
tablename	Имя таблицы.
operation	Одна из допустимых операций по изменению структуры таблицы.
colname	Имя столбца таблицы, может содержать до 63 символов. Должно быть уникальным внутри таблицы.
newname	Новое имя столбца таблицы, может содержать до 63 символов. Должно быть уникальным внутри таблицы.
gencolname	Имя вычисляемого столбца таблицы.
idencolname	Имя столбца идентификации.
newpos	Новая позиция столбца в таблице. Целое число в диапазоне от 1 до количества столбцов таблицы.
datatype	Тип данных SQL.
domain_name	Имя домена.
startvalue	Начальное значение столбца идентификации.
incrementvalue	Значение приращения для столбца идентификации. Должно быть отлично от 0.

Параметр	Описание
col_constraint	Ограничение столбца.
tconstraint	Ограничение таблицы.
constr_name	Имя ограничения, может содержать до 63 символов.
other_table	Имя таблицы, на которую ссылается внешний ключ.
other_col	Столбец таблицы, на которую ссылается внешний ключ.
using_index	Позволяет задать имя автоматически создаваемого индекса для ограничения, и дополнительно определить, какой это будет индекс — по возрастанию (по умолчанию) или по убыванию.
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом данных столбца.
check_condition	Условие проверки ограничения. Выполняется, если оценивается как TRUE или NULL/UNKNOWN.
collation_name	Имя порядка сортировки. Необходимо указывать если вы хотите чтобы порядок сортировки для столбца отличался от порядка сортировки для набора символов по умолчанию этого столбца.
select_one	Оператор SELECT выбирающий один столбец и возвращающий только одну строку.
select_list	Оператор SELECT выбирающий один столбец и возвращающий ноль и более строк.
select_expr	Оператор SELECT выбирающий несколько столбцов и возвращающий ноль и более строк.
expression	Выражение.
genname	Имя последовательности (генератора).
func	Скалярная функция.

Оператор ALTER TABLE изменяет структуру существующей таблицы. Одиночный оператор ALTER TABLE позволяет производить множество операций добавления/удаления столбцов и ограничений, а также модификаций столбцов. Список операций выполняемых при модификации таблицы разделяется запятой.

Счётчик форматов

Некоторые изменения структуры таблицы увеличивают счётчик форматов, закреплённый за каждой таблицей. Количество форматов для каждой таблицы ограничено значением 255. После того как счётчик форматов достигнет этого значения, вы не сможете больше менять структуру таблицы.

Сброс счётчика форматов

Для сброса счётчика форматов необходимо сделать резервное копирование и восстановление базы данных (утилитой `gbak`).

Предложение ADD

Предложение `ADD` позволяет добавить новый столбец или новое ограничение таблицы. Синтаксис определения столбца и синтаксис описания ограничения таблицы полностью совпадают с синтаксисом, описанным в операторе [CREATE TABLE](#).

Воздействие на счётчик форматов:

- При каждом добавлении нового столбца номер формата увеличивается на единицу.
- Добавление нового ограничения таблицы не влечёт за собой увеличение номера формата.

Пример 111. Добавление столбца в таблицу

```
ALTER TABLE COUNTRY
ADD CAPITAL VARCHAR(25);
```

Пример 112. Добавление столбца с ограничением NOT NULL

```
ALTER TABLE OBJECTS
ADD QUANTITY INT DEFAULT 1 NOT NULL;
```

Обратите внимание на предложение `DEFAULT`, которое обязательно при добавлении ограничения `NOT NULL`, если в таблице есть данные. Дело в том, что в этом случае также происходит проверка данных на допустимость. А поскольку при добавлении нового столбца, он для всех строк таблицы содержит значение `NULL`, будет генерировано исключение.



Пример 113. Добавление столбца с ограничением уникальности и удаление другого столбца

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

Для добавления ограничений уровня таблицы необходимо использовать предложение `ADD [CONSTRAINT]`.

Пример 114. Добавление проверочного ограничения и внешнего ключа

```
ALTER TABLE JOB
ADD CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY),
ADD FOREIGN KEY (JOB_COUNTRY)
REFERENCES COUNTRY (COUNTRY);
```



Будьте осторожны, при добавлении нового ограничения CHECK не осуществляется проверка соответствия ему ранее внесённых данных. Поэтому перед добавлением такого ограничения рекомендуем производить предварительную проверку данных в таблице.

Предложение DROP

Предложение **DROP** удаляет указанный столбец таблицы. Столбец таблицы не может быть удалён, если от него существуют зависимости. Другими словами для успешного удаления столбца на него должны отсутствовать ссылки. Ссылки на столбец могут содержаться:

- в ограничениях столбцов или таблицы;
- в индексах;
- в хранимых процедурах и триггерах;
- в представлениях.

При каждом удалении столбца номер формата увеличивается на единицу.

Предложение DROP CONSTRAINT

Предложение **DROP CONSTRAINT** удаляет указанное ограничение столбца или таблицы. Ограничение первичного ключа или уникального ключа не могут быть удалены, если они используются в ограничении внешнего ключа другой таблицы. В этом случае необходимо удалить ограничение FOREIGN KEY до удаления PRIMARY KEY или UNIQUE ключа, на которые оно ссылается.

Удаление ограничения столбца или ограничения таблицы не влечёт за собой увеличение номера формата.

Предложение DROP SQL SECURITY

Предложение **DROP SQL SECURITY** удаляет привилегии выполнения для таблицы. После удаления привилегий выполнения вычисляемые столбцы таблицы будут вычисляться с привилегиями вызывающего пользователя. Триггеры также будут выполняться с привилегиями вызывающего пользователя, если их привилегии выполнения не переопределены в триггере явно.

Предложение ALTER [COLUMN]

Предложение ALTER [COLUMN] позволяет изменить следующие характеристики существующих столбцов:

- изменение имени (не изменяет номер формата);
- изменение типа данных (увеличивает номер формата на единицу);
- изменение позиции столбца в списке столбцов таблицы (не изменяет номер формата);
- удаление значения по умолчанию столбца (не изменяет номер формата);
- добавление значения по умолчанию столбца (не изменяет номер формата);
- изменение типа и выражения для вычисляемого столбца (не изменяет номер формата);
- добавление ограничения NOT NULL (не изменяет номера формата);
- удаление ограничения NOT NULL (не изменяет номера формата).

Переименование столбца

Ключевое слово TO переименовывает существующий столбец. Новое имя столбца не должно присутствовать в таблице.

Невозможно изменение имени столбца, если этот столбец включён в какое-либо ограничение — первичный или уникальный ключ, внешний ключ, ограничение столбца или проверочное ограничение таблицы CHECK. Имя столбца также нельзя изменить, если этот столбец таблицы используется в каком-либо триггере, в хранимой процедуре или представлении.

Пример 115. Переименование столбца таблицы

```
ALTER TABLE STOCK
ALTER COLUMN MODELNAME TO NAME;
```

Изменение типа столбца

Ключевое слово TYPE изменяет тип существующего столбца на другой допустимый тип. Не допустимы любые изменения типа, которые могут привести к потере данных. Например, количество символов в новом типе для столбца не может быть меньше, чем было установлено ранее.

Если столбец был объявлен как массив, то изменить ни его тип, ни размерность нельзя.

Нельзя изменить тип данных у столбца, который принимает участие в связке внешний ключ / первичный (уникальный) ключ.

Пример 116. Изменение типа столбца таблицы

```
ALTER TABLE STOCK
ALTER COLUMN ITEMID TYPE BIGINT;
```

Изменение позиции столбца

Ключевое слово POSITION изменяет позицию существующего столбца. Позиции столбцов нумеруются с единицы.

- Если будет задан номер позиции меньше 1, то будет выдано соответствующее сообщение об ошибке.
- Если будет задан номер позиции, превышающий количество столбцов в таблице, то изменения не будут выполнены, но ни ошибки, ни предупреждения не последуют.

Пример 117. Изменение позиции столбца таблицы

```
ALTER TABLE STOCK
ALTER COLUMN ITEMID POSITION 5;
```

Установка и удаление значения по умолчанию

Предложение DROP DEFAULT удаляет значение по умолчанию для столбца таблицы.

- Если столбец основан на домене со значением по умолчанию — доменное значение перекроет это удаление.
- Если удаление значения по умолчанию производится над столбцом, у которого нет значения по умолчанию, или чьё значение по умолчанию основано на домене, то это приведёт к ошибке выполнения данного оператора.

Пример 118. Удаление значения по умолчанию для столбца

```
ALTER TABLE STOCK
ALTER COLUMN MODEL DROP DEFAULT;
```

Предложение SET DEFAULT устанавливает значение по умолчанию для столбца таблицы. Если столбец уже имел значение по умолчанию, то оно будет заменено новым. Значение по умолчанию для столбца всегда перекрывает доменное значение по умолчанию.

Пример 119. Установка значения по умолчанию для столбца

```
ALTER TABLE STOCK
ALTER COLUMN MODEL SET DEFAULT 1;
```

Установка и удаление ограничения NOT NULL

Предложение SET NOT NULL добавляет ограничение NOT NULL для столбца таблицы.

Успешное добавление ограничения NOT NULL происходит, только после полной проверки данных таблицы, для того чтобы убедится что столбец не содержит значений NULL.



Явное ограничение NOT NULL на столбце, базирующемся на домене, преобладает над установками домена. В этом случае изменение домена для допустимости значения NULL, не распространяется на столбец таблицы.

Пример 120. Добавление ограничения NOT NULL

```
ALTER TABLE STOCK
ALTER COLUMN PROPID SET NOT NULL;
```

Предложение DROP NOT NULL удаляет ограничение NOT NULL для столбца таблицы. Если столбец основан на домене с ограничением NOT NULL, то ограничение домена перекроет это удаление.

Пример 121. Удаление ограничения NOT NULL

```
ALTER TABLE STOCK
ALTER COLUMN ITEMID DROP NOT NULL;
```

Изменение столбцов идентификации

Для столбцов идентификации позволено изменять способ генерации, начальное значение и значение приращения.

Предложение SET GENERATED позволяет изменить способ генерации столбца идентификации. Существует два способа генерации столбца идентификации:

- BY DEFAULT столбцы позволяют переписать сгенерированное системой значение в операторах INSERT, UPDATE OR INSERT, MERGE просто указав значение этого столбца в списке значений.
- ALWAYS столбцы не позволяют переписать сгенерированное системой значение, при

попытке переписать значение такого столбца идентификации будет выдана ошибка. Переписать значение этого столбца в операторе INSERT можно только при указании директивы **OVERRIDING SYSTEM VALUE**.

Пример 122. Изменение способа генерации столбца идентификации

```
ALTER TABLE objects
ALTER ID SET GENERATED ALWAYS;
```

Если указано только предложение RESTART, то происходит сброс значения генератора в ноль. Необязательное предложение WITH позволяет указать для нового значения внутреннего генератора отличное от нуля значение.

Пример 123. Изменение текущего значения генератора для столбца идентификации

```
ALTER TABLE objects
ALTER ID RESTART WITH 100;
```

Предложение SET INCREMENT [BY] позволяет изменить значение приращения столбца идентификации. Значение приращения должно быть отлично от 0.

Пример 124. Изменение приращения столбца идентификации

```
ALTER TABLE objects
ALTER ID SET INCREMENT BY 2;
```

В одном операторе можно изменить сразу несколько свойств столбца идентификации, например:

Пример 125. Изменение нескольких свойств столбца идентификации

```
ALTER TABLE objects
ALTER ID SET GENERATED ALWAYS RESTART SET INCREMENT BY 2;
```

Предложение DROP IDENTITY удаляет связанный со столбцом идентификации системную последовательность и преобразует его в обычный столбец.

Пример 126. Превращение столбца идентификации в обычный столбец

```
ALTER TABLE objects
ALTER ID DROP IDENTITY;
```

Изменение вычисляемых столбцов

Для вычисляемых столбцов (GENERATED ALWAYS AS или COMPUTED BY) позволяет изменить тип и выражение вычисляемого столбца. Невозможно изменить обычный столбец на вычисляемый и наоборот.

Пример 127. Изменение вычисляемых столбцов

```
ALTER TABLE SALARY_HISTORY
ALTER NEW_SALARY GENERATED ALWAYS
AS (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
ALTER SALARY_CHANGE COMPUTED
BY (OLD_SALARY * PERCENT_CHANGE / 100);
```

Не изменяемые атрибуты

На данный момент не существует возможности изменить сортировку по умолчанию.

Предложение ALTER SQL SECURITY

Предложение ALTER SQL SECURITY позволяет изменить привилегии с которыми вычисляются вычисляемые столбцы. Если выбрана опция INVOKER, то вычисляемые столбцы вычисляются с привилегиями вызывающего пользователя. Если выбрана опция DEFINER, то вычисляемые столбцы вычисляются с привилегиями определяющего пользователя (владельца). По умолчанию вычисляемые столбцы вычисляются с привилегиями вызывающего пользователя. Кроме того триггеры наследуют привилегии выполнения у таблицы, если они не переопределены у самих триггеров.

```
ALTER TABLE COUNTRY
ALTER SQL SECURITY DEFINER;
```

Управление репликацией

Предложение ENABLE PUBLICATION включает таблицу в набор репликации (публикацию). Соответственно предложение DISABLE PUBLICATION исключает таблицу из набора репликации.

Пример 128. Добавление таблицы в набор репликации

```
ALTER TABLE COUNTRY
ENABLE PUBLICATION;
```

Кто может изменить таблицу?

Выполнить оператор ALTER TABLE могут:

- Администраторы
- Владелец таблицы;
- Пользователи с привилегией ALTER ANY TABLE.

См. также:

[CREATE TABLE](#), [RECREATE TABLE](#).

5.4.3. DROP TABLE

Назначение

Удаление существующей таблицы.

Доступно в

DSQL, ESQL.

Синтаксис

```
DROP TABLE tablename
```

Таблица 31. Параметры оператора DROP TABLE

Параметр	Описание
tablename	Имя таблицы.

Оператор DROP TABLE удаляет существующую таблицу. Если таблица имеет зависимости, то удаление не будет произведено. При удалении таблицы будут также удалены все триггеры на её события и индексы, построенные для её полей.

Пример 129. Удаление таблицы

```
DROP TABLE COUNTRY;
```

Кто может удалить таблицу?

Выполнить оператор DROP TABLE могут:

- Администраторы
- Владелец таблицы;
- Пользователи с привилегией DROP ANY TABLE.

См. также:

[CREATE TABLE](#), [RECREATE TABLE](#).

5.4.4. RECREATE TABLE

Назначение

Создание новой таблицы или пересоздание существующей.

Доступно в

DSQL.

Синтаксис

```
RECREATE [GLOBAL TEMPORARY] TABLE tablename
[EXTERNAL [FILE] 'filespec']
(<col_def> [, <col_def> | <tconstraint> ...])
[ON COMMIT {DELETE | PRESERVE} ROWS]
[SQL SECURITY {DEFINER | INVOKER}]
```

Полное описание определений столбцов и ограничений таблицы смотрите в разделе [CREATE TABLE](#).

Создаёт или пересоздаёт таблицу. Если таблица с таким именем уже существует, то оператор RECREATE TABLE попытается удалить её и создать новую. Оператор RECREATE TABLE не выполнится, если существующая таблица имеет зависимости.

Примеры

Пример 130. Создание или пересоздание таблицы

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

См. также:

[CREATE TABLE](#), [DROP TABLE](#).

5.5. INDEX

Индекс — это объект базы данных, используемый для более быстрого извлечения данных из таблицы или для ускорения сортировки в запросе. Кроме того, индексы используются для

обеспечения ограничений целостности — PRIMARY KEY, FOREIGN KEY, UNIQUE.

В данном разделе описываются вопросы создания индексов, перевода их в активное/неактивное состояние, удаление индексов и сбор статистики (пересчёт селективности) для индексов.

5.5.1. CREATE INDEX

Назначение

Создание индекса для таблицы.

Доступно в

DSQL, ESQL.

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

Таблица 32. Параметры оператора CREATE INDEX

Параметр	Описание
indexname	Имя индекса. Может содержать до 63 символов.
tablename	Имя таблицы, для которой строится индекс.
col	Столбец таблицы. В качестве столбцов не могут быть использованы поля типа BLOB, ARRAY и вычисляемые поля.
expression	Выражение, содержащее столбцы таблицы.

Оператор CREATE INDEX создаёт индекс для таблицы, который может быть использован для ускорения поиска, сортировки и/или группирования. Кроме того, индекс может быть использован при определении ограничений, таких как первичный ключ, внешний ключ или ограничениях уникальности. Индекс может быть построен на столбцах любого типа кроме BLOB и массивов. Имя индекса должно быть уникальным среди всех имён индексов.

Индексы в ключах

При добавлении ограничений первичного ключа, внешнего ключа или ограничения уникальности будет неявно создан одноименный индекс. Так, например, при выполнении следующего оператора будет неявно создан индекс PK_COUNTRY.



```
ALTER TABLE COUNTRY
ADD CONSTRAINT PK_COUNTRY PRIMARY KEY (ID);
```

Уникальные индексы

Если при создании индекса указано ключевое слово UNIQUE, то индекс гарантирует

的独特性。这样的索引被称为唯一索引。唯一索引不是唯一性的限制。

唯一的索引不能包含键值的副本（或组合键值的副本，在复合、多列或段落索引的情况下）。键值的副本允许在与 SQL-99 标准一致的情况下（包括在多段落索引中）。

Направление индекса

所有索引在 Firebird 中都是单向的。索引可以是升序或降序的。关键字 ASC[ENDING]（缩写为 ASC）和 DESC[ENDING] 用于指定索引的方向。默认情况下，创建的是升序的 ASC[ENDING] 索引。同时允许在同一列或同一集合上指定升序和降序的索引。



降序 (DESC[ENDING]) 索引可能在搜索最高值（最大值，最近值等）时很有用。

Вычисляемые индексы или индексы по выражению

在创建索引时，除了一个或多个列外，您还可以指定一个表达式，使用 COMPUTED BY 子句。这种索引称为计算索引或表达式索引。计算索引在查询中使用，其中 WHERE、ORDER BY 或 GROUP BY 子句中的条件与索引的表达式完全匹配。表达式可以在计算索引中使用多个列。

Ограничения на индексы

索引键的最大长度为页面大小的 1/4。

Ограничения на длину индексируемой строки

索引字符串的最大长度为 9 字节，小于键的长度。索引字符串的长度取决于页面大小和字符集。

Таблица 33. Длина индексируемой строки и набор символов

Размер страницы	Максимальная длина индексируемой строки для набора символов, байт/символ				
	1	2	3	4	6
4096	1015	507	338	253	169
8192	2039	1019	679	509	339
16384	4087	2043	1362	1021	681
32768	8183	4091	2727	2045	1363

Максимальное количество индексов на таблицу

Для каждой таблицы максимально возможное количество индексов ограничено и зависит от размера страницы и количества столбцов в индексе.

Таблица 34. Число индексов и количество столбцов

Размер страницы	Число индексов в зависимости от количества столбцов в индексе		
	1	2	3
4096	203	145	113
8192	408	291	227
16384	818	584	454
32768	1637	1169	909

Кто может создать индекс?

Выполнить оператор `CREATE INDEX` могут:

- Администраторы
- Владелец таблицы, для которой создаётся индекс;
- Пользователи с привилегией `ALTER ANY TABLE`.

Примеры

Пример 131. Создание индекса

```
CREATE INDEX IDX_UPDATER ON SALARY_HISTORY (UPDATER_ID);
```

Пример 132. Создание индекса с сортировкой ключей по убыванию

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

Пример 133. Создание многосегментного индекса

```
CREATE INDEX IDX_SALESTAT ON SALES (ORDER_STATUS, PAID);
```

Пример 134. Создание индекса, не допускающего дубликаты значений

```
CREATE UNIQUE INDEX UNQ_COUNTRY_NAME ON COUNTRY (NAME);
```

Пример 135. Создание вычисляемого индекса

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

Такой индекс может быть использован для регистронезависимого поиска.

```
SELECT *
FROM PERSONS
WHERE UPPER(NAME) STARTING WITH UPPER('Iv');
```

См. также:

[ALTER INDEX, DROP INDEX.](#)

5.5.2. ALTER INDEX

Назначение

Перевод индекса в активное/неактивное состояние, перестройка индекса.

Доступно в

DSQL, ESQL.

Синтаксис

```
ALTER INDEX indexname {ACTIVE | INACTIVE};
```

Таблица 35. Параметры оператора ALTER INDEX

Параметр	Описание
indexname	Имя индекса.

Оператор ALTER INDEX переводит индекс в активное/неактивное состояние. Возможность изменения структуры и порядка сортировки ключей этот оператор не предусматривает.

INACTIVE

При выборе опции INACTIVE, индекс переводится из активного в неактивное состояние. Перевод индекса в неактивное состояние по своему действию похоже на команду DROP INDEX за исключением того, что определение индекса сохраняется в базе данных. Невозможно перевести в неактивное состояние индекс участвующий в ограничении.

Активный индекс может быть отключен, только если отсутствуют запросы использующие этот индекс, иначе будет возвращена ошибка “object in use”.

Активация неактивного индекс также безопасна. Тем не менее, если есть активные транзакции, модифицирующие таблицу, то транзакция, содержащая оператор ALTER INDEX потерпит неудачу, если она имеет атрибут NO WAIT. Если транзакция находится в режиме WAIT, то она будет ждать завершения параллельных транзакций.

С другой стороны, если наш оператор ALTER INDEX начинает перестраивать индекс на COMMIT, то другие транзакции, изменяющие эту таблицу, потерпят неудачу или будут ожидать в соответствии с их WAIT/NO WAIT атрибутами. Та же самая ситуация будет и при выполнении CREATE INDEX.



Перевод индекса в неактивное состояние может быть полезен при массовой вставке, модификации или удалении записей из таблицы, для которой этот индекс построен.

ACTIVE

При выборе альтернативы ACTIVE индекс переводится из неактивного состояния в активное. При переводе индекса из неактивного состояния в активное индекс перестраивается.



Даже если индекс находится в активном состоянии оператор ALTER INDEX … ACTIVE всё равно перестраивает индекс. Таким образом, эту команду можно использовать как часть обслуживания БД для перестройки индексов, автоматически созданных для ограничений PRIMARY KEY, FOREIGN KEY, UNIQUE, для которых выполнение оператора ALTER INDEX … INACTIVE невозможно.

Использование ALTER INDEX для индексов ограничений

Принудительный перевод индексов созданных для ограничений PRIMARY KEY, FOREIGN KEY и UNIQUE не допускается. Тем не менее выполнение оператора ALTER INDEX … INACTIVE работает так же хорошо для индексов ограничений, как и другие инструменты для других индексов.

Кто может выполнить ALTER INDEX?

Выполнить оператор ALTER INDEX могут:

- Администраторы
- Владелец таблицы, для которой построен индекс;
- Пользователи с привилегией ALTER ANY TABLE.

Примеры

Пример 136. Перевод индекса в неактивное состояние

```
ALTER INDEX IDX_UPDATER INACTIVE;
```

Пример 137. Возврат индекса в активное состояние

```
ALTER INDEX IDX_UPDATER ACTIVE;
```

См. также:

CREATE INDEX, DROP INDEX.

5.5.3. DROP INDEX

Назначение

Удаление индекса из базы данных.

Доступно в

DSQL, ESQL.

Синтаксис

```
DROP INDEX indexname
```

Таблица 36. Параметры оператора DROP INDEX

Параметр	Описание
indexname	Имя индекса.

Оператор **DROP INDEX** удаляет существующий индекс из базы данных. При наличии зависимостей для существующего индекса (если он используется в ограничении) удаление не будет выполнено.

Кто может удалить индекс?

Выполнить оператор **DROP INDEX** могут:

- Администраторы
- Владелец таблицы, для которой построен индекс;
- Пользователи с привилегией **ALTER ANY TABLE**.

Примеры

Пример 138. Удаление индекса

```
DROP INDEX IDX_UPDATER;
```

См. также:

[CREATE INDEX](#), [ALTER INDEX](#).

5.5.4. SET STATISTICS

Назначение

Пересчёт селективности индекса.

Доступно в

DSQL, ESQL.

Синтаксис

```
SET STATISTICS INDEX indexname
```

Таблица 37. Параметры оператора SET STATISTICS

Параметр	Описание
indexname	Имя индекса.

Оператор SET STATISTICS пересчитывает значение селективности для указанного индекса.

Селективность индекса

Селективность (избирательность) индекса — это оценочное количество строк, которые могут быть выбраны при поиске по каждому значению индекса. Уникальный индекс имеет максимальную селективность, поскольку при его использовании невозможно выбрать более одной строки для каждого значения ключа индекса. Актуальность селективности индекса важна для выбора наиболее оптимального плана выполнения запросов оптимизатором.

Пересчёт селективности индекса может потребоваться после массовой вставки, модификации или удалении большого количества записей из таблицы, поскольку она становится неактуальной.



Отметим, что в Firebird статистика индексов автоматически не пересчитывается ни после массовых изменений данных, ни при каких либо других условиях. При создании (CREATE) или его активации (ALTER INDEX ACTIVE) статистика индекса полностью соответствует его содержимому.

Пересчёт селективности индекса может быть выполнен под высокой параллельной нагрузкой без риска его повреждения. Тем не менее следует помнить, что при высоком

параллелизме рассчитанная статистика может устареть, как только закончится выполнение оператора SET STATISTICS.

Кто может обновить статистику?

Выполнить оператор SET STATISTICS могут:

- Администраторы
- Владелец таблицы, для которой построен индекс;
- Пользователи с привилегией ALTER ANY TABLE.

Примеры

Пример 139. Пересчёт селективности индекса IDX_UPDATER

```
SET STATISTICS INDEX IDX_UPDATER;
```

См. также:

[CREATE INDEX, ALTER INDEX.](#)

5.6. VIEW

Представление (view) — виртуальная таблица, которая по своей сути является именованным запросом SELECT выборки данных произвольной сложности. Выборка данных может осуществляться из одной и более таблиц, других представлений, а также селективных хранимых процедур.

В отличие от обычных таблиц реляционных баз данных, представление не является самостоятельным набором данных, хранящимся в базе данных. Результат в виде набора данных динамически создаётся при обращении к представлению.

Метаданные представлений доступны для генерации двоичного кода хранимых процедур, функций, пакетов и триггеров так, как будто они являются обычной таблицей, хранящей постоянные данные.

5.6.1. CREATE VIEW

Назначение

Создание нового представления.

Доступно в

DSQL

Синтаксис

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

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

Таблица 38. Параметры оператора *CREATE VIEW*

Параметр	Описание
viewname	Имя представления. Может содержать до 63 символов.
select_statement	Оператор SELECT.
full_column_list	Список столбцов представления.
colname	Имя столбца представления. Дубликаты имён столбцов не позволяются.

Оператор *CREATE VIEW* создаёт новое представление. Имя представления должно быть уникальным среди имён всех представлений, таблиц и хранимых процедур базы данных.

После имени создаваемого представления может идти список имён столбцов, получаемых в результате обращения к представлению. Имена в списке могут быть никак не связаны с именами столбцов базовых таблиц. При этом их количество должно точно соответствовать количеству столбцов в списке выбора главного оператора *SELECT* представления.

Если список столбцов представления отсутствует, то будут использоваться имена столбцов базовых таблиц или псевдонимов (алиасов) полей оператора *SELECT*. Если имена полей повторяются или присутствуют выражения столбцов без псевдонимов, которые делают невозможным получение допустимого списка столбцов, то создание представления завершается ошибкой.

Количество столбцов в списке столбцов представления должно совпадать с количеством столбцов указанном в списке выбора оператора *SELECT* указанного в определении представления.

Дополнительные моменты



- Если указан полный список столбцов, то задание псевдонимов в операторе *SELECT* не имеет смысла, поскольку они будут переопределены именами из списка столбцов;
- Список столбцов необязателен при условии, что все столбцы в операторе *SELECT* имеют явное имя, и эти имена будут уникальными в списке столбцов.

Обновляемые представления

Представление может быть обновляемым и только для чтения. Если представление обновляемое, то данные, полученные при обращении к такому представлению, можно

изменить при помощи DML операторов `INSERT`, `UPDATE`, `DELETE`, `UPDATE OR INSERT`, `MERGE`. Изменения, выполняемые над представлением, применяются к базовой таблице(ам).

Представление только для чтения можно сделать обновляемым при помощи вспомогательных триггеров. После того как на представлении будет определён один или несколько триггеров, то изменения не будут автоматически попадать в базовую таблицу, даже если перед этим представление было обновляемым. В этом случае ответственность за обновление (удаление или вставку) записей базовых таблиц, лежит на программисте, определяющем триггеры.

Для того чтобы представление было обновляемым, необходимо выполнение следующих условий:

- оператор выборки `SELECT` обращается только к одной таблице или одному изменяемому представлению;
- оператор выборки `SELECT` не должен обращаться к хранимым процедурам;
- все столбцы базовой таблицы или обновляемого представления, которые не присутствуют в данном представлении, должны удовлетворять одному из следующих условий:
 - позволять значение `NULL`
 - `NOT NULL` столбцы должны иметь значение по умолчанию
 - значение `NOT NULL` столбцов должны быть инициализированы в триггерах базовых таблиц
- оператор выборки `SELECT` не содержит полей определённых через подзапросы или другие выражения;
- оператор выборки `SELECT` не содержит полей определённых через агрегатные функции (`MIN`, `MAX`, `AVG`, `COUNT`, `LIST`), статистические функции (`CORR`, `COVAR_POP`, `COVAR_SAMP` и др.), функции линейной регрессии (`REGR_AVGX`, `REGR_AVGY` и др.) и все виды оконных функций;
- оператор выборки `SELECT` не содержит предложений `ORDER BY`, `GROUP BY`, `HAVING`;
- оператор выборки `SELECT` не содержит ключевого слова `DISTINCT` и ограничений количества строк с помощью `ROWS`, `FIRST/SKIP`, `OFFSET/FETCH`.

WITH CHECK OPTIONS

Необязательное предложение `WITH CHECK OPTIONS` задаёт для изменяемого представления требования проверки вновь введённых или модифицируемых данных условию, указанному в предложении `WHERE` оператора выборки `SELECT`. При попытке вставки новой записи или модификации записи проверяется, выполняется ли для этой записи условие в предложении `WHERE`, если условие не выполняется, то вставка/модификация не выполняется и будет выдано соответствующее диагностическое сообщение.

Предложение `WITH CHECK OPTION` может задаваться в операторе создания представления только в том случае, если в главном операторе `SELECT` представления указано предложение `WHERE`. Иначе будет выдано сообщение об ошибке.

Если используется предложение `WITH CHECK OPTIONS`, то система проверяет входные значения на соответствие условию в предложении `WHERE` до того как они будут переданы в базовую таблицу. Таким образом, если входные значения не проходят проверку, то предложения `DEFAULT` или триггеры на базовой таблице, не могут исправить входные значения, поскольку действия никогда не будут выполнены.



Кроме того, поля представления не указанные в операторе `INSERT` передаются в базовую таблицу как значения `NULL`, независимо от их наличия или отсутствия в предложении `WHERE`. В результате значения по умолчанию, определённые на таких полях базовой таблицы, не будут применены. С другой стороны, триггеры будут вызываться и работать как ожидалось.

Для представлений у которых отсутствует предложение `WITH CHECK OPTIONS`, поля, отсутствующие в операторе `INSERT`, не передаются вовсе, поэтому любые значения по умолчанию будут применены.

Привилегии выполнения

Выполнение SQL кода представлений всегда осуществляется с привилегиями определяющего пользователя (владельца).

Кто может создать представление?

Выполнить оператор `CREATE VIEW` могут:

- Администраторы
- Пользователи с привилегией `CREATE VIEW`.

Пользователь, создавший представление, становится его владельцем.

Для создания представления пользователями, которые не имеют административных привилегий, необходимы также привилегии на чтение (`SELECT`) данных из базовых таблиц и представлений, и привилегии на выполнение (`EXECUTE`) используемых селективных хранимых процедур.

Для разрешения вставки, обновления и удаления через представление, необходимо чтобы создатель (владелец) представления имел привилегии `INSERT`, `UPDATE` и `DELETE` на базовые объекты метаданных.

Предоставить привилегии на представление другим пользователям возможно только если владелец представления сам имеет эти привилегии на базовых объектах. Она будет всегда, если владелец представления является владельцем базовых объектов метаданных.

Примеры

Пример 140. Создание представления

```
CREATE VIEW ENTRY_LEVEL_JOBS AS
SELECT JOB_CODE, JOB_TITLE
FROM JOB
WHERE MAX_SALARY < 15000;
```

Пример 141. Создание представления с проверкой условия фильтрации

Создание представления возвращающего столбцы JOB_CODE и JOB_TITLE только для тех работ, где MAX_SALARY меньше \$15000. При вставке новой записи или изменении существующей будет осуществляться проверка условия MAX_SALARY < 15000, если условие не выполняется, то вставка/изменение будет отвергнуто.

```
CREATE VIEW ENTRY_LEVEL_JOBS AS
SELECT JOB_CODE, JOB_TITLE
FROM JOB
WHERE MAX_SALARY < 15000
WITH CHECK OPTIONS;
```

Пример 142. Создание представления с использованием списка столбцов

```
CREATE VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST_WITH_MARKUP
) AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.1
FROM PRICE;
```

Пример 143. Создание представления с использованием псевдонимов полей

```
CREATE VIEW PRICE_WITH_MARKUP AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.1 AS COST_WITH_MARKUP
FROM PRICE;
```

Пример 144. Создание необновляемого представления с использованием хранимой процедуры

```
CREATE VIEW GOODS_PRICE AS
SELECT
    goods.name AS goodsname,
    price.cost AS cost,
    b.quantity AS quantity
FROM
    goods
JOIN price ON goods.code_goods = price.code_goods
LEFT JOIN sp_get_balance(goods.code_goods) b ON 1 = 1;
```

Пример 145. Создание обновляемого представления с использованием триггеров

```
-- базовые таблицы
RECREATE TABLE t_films(id INT PRIMARY KEY, title VARCHAR(100));
RECREATE TABLE t_sound(id INT PRIMARY KEY, audio BLOB);
RECREATE TABLE t_video(id INT PRIMARY KEY, video BLOB);
COMMIT;

-- создание необновляемого представления
RECREATE VIEW v_films AS
    SELECT f.id, f.title, s.audio, v.video
    FROM t_films f
    LEFT JOIN t_sound s ON f.id = s.id
    LEFT JOIN t_video v ON f.id = v.id;

/* Для того чтобы сделать представление обновляемым создадим
триггер, который будет производить манипуляции над базовыми
таблицами.
*/
SET TERM ^;
CREATE OR ALTER TRIGGER v_films_biud FOR v_films
ACTIVE BEFORE INSERT OR UPDATE OR DELETE POSITION 0 AS
BEGIN
    IF (INSERTING) THEN
        new.id = COALESCE(new.id, GEN_ID(g_films, 1));
    IF (NOT DELETING) THEN
        BEGIN
            UPDATE OR INSERT INTO t_films(id, title)
            VALUES(new.id, new.title)
            MATCHING(id);

            UPDATE OR INSERT INTO t_sound(id, audio)
            VALUES(new.id, new.audio)
            MATCHING(id);

            UPDATE OR INSERT INTO t_video(id, video)
            VALUES(new.id, new.video)
            MATCHING(id);
        END
    ELSE
        BEGIN
            DELETE FROM t_films WHERE id = old.id;
            DELETE FROM t_sound WHERE id = old.id;
            DELETE FROM t_video WHERE id = old.id;
        END
END^
SET TERM ;^

/*
* Теперь мы можем производить манипуляции над

```

```
* этим представлением как будто мы работаем с таблицей
*/
INSERT INTO v_films(title, audio, video)
VALUES('007 coordinates skyfall', 'pif-paf!', 'oh! waw!');
```

См. также:

ALTER VIEW, CREATE OR ALTER VIEW, RECREATE VIEW, DROP VIEW.

5.6.2. ALTER VIEW

Назначение

Изменение существующего представления.

Доступно в

DSQL

Синтаксис

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

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

Таблица 39. Параметры оператора ALTER VIEW

Параметр	Описание
viewname	Имя существующего представления.
select_statement	Оператор SELECT.
full_column_list	Список столбцов представления.
colname	Имя столбца представления. Дубликаты имён столбцов не позволяются.

Оператор ALTER VIEW изменяет определение существующего представления, существующие права на представления и зависимости при этом сохраняются. Синтаксис оператора ALTER VIEW полностью аналогичен синтаксису оператора CREATE VIEW.



Будьте осторожны при изменении количества столбцов представления. Существующий код приложения может стать неработоспособным. Кроме того, PSQL модули, использующие изменённое представление, могут стать некорректными. Информация о том, как это обнаружить, находится в приложении [Поле RDB\\$VALID_BLR](#).

Кто может изменить представление?

Выполнить оператор ALTER VIEW могут:

- Администраторы
- Владелец представления;
- Пользователи с привилегией ALTER ANY VIEW.

Примеры

Пример 146. Изменение представления

```
ALTER VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST_WITH_MARKUP
) AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.15
FROM PRICE;
```

См. также:

[CREATE VIEW](#), [CREATE OR ALTER VIEW](#), [RECREATE VIEW](#).

5.6.3. CREATE OR ALTER VIEW

Назначение

Создание нового или изменение существующего представления.

Доступно в

DSQL

Синтаксис

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

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

Таблица 40. Параметры оператора CREATE OR ALTER VIEW

Параметр	Описание
viewname	Имя представления. Может содержать до 63 символов.
select_statement	Оператор SELECT.
full_column_list	Список столбцов представления.

Параметр	Описание
colname	Имя столбца представления. Дубликаты имён столбцов не позволяются.

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

Примеры

Пример 147. Создание нового или изменение существующего представления

```
CREATE OR ALTER VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST_WITH_MARKUP
) AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.15
FROM PRICE;
```

См. также:

[CREATE VIEW](#), [ALTER VIEW](#), [RECREATE VIEW](#).

5.6.4. DROP VIEW

Назначение

Удаление существующего представления.

Доступно в

DSQL

Синтаксис

```
DROP VIEW viewname
```

Таблица 41. Параметры оператора DROP VIEW

Параметр	Описание
viewname	Имя представления.

Оператор DROP VIEW удаляет существующее представление. Если представление имеет зависимости, то удаление не будет произведено.

Кто может удалить представление?

Выполнить оператор `DROP VIEW` могут:

- Администраторы
- Владелец представления;
- Пользователи с привилегией `DROP ANY VIEW`.

Примеры

Пример 148. Удаление представления

```
DROP VIEW PRICE_WITH_MARKUP;
```

См. также:

`CREATE VIEW`, `RECREATE VIEW`.

5.6.5. RECREATE VIEW

Назначение

Создание нового или пересоздание существующего представления.

Доступно в

DSQL

Синтаксис

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

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

Таблица 42. Параметры оператора `RECREATE VIEW`

Параметр	Описание
<code>viewname</code>	Имя представления. Может содержать до 63 символов.
<code>select_statement</code>	Оператор <code>SELECT</code> .
<code>full_column_list</code>	Список столбцов представления.
<code>colname</code>	Имя столбца представления. Дубликаты имён столбцов не позволяются.

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 имеет зависимости.

Примеры

Пример 149. Создание нового или пересоздание существующего представления

```
RECREATE VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST_WITH_MARKUP
) AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.15
FROM PRICE;
```

См. также:

[CREATE VIEW](#), [CREATE OR VIEW](#), [DROP VIEW](#).

5.7. TRIGGER

Триггер (trigger)—это хранимая процедура особого типа, которая не вызывается непосредственно, а исполнение которой обусловлено наступлением одного из событий, относящегося к одной конкретной таблице (представлению), или наступлению одного из событий базы данных.

Триггер—это особый тип хранимой процедуры, которая не вызывается напрямую, а выполняется, когда в связанной таблице или представлении происходит указанное событие. DML триггер специфичен для одного и только одного отношения (таблица или представление) и одной фазы во времени события (ДО или ПОСЛЕ). Его можно задать для выполнения для одного конкретного события (вставка, обновление, удаление) или для некоторой комбинации двух или трех из этих событий.

Помимо DML триггеров существуют также:

- * Триггеры на события базы данных, которые происходят при начале или завершении соединения, или транзакции.
- * DDL триггеры, которые срабатывают до или после выполнения одного или нескольких типов DDL операторов.

5.7.1. CREATE TRIGGER

Назначение

Создание нового триггера.

Доступно в

DSQL, ESQL

Синтаксис

```

CREATE TRIGGER trigname {
    <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger> }
<routine body>

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

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

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

<ddl_trigger> ::==
    [ACTIVE | INACTIVE]
    {BEFORE | AFTER} <ddl_events>
    [POSITION number]

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= INSERT | UPDATE | DELETE

<db_event> ::==
    CONNECT | DISCONNECT
  | TRANSACTION {START | COMMIT | ROLLBACK}

<ddl_events> ::= {
    ANY DDL STATEMENT
  | <ddl_event_item> [{OR <ddl_event_item>} ...]
}

<ddl_event_item> ::==
    {CREATE | ALTER | DROP} TABLE
  | {CREATE | ALTER | DROP} PROCEDURE
  | {CREATE | ALTER | DROP} FUNCTION
  | {CREATE | ALTER | DROP} TRIGGER
  | {CREATE | ALTER | DROP} EXCEPTION
  | {CREATE | ALTER | DROP} VIEW
  | {CREATE | ALTER | DROP} DOMAIN

```

```

| {CREATE | ALTER | DROP} ROLE
| {CREATE | ALTER | DROP} SEQUENCE
| {CREATE | ALTER | DROP} USER
| {CREATE | ALTER | DROP} INDEX
| {CREATE | DROP} COLLATION
| ALTER CHARACTER SET
| {CREATE | ALTER | DROP} PACKAGE
| {CREATE | DROP} PACKAGE BODY
| {CREATE | ALTER | DROP} MAPPING

<routine body> ::=

  <SQL routine spec>
  | <external body reference>

<SQL routine spec> ::=

  [<rights clause>] <SQL routine body>

<rights clause> ::=

  SQL SECURITY {DEFINER | INVOKER}

<SQL routine body> ::=

  AS
  [<declarations>]
  BEGIN
  [<PSQL_statements>]
  END

<declarations> ::= <declare_item> [<declare_item> ...]

<declare_item> ::=

  <declare_var>;
  | <declare_cursor>;
  | <subroutine declaration>;
  | <subroutine implementation>

<subroutine declaration> ::= <subfunc_decl> | <subproc_decl>

<subroutine implementation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=

  EXTERNAL NAME '<extname>' ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

```

Таблица 43. Параметры оператора CREATE TRIGGER

Параметр	Описание
trigname	Имя триггера. Может содержать до 63 символов.
relation_trigger_legacy	Объявление табличного триггера (унаследованное).

Параметр	Описание
relation_trigger_sql2003	Объявление табличного триггера согласно стандарту SQL-2003.
database_trigger	Объявление триггера базы данных.
ddl_trigger	Объявление DDL триггера.
tablename	Имя таблицы.
viewname	Имя представления.
mutation_list	Список событий таблицы.
mutation	Одно из событий таблицы.
db_event	Событие соединения или транзакции.
ddl_events	Список событий изменения метаданных.
ddl_event_item	Одно из событий изменения метаданных.
number	Порядок срабатывания триггера. От 0 до 32767.
declarations	Секция объявления локальных переменных, подпрограмм и именованных курсоров.
PSQL_statments	Операторы языка PSQL.
extbody	Тело внешнего триггера. Строковый литерал который может использоваться UDR для различных целей.
module name	Имя внешнего модуля.
routine name	Имя точки входа внутри модуля.
misc info	Различная информация используемая внешним триггером по своему усмотрению.

Оператор `CREATE TRIGGER` создаёт новый триггер. Триггер может быть создан для события (или событий) отношения (таблицы или представления), для события (событий) изменения метаданных или для одного из событий базы данных.

Оператор `CREATE TRIGGER` как и его родственники `ALTER TRIGGER`, `CREATE OR ALTER TRIGGER` и `RECREATE TRIGGER` являются составными операторами, содержащими заголовок и тело.

Заголовок определяет имя триггера, а также содержит имя отношения (для табличных триггеров), фазу триггера, событие (или события) на которые срабатывает триггер и позицию. Имя триггера должно быть уникальным среди имён других триггеров.

Привилегии выполнения

Необязательное предложение `SQL SECURITY` позволяет задать с какими привилегиями выполняется триггер. Если выбрана опция `INVOKER`, то триггер выполняется с привилегиями вызывающего пользователя. Если выбрана опция `DEFINER`, то триггер выполняется с привилегиями определяющего пользователя (владельца). Эти привилегии будут дополнены привилегиями выданные самому триggerу с помощью оператора `GRANT`. По умолчанию триггер наследует привилегии выполнения указанные для таблицы. Триггера на события

базы данных по умолчанию выполняются с привилегиями определяющего пользователя (владельца).

Тело триггера

Тело триггера состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов или блоков операторов, заключённых во внешнем блоке, который начинается с ключевого слова BEGIN и заканчивается ключевым словом END. Объявления и внутренние операторы завершаются точкой с запятой (;).

Терминатор оператора

Некоторые редакторы SQL-операторов — в частности утилита isql, которая идёт в комплекте с Firebird, и возможно некоторые сторонние редакторы — используют внутреннее соглашение, которое требует, чтобы все операторы были завершены с точкой с запятой.

Это создает конфликт с синтаксисом PSQL при кодировании в этих средах. Если вы не знакомы с этой проблемой и её решением, пожалуйста, изучите детали в главе PSQL в разделе, озаглавленном [Изменение терминатора в isql](#).

DML триггеры (на таблицу или представление)

DML триггеры выполняются на уровне строки (записи) каждый раз, когда изменяется образ строки. Они могут быть определены и для таблиц и представлений.

Форма объявления

Объявление DML триггера существует в двух вариантах:

- унаследованная форма;
- SQL-2003 совместимая (рекомендуемая).

В настоящее время рекомендуется использовать SQL-2003 совместимую форму.

Для DML триггера обязательно указывается фаза и одно или несколько событий.

Состояние триггера

Триггер может быть в одном из двух состояний активном (ACTIVE) или неактивном (INACTIVE). Запускаются только активные триггеры. По умолчанию триггеры создаются в активном состоянии.

Фаза

Триггер может выполняться в одной из двух фаз, связанных с запрошенными изменениями состояния данных. Ключевое слово BEFORE означает, что триггер вызывается до наступления соответствующего события (событий, если их указано несколько), AFTER — после наступления события (событий).

События

Для DML триггера может быть указано одно из событий таблицы (представления) — **INSERT** (добавление), **UPDATE** (изменение), **DELETE** (удаление) — или несколько событий, разделённых ключевым словом **OR**, при которых вызывается триггер. При создании триггера каждое событие (**INSERT**, **UPDATE** или **DELETE**) не должно упоминаться более одного раза.

Контекстные переменные **INSERTING**, **UPDATING** и **DELETING** логического типа могут быть использованы в теле триггера для определения события, которое вызвало срабатывание триггера.

Порядок срабатывания

Ключевое слово **POSITION** позволяет задать порядок, в котором будут выполняться триггеры с одинаковой фазой и событием (или группы событий). По умолчанию позиция равна 0. Если позиции для триггеров не заданы или несколько триггеров имеют одно и то же значение позиции, то такие триггеры будут выполняться в алфавитном порядке их имен.

Тело триггера

После ключевого слова **AS** следует тело триггера.

Объявление локальных переменных, курсоров и подпрограмм

В необязательной секции **declarations** описаны локальные переменные триггера, именованные курсоры и подпрограммы (подпроцедуры и подфункции). Подробности вы можете посмотреть в главе “Процедурный язык PSQL” в разделах **DECLARE VARIABLE** и **DECLARE CURSOR, DECLARE PROCEDURE, DECLARE FUNCTION**.

После необязательной секции деклараций обязательно следует составной оператор. Составной оператор состоит из одного или нескольких PSQL операторов, заключенных между ключевыми словами **BEGIN** и **END**. Составной оператор может содержать один или несколько других составных операторов. Вложенность ограничена 512 уровнями. Любой из **BEGIN ... END** блоков может быть пустым, в том числе и главный блок.

Внешние триггеры

Триггер может быть расположена во внешнем модуле. В этом случае вместо тела триггера указывается место его расположения во внешнем модуле с помощью предложения **EXTERNAL NAME**. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя процедуры внутри модуля и определённая пользователем информация. В предложении **ENGINE** указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок **UDR**. После ключевого слова **AS** может быть указан строковый литерал — “тело” внешнего триггера, оно может быть использовано внешним модулем для различных целей.

Кто может создать DML триггер?

DML триггеры могут создать:

- Администраторы

- Владелец таблицы (представления);
- Пользователи с привилегией ALTER ANY {TABLE | VIEW}.

Примеры

Пример 150. Создание DML триггера в Legacy стиле

```
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.CUST_NO IS NULL) THEN
    NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
END
```

Пример 151. Создание DML триггера согласно стандарту SQL-2003

```
CREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT ON customer POSITION 0
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

Пример 152. Создание DML триггера выполняющегося с правами определяющего пользователя

```
CREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT ON customer POSITION 0
SQL SECURITY DEFINER
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

Пример 153. Создание DML триггера на несколько событий

```

CREATE TRIGGER TR_CUST_LOG
ACTIVE AFTER INSERT OR UPDATE OR DELETE
ON CUSTOMER POSITION 10
AS
BEGIN
    INSERT INTO CHANGE_LOG (LOG_ID,
                           ID_TABLE,
                           TABLE_NAME,
                           MUTATION)
VALUES (NEXT VALUE FOR SEQ_CHANGE_LOG,
        OLD.CUST_NO,
        'CUSTOMER',
        CASE
            WHEN INSERTING THEN 'INSERT'
            WHEN UPDATING THEN 'UPDATE'
            WHEN DELETING THEN 'DELETE'
        END);
END

```

См. также:

[ALTER TRIGGER, DROP TRIGGER.](#)

Триггеры на событие базы данных

Триггер может быть создан для одного из событий базы данных:

- CONNECT (соединение с базой данных или после сброса сеанса);
- DISCONNECT (отсоединение от базы данных или перед сбросом сеанса);
- TRANSACTION START (старт транзакции);
- TRANSACTION COMMIT (подтверждение транзакции);
- TRANSACTION ROLLBACK (откат транзакции).

Контекстная переменная **RESETTING** может использоваться в триггерах на события CONNECT и DISCONNECT для того, чтобы отличить сброс сеанса от подключения/отключения от базы данных.

Указать для триггера несколько событий базы данных невозможно.

Выполнение триггеров на событие базы данных и обработка исключений

Триггеры 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 возвращаются клиенту.

Триггеры на события CONNECT и DISCONNECT срабатывают также при выполнении оператора сброса сессионного окружения. Особенности обработки ошибок в триггерах на события CONNECT и DISCONNECT смотри в секции [ALTER SESSION RESET](#).

Триггеры на события транзакций срабатывают при старте транзакции, её подтверждении или откате. Не перехваченные исключения обрабатываются в зависимости от типа события:

- для события TRANSACTION START исключение возвращается клиенту, а транзакция отменяется;
- для события TRANSACTION COMMIT исключение возвращается клиенту, действия, выполненные триггером, и транзакция отменяются;
- для события TRANSACTION ROLLBACK исключение не возвращается клиенту, а транзакция, как и предусмотрено, отменяется.

Ловушки

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 до тех пор, пока вы не отключите триггеры базы данных и не исправите ошибочный код.

Отключение триггеров

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.

Двухфазное подтверждение транзакций

В случае двухфазных транзакций триггеры на событие TRANSACTION START срабатывают в фазе подготовки (reprepare), а не в фазе commit.

Предостережения

1. Триггеры для событий базы данных DISCONNECT и ROLLBACK не будут вызваны при отключении клиентов через таблицы мониторинга (DELETE FROM MON\$ATTACHMENTS).

2. Использование оператора IN AUTONOMOUS TRANSACTION DO в триггерах на событие базы данных связанные с транзакциями (COMMIT, ROLLBACK, START) может привести к его заикливанию.

Кто может создать триггеры на события базы данных?

Триггеры для событий базы данных могут создать:

- Администраторы
- Владелец базы данных;
- Пользователи с привилегией ALTER DATABASE.

Примеры

Пример 154. Создание триггера на событие подключения к БД для логирования события

```
CREATE TRIGGER tr_log_connect
INACTIVE ON CONNECT POSITION 0
AS
BEGIN
    INSERT INTO LOG_CONNECT (ID,
                             USERNAME,
                             ATIME)
    VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT,
            CURRENT_USER,
            CURRENT_TIMESTAMP);
END
```

Пример 155. Создание триггера на событие подключения к БД для контроля доступа

```
CREATE EXCEPTION E_INCORRECT_WORKTIME 'Рабочий день ещё не начался';

CREATE TRIGGER TR_LIMIT_WORKTIME ACTIVE
ON CONNECT POSITION 1
AS
BEGIN
    IF ((CURRENT_USER <> 'SYSDBA') AND
        NOT (CURRENT_TIME BETWEEN time '9:00' AND time '17:00')) THEN
        EXCEPTION E_INCORRECT_WORKTIME;
END
```

См. также:

[ALTER TRIGGER, DROP TRIGGER.](#)

Триггеры на события изменения метаданных

Триггеры на события изменения метаданных (DDL триггеры) предназначены для обеспечения ограничений, которые будут распространены на пользователей, которые пытаются создать, изменить или удалить DDL объект. Другое их назначение — ведение журнала изменений метаданных.

Триггеры на события изменения метаданных являются одним из подвидов триггеров на события базы данных.

Особенности:

1. BEFORE триггеры запускаются до изменений в системных таблицах. AFTER триггеры запускаются после изменений в системных таблицах.
2. Когда оператор DDL запускает триггер, в котором возбуждается исключение (BEFORE или AFTER, преднамеренно или неумышленно), оператор не будет фиксирован. Т.е. исключения могут использоваться, чтобы гарантировать, что оператор DDL будет отменен, если некоторые условия не будут соблюдены.
3. Действия DDL триггеров выполняются только при фиксации транзакции, в которой работает затронутая DDL команда. Никогда не забывайте о том, что в AFTER триггере, возможно сделать только то, что возможно сделать после DDL команды без автоматической фиксации транзакций. Вы не можете, например, создать таблицу в триггере и использовать её там.
4. Для операторов CREATE OR ALTER … триггер срабатывает один раз для события CREATE или события ALTER, в зависимости от того существовал ли ранее объект. Для операторов RECREATE триггер вызывается для события DROP, если объект существовал, и после этого для события CREATE.
5. Если объект метаданных не существует, то обычно триггеры на события ALTER и DROP не запускаются. Исключения описаны в пункте 6.
6. Исключением из правила 5 являются BEFORE {ALTER | DROP} USER триггеры, которые будут вызваны, даже если имя пользователя не существует. Это вызвано тем, что эти команды выполняются для базы данных безопасности, для которой не делается проверка существования пользователей перед их выполнением. Данное поведение, вероятно, будет отличаться для встроенных пользователей, поэтому не пишите код, который зависит от этого.
7. Если некоторое исключение возбуждено после того как начала выполняться DDL команда и до того как запущен AFTER триггер, то AFTER триггер не запускается.
8. Для процедур и функций в составе пакетов не запускаются индивидуальные триггеры {CREATE | ALTER | DROP} {PROCEDURE | FUNCTION}.
9. Оператор ALTER DOMAIN old name TO new name устанавливает контекстные переменные OLD_OBJECT_NAME и NEW_OBJECT_NAME в обоих триггерах BEFORE и AFTER. Контекстная переменная OBJECT_NAME будет содержать старое имя объекта метаданных в триггере BEFORE, и новое — в триггере AFTER.

Если в качестве события указано предложение ANY DDL STATEMENT, то триггер будет вызван при наступлении любого из DDL событий.

Пространство имён DDL_TRIGGER

Во время работы DDL триггера доступно пространство имён DDL_TRIGGER для использования в функции RDB\$GET_CONTEXT. Его использование также допустимо в хранимых процедурах и функциях, вызванных триггерами DDL.

Контекст DDL_TRIGGER работает как стек. Перед возбуждением DDL триггера, значения, относящиеся к выполняемой команде, помещаются в этот стек. После завершения работы триггера значения выталкиваются. Таким образом. В случае каскадных DDL операторов, когда каждая пользовательская DDL команда возбуждает DDL триггер, и этот триггер запускает другие DDL команды, с помощью EXECUTE STATEMENT, значения переменных в пространстве имён DDL_TRIGGER будут соответствовать команде, которая вызвала последний DDL триггер в стеке вызовов.

Переменные доступные в пространстве имён DDL_TRIGGER

- EVENT_TYPE – тип события (CREATE, ALTER, DROP)
- OBJECT_TYPE – тип объекта (TABLE, VIEW и д.р.)
- DDL_EVENT – имя события (<ddl event item>),
где <ddl event item> = EVENT_TYPE || ' ' || OBJECT_TYPE
- OBJECT_NAME – имя объекта метаданных
- OLD_OBJECT_NAME – имя объекта метаданных до переименования
- NEW_OBJECT_NAME – имя объекта метаданных после переименования
- SQL_TEXT – текст SQL запроса

Отключение триггеров

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.

Кто может создать триггеры на события изменения метаданных?

Триггеры на события изменения метаданных могут создать:

- Администраторы
- Владелец базы данных;
- Пользователи с привилегией ALTER DATABASE.

Примеры

Пример 156. Контроль наименования объектов базы данных с помощью DDL триггера

```

CREATE EXCEPTION e_invalid_sp_name
  'Неверное имя хранимой процедуры (должно начинаться с SP_)';

SET TERM !;

CREATE TRIGGER trig_ddl_sp BEFORE CREATE PROCEDURE
AS
BEGIN
  IF (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME')
      NOT STARTING 'SP_') THEN
    EXCEPTION e_invalid_sp_name;
END!

-- Test
CREATE PROCEDURE sp_test
AS
BEGIN
END!

CREATE PROCEDURE test
AS
BEGIN
END!

```

```

-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
-- -Неверное имя хранимой процедуры (должно начинаться с SP_)
-- -At trigger 'TRIG_DDL_SP' line: 4, col: 5

```

```
SET TERM ;!
```

Пример 157. Контроль безопасности DDL операторов

```
CREATE EXCEPTION e_access_denied 'Access denied';

SET TERM !;

CREATE TRIGGER trig_ddl BEFORE ANY DDL STATEMENT
AS
BEGIN
    IF (current_user <> 'SUPER_USER') THEN
        EXCEPTION e_access_denied;
END!

-- Test
CREATE PROCEDURE sp_test
AS
BEGIN
END!
```

```
-- The last command raises this exception and procedure SP_TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5
```

```
SET TERM ;!
```

В Firebird существуют привилегии на DDL операторы, поэтому прибегать к написанию DDL триггера нужно только в случае, если того же самого эффекта невозможно достичь стандартными методами.

Пример 158. Использование DDL триггеров для регистрации событий изменения метаданных

```

CREATE SEQUENCE ddl_seq;

CREATE TABLE ddl_log (
    id BIGINT NOT NULL PRIMARY KEY,
    moment TIMESTAMP NOT NULL,
    user_name VARCHAR(63) NOT NULL,
    event_type VARCHAR(25) NOT NULL,
    object_type VARCHAR(25) NOT NULL,
    ddl_event VARCHAR(25) NOT NULL,
    object_name VARCHAR(63) NOT NULL,
    old_object_name VARCHAR(63),
    new_object_name VARCHAR(63),
    sql_text BLOB sub_type text NOT NULL,
    ok CHAR(1) NOT NULL
);

SET TERM !;

CREATE TRIGGER trig_ddl_log_before BEFORE ANY DDL STATEMENT
AS
    DECLARE id TYPE OF COLUMN ddl_log.id;
BEGIN
    -- Мы должны производить изменения в AUTONOMOUS TRANSACTION,
    -- таким образом, если произойдёт исключение и команда
    -- не будет запущена, она всё равно будет зарегистрирована.
    IN AUTONOMOUS TRANSACTION DO
        BEGIN
            INSERT INTO ddl_log (
                id, moment, user_name, event_type, object_type, ddl_event,
                object_name, old_object_name, new_object_name, sql_text, ok)
            VALUES (NEXT VALUE FOR ddl_seq,
                    current_timestamp, current_user,
                    rdb$get_context('DDL_TRIGGER', 'EVENT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'DDL_EVENT'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME'),
                    rdb$get_context('DDL_TRIGGER', 'OLD_OBJECT_NAME'),
                    rdb$get_context('DDL_TRIGGER', 'NEW_OBJECT_NAME'),
                    rdb$get_context('DDL_TRIGGER', 'SQL_TEXT'),
                    'N')
            RETURNING id INTO id;
            rdb$set_context('USER_SESSION', 'trig_ddl_log_id', id);
        END
    END!
END!

-- Примечание:
-- созданный выше триггер будет запущен для этой DDL.
-- Хорошой идеей является использование -nodbtriggers

```

```
-- при работе с ним
CREATE TRIGGER trig_ddl_log_after AFTER ANY DDL STATEMENT
AS
BEGIN
    -- Здесь нам требуется автономная транзакция,
    -- потому что в оригинальной транзакции
    -- мы не увидим запись, вставленную в
    -- BEFORE триггере в автономной транзакции,
    -- если пользовательская транзакции не запущена
    -- с режимом изоляции READ COMMITTED.
    IN AUTONOMOUS TRANSACTION DO
        UPDATE ddl_log SET ok = 'Y'
        WHERE
            id = rdb$get_context('USER_SESSION', 'trig_ddl_log_id');
    END!

    COMMIT!

SET TERM ;!

-- Удаляем запись о создании trig_ddl_log_after.
DELETE FROM ddl_log;
COMMIT;

-- Тест

-- Эта команда будет зарегистрирована единожды
-- (т.к. T1 не существует, RECREATE вызовет событие CREATE)
-- с OK = Y.
RECREATE TABLE t1 (
    n1 INTEGER,
    n2 INTEGER
);

-- Оператор не выполнится, т.к. T1 уже существует,
-- таким образом OK будет иметь значение N.
CREATE TABLE t1 (
    n1 INTEGER,
    n2 INTEGER
);

-- T2 не существует. Это действие не будет зарегистрировано.
DROP TABLE t2;

-- Это действие будет зарегистрировано дважды
-- (т.к. T1 существует, действие RECREATE рассматривается
-- как DROP и CREATE) с полем OK = Y.
RECREATE TABLE t1 (
    n INTEGER
);
```

```
CREATE DOMAIN dom1 AS INTEGER;
```

```
ALTER DOMAIN dom1 TYPE BIGINT;
```

```
ALTER DOMAIN dom1 TO dom2;
```

```
COMMIT;
```

```
SELECT
```

```
    id,  
    ddl_event,  
    object_name as name,  
    sql_text,  
    ok  
FROM ddl_log  
ORDER BY id;
```

ID DDL_EVENT	OBJECT_NAME	SQL_TEXT	OK
2 CREATE TABLE	T1		80:3 Y
SQL_TEXT:			
recreate table t1 (
n1 integer,			
n2 integer			
)			
3 CREATE TABLE	T1		80:2 N
SQL_TEXT:			
create table t1 (
n1 integer,			
n2 integer			
)			
4 DROP TABLE	T1		80:6 Y
SQL_TEXT:			
recreate table t1 (
n integer			
)			
5 CREATE TABLE	T1		80:9 Y
SQL_TEXT:			
recreate table t1 (
n integer			
)			

См. также:

[ALTER TRIGGER](#), [DROP TRIGGER](#).

5.7.2. ALTER TRIGGER

Назначение

Изменение существующего триггера.

Доступно в

DSQL, ESQL

Синтаксис

```

ALTER TRIGGER trigname
[ACTIVE | INACTIVE]
[{BEFORE | AFTER} <mutation_list>]
[POSITION number]
[SQL SECURITY {DEFINER | INVOKER} | DROP SQL SECURITY]
[<routine body>]

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }

```

Полное описание оператора см. [CREATE TRIGGER](#).

Допустимые изменения

В операторе изменения триггера можно изменить:

- Состояние активности (ACTIVE | `INACTIVE);
- Фазу (BEFORE | AFTER);
- Событие(я);
- Позицию срабатывания;
- Привилегии выполнения триггера:зывающего пользователя (SQL SECURITY INVOKER), определяющего пользователя (SQL SECURITY DEFINER) или наследует у таблицы (DROP SQL SECURITY);
- Код тела триггера.

Если какой-либо элемент не указан, то он остаётся без изменений.



DML триггер невозможно изменить в триггер на событие базы данных и наоборот.

Событие в триггере базы данных невозможно изменить.

Помните

Триггер с ключевым словом BEFORE наступает до соответствующего события, с ключевым словом AFTER — после соответствующего события.



Один DML триггер может содержать более одного события (INSERT, UPDATE, DELETE). События должны быть разделены ключевым словом OR. Каждое из событий может быть указано не более одного раза.

Ключевое слово POSITION позволяет задать дополнительный порядок выполнения с одинаковыми фазой и событием. По умолчанию позиция равна 0. Если позиция не задана, или если несколько триггеров имеют один и тот же номер позиции, то триггеры будут выполнены в алфавитном порядке их наименований.

Кто может изменить триггеры?

DML триггеры могут изменить:

- Администраторы
- Владелец таблицы (представления);
- Пользователи с привилегией ALTER ANY {TABLE | VIEW}.

Триггеры для событий базы данных и триггеры событий на изменение метаданных могут изменить:

- Администраторы
- Владелец базы данных;
- Пользователь, имеющий привилегию ALTER DATABASE.

Примеры

Пример 159. Отключение (перевод в неактивное состояние) триггера

```
ALTER TRIGGER set_cust_no INACTIVE;
```

Пример 160. Изменение позиции триггера

```
ALTER TRIGGER set_cust_no POSITION 14;
```

Пример 161. Перевод триггера в неактивное состояние и изменение списка событий

```
ALTER TRIGGER TR_CUST_LOG
INACTIVE AFTER INSERT OR UPDATE;
```

Пример 162. Изменение привилегий выполнения триггера

После выполнения данного оператора триггер будет выполняться с привилегиями определяющего пользователя (владельца).

```
ALTER TRIGGER TR_CUST_LOG
SQL SECURITY DEFINER;
```

Пример 163. Удаление привилегий выполнения триггера

После удаления привилегий выполнения триггера, триггер выполняется с привилегиями унаследованными от таблицы. Если у таблицы не определены привилегии выполнения, то триггер будет выполнять с привилегиями вызывающего пользователя.

```
ALTER TRIGGER TR_CUST_LOG
DROP SQL SECURITY;
```

Пример 164. Перевод триггера в активное состояние, изменение его позиции и его тела

```
ALTER TRIGGER tr_log_connect
ACTIVE POSITION 1
AS
BEGIN
    INSERT INTO LOG_CONNECT (ID,
                             USERNAME,
                             ROLENAME,
                             ATIME)
    VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT,
            CURRENT_USER,
            CURRENT_ROLE,
            CURRENT_TIMESTAMP);
END
```

См. также:

[CREATE TRIGGER](#), [CREATE OR ALTER TRIGGER](#), [RECREATE TRIGGER](#).

5.7.3. CREATE OR ALTER TRIGGER

Назначение

Создание нового или изменение существующего триггера.

Доступно в

DSQL, ESQL

Синтаксис

```
CREATE OR ALTER TRIGGER trigname {
    <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger> }
<routine body>
```

Полное описание оператора см. [CREATE TRIGGER](#).

Оператор CREATE OR ALTER TRIGGER создаёт новый триггер, если он не существует, или изменяет и перекомпилирует его в противном случае, при этом существующие права и зависимости сохраняются.

Примеры

Пример 165. Создание нового или изменение существующего триггера

```
CREATE OR ALTER TRIGGER set_cust_no
ACTIVE BEFORE INSERT ON customer POSITION 0
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

См. также:

[CREATE TRIGGER](#), [ALTER TRIGGER](#), [RECREATE TRIGGER](#).

5.7.4. DROP TRIGGER

Назначение

Удаление существующего триггера.

Доступно в

DSQL, ESQL

Синтаксис

```
DROP TRIGGER trigname
```

Таблица 44. Параметры оператора DROP TRIGGER

Параметр	Описание
trigname` `	Имя триггера.

Оператор `DROP TRIGGER` удаляет существующий триггер.

Кто может удалить триггеры?

DML триггеры могут удалить:

- Администраторы
- Владелец таблицы (представления);
- Пользователи с привилегией `ALTER ANY {TABLE | VIEW}`.

Триггеры для событий базы данных и триггеры событий на изменение метаданных могут удалить:

- Администраторы
- Владелец базы данных;
- Пользователь, имеющий привилегию `ALTER DATABASE`.

Примеры*Пример 166. Удаление триггера*

```
DROP TRIGGER set_cust_no;
```

См. также:

[CREATE TRIGGER](#), [ALTER TRIGGER](#).

5.7.5. RECREATE TRIGGER*Назначение*

Создание нового или пересоздание существующего триггера.

Доступно в

DSQL, ESQL

Синтаксис

```
RECREATE TRIGGER trigname {
    <relation_trigger_legacy>
    | <relation_trigger_sql2003>
    | <database_trigger>
    | <ddl_trigger> }
<routine body>
```

Полное описание оператора см. [CREATE TRIGGER](#).

Оператор RECREATE TRIGGER создаёт новый триггер, если триггер с указанным именем не существует, в противном случае оператор RECREATE TRIGGER попытается удалить его и создать новый.

Примеры

Пример 167. Создание или пересоздание триггера

```
RECREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT ON customer POSITION 0
AS
BEGIN
    IF (NEW.cust_no IS NULL) THEN
        NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

См. также:

[CREATE TRIGGER](#), [DROP TRIGGER](#), [CREATE OR ALTER TRIGGER](#).

5.8. PROCEDURE

Хранимая процедура (ХП) — это программный модуль, который может быть вызван с клиента, из другой процедуры, функции, выполнимого блока (executable block) или триггера. Хранимые процедуры, хранимые функции, исполняемые блоки и триггеры 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.

5.8.1. CREATE PROCEDURE

Назначение

Создание новой хранимой функции.

Доступно в

DSQL, ESQL

Синтаксис

```

CREATE PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
<routine body>

<inparam> ::= <param_decl> [= | DEFAULT] <value>
<outparam> ::= <param_decl>
<value> ::= {literal | NULL | context_var}
<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]
<extname> ::= '<module name>!<routine name>[!<misc info>]'
<type> ::= <datatype> | [TYPE OF] domain_name | TYPE OF COLUMN rel.col
<datatype> ::=
  <scalar_datatype> | <blob_datatype> | <array_datatype>
<scalar_datatype> ::= См. Синтаксис скалярных типов данных
<blob_datatype> ::= См. Синтаксис типа данных BLOB
<array_datatype> ::= См. Синтаксис массивов
<routine body> ::=
  <SQL routine spec>
  | <external body reference>
<SQL routine spec> ::=
  [<rights clause>] <SQL routine body>
<rights clause> ::=
  SQL SECURITY {DEFINER | INVOKER}
<SQL routine body> ::=
  AS
  [<declarations>]
  BEGIN
  [<PSQL_statements>]
  END
<declarations> ::= <declare_item> [<declare_item> ...]
<declare_item> ::=

```

```

<declare_var>;
| <declare_cursor>;
| <routine declaration>;
| <routine implementation>

<routine declaration> ::= <subfunc_decl> | <subproc_decl>

<routine implementation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=
    EXTERNAL NAME '<extname>' ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

```

Таблица 45. Параметры оператора CREATE PROCEDURE

Параметр	Описание
procname	Имя хранимой процедуры. Может содержать до 63 символов.
inparam	Описание входного параметра.
outparam	Описание выходного параметра.
declarations	Секция объявления локальных переменных, именованных курсоров, подпроцедур и подфункций.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
subfunc_decl	Объявление подпрограммы – функции.
subproc_decl	Объявление подпрограммы – процедуры.
subfunc_impl	Реализация подпрограммы – функции.
subproc_impl	Реализация подпрограммы – процедуры.
literal	Литерал, совместимый по типу с параметром.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного или выходного параметра процедуры. Может содержать до 63 символов. Имя параметра должно быть уникальным среди входных и выходных параметров процедуры, а также её локальных переменных.
extbody	Тело внешней процедуры. Строковый литерал который может использоваться UDR для различных целей.
module name	Имя внешнего модуля, в котором расположена функция.
routine name	Внутреннее имя функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.

Параметр	Описание
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain_name	Имя домена.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.

Оператор `CREATE PROCEDURE` создаёт новую хранимую процедуру. Имя хранимой процедуры должно быть уникальным среди имён всех хранимых процедур, таблиц и представлений базы данных.



Желательно также, чтобы имя хранимой процедуры было уникальным и среди имён процедур расположенных в PSQL пакетах (package), хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноименная функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

`CREATE PROCEDURE` является составным оператором, состоящий из заголовка и тела.

Заголовок определяет имя хранимой процедуры и объявляет входные и выходные параметры, если они должны быть возвращены процедурой.

Тело процедуры состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешнем блоке, который начинается с ключевого слова `BEGIN`, и завершается ключевым словом `END`. Объявления локальных переменных и именованных курсоров, а также внутренние операторы должны завершаться точкой с запятой (;).

Терминатор оператора

Некоторые редакторы SQL-операторов — в частности утилита `isql`, которая идёт в комплекте с Firebird, и возможно некоторые сторонние редакторы — используют внутреннее соглашение, которое требует, чтобы все операторы были завершены с точкой с запятой.

Это создает конфликт с синтаксисом PSQL при кодировании в этих средах. Если вы не знакомы с этой проблемой и её решением, пожалуйста, изучите детали в главе PSQL в разделе, озаглавленном [Изменение терминатора в `isql`](#).

Параметры

У каждого параметра указывается тип данных. Кроме того, для параметра можно указать ограничение `NOT NULL`, тем самым запретив передавать в него значение `NULL`.

Для параметра строкового типа существует возможность задать порядок сортировки с помощью предложения COLLATE.

Входные параметры

Входные параметры заключаются в скобки после имени хранимой процедуры. Они передаются в процедуру по значению, то есть любые изменения входных параметров внутри процедуры никак не повлияет на значения этих параметров в вызывающей программе.

Входные параметры могут иметь значение по умолчанию. Параметры, для которых заданы значения, должны располагаться в конце списка параметров.

Выходные параметры

Необязательное предложение RETURNS позволяет задать список выходных параметров хранимой процедуры.

Использование доменов при объявлении параметров

В качестве типа параметра можно указать имя домена. В этом случае параметр будет наследовать все характеристики домена.

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.

Использование типа столбца при объявлении параметров

Входные и выходные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение TYPE OF COLUMN, после которого указывается имя таблицы или представления и через точку имя столбца.

При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё и набор символов, и порядок сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Привилегии выполнения

Необязательное предложение SQL SECURITY позволяет задать с какими привилегиями выполняется хранимая процедура. Если выбрана опция INVOKER, то хранимая процедура выполняется с привилегиями вызывающего пользователя. Если выбрана опция DEFINER, то хранимая процедура выполняется с привилегиями определяющего пользователя (владельца ХП). Эти привилегии будут дополнены привилегиями выданные самой хранимой процедуре с помощью оператора GRANT. По умолчанию хранимая процедура выполняется с привилегиями вызывающего пользователя.

Привилегии выполнения по умолчанию для вновь создаваемых объектов метаданных можно изменить с помощью оператора



```
ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

Тело хранимой процедуры

После ключевого слова AS следует тело хранимой процедуры.

Объявление локальных переменных, курсоров и подпрограмм

В необязательной секции declarations описаны локальные переменные процедуры, подпрограммы и именованные курсоры. В отношении спецификации типа данных локальные переменные подчиняются тем же правилам, что и входные и выходные параметры процедуры. Подробности вы можете посмотреть в главе “Процедурный язык PSQL” в разделах [DECLARE VARIABLE](#) и [DECLARE CURSOR](#), [DECLARE PROCEDURE](#), [DECLARE FUNCTION](#).

После необязательной секции деклараций обязательно следует составной оператор. Составной оператор состоит из одного или нескольких PSQL операторов, заключенных между ключевыми словами BEGIN и END. Составной оператор может содержать один или несколько других составных операторов. Вложенность ограничена 512 уровнями. Любой из BEGIN … END блоков может быть пустым, в том числе и главный блок.

Внешние хранимые процедуры

Хранимая процедура может быть расположена во внешнем модуле. В этом случае вместо тела процедуры указывается место её расположения во внешнем модуле с помощью предложения EXTERNAL NAME. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя процедуры внутри модуля и определённая пользователем информация. В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок UDR. После ключевого слова AS может быть указан строковый литерал — “тело” внешней процедуры, оно может быть использовано внешним модулем для различных целей.

Кто может создать хранимую процедуру?

Выполнить оператор CREATE PROCEDURE могут:

- Администраторы
- Пользователи с привилегией CREATE PROCEDURE.

Пользователь, создавший хранимую процедуру, становится её владельцем.

Примеры

Пример 168. Создание хранимой процедуры

```

CREATE PROCEDURE ADD_BREED (
    NAME D_BREEDNAME, /* Наследуются характеристики домена */
    NAME_EN TYPE OF D_BREEDNAME, /* Наследуется только тип домена */
    SHORTNAME TYPE OF COLUMN BREED.SHORTNAME, /* Наследуется тип столбца таблицы */
    REMARK VARCHAR(120) CHARACTER SET WIN1251 COLLATE PXW_CYRL,
    CODE_ANIMAL INT NOT NULL DEFAULT 1
)
RETURNS (
    CODE_BREED INT
)
AS
BEGIN
    INSERT INTO BREED (
        CODE_ANIMAL, NAME, NAME_EN, SHORTNAME, REMARK)
    VALUES (
        :CODE_ANIMAL, :NAME, :NAME_EN, :SHORTNAME, :REMARK)
    RETURNING CODE_BREED INTO CODE_BREED;
END

```

То же самое, но процедура будет выполняться с правами определяющего пользователя (владельца процедуры).

```

CREATE PROCEDURE ADD_BREED (
    NAME D_BREEDNAME, /* Наследуются характеристики домена */
    NAME_EN TYPE OF D_BREEDNAME, /* Наследуется только тип домена */
    SHORTNAME TYPE OF COLUMN BREED.SHORTNAME, /* Наследуется тип столбца таблицы */
    REMARK VARCHAR(120) CHARACTER SET WIN1251 COLLATE PXW_CYRL,
    CODE_ANIMAL INT NOT NULL DEFAULT 1
)
RETURNS (
    CODE_BREED INT
)
SQL SECURITY DEFINER
AS
BEGIN
    INSERT INTO BREED (
        CODE_ANIMAL, NAME, NAME_EN, SHORTNAME, REMARK)
    VALUES (
        :CODE_ANIMAL, :NAME, :NAME_EN, :SHORTNAME, :REMARK)
    RETURNING CODE_BREED INTO CODE_BREED;
END

```

Пример 169. Создание внешней хранимой процедуры

Создание процедуры находящейся во внешнем модуле (UDR). Реализация процедуры расположена во внешнем модуле `udrcpp_example`. Имя процедуры внутри модуля — `gen_rows`.

```
CREATE PROCEDURE gen_rows (
    start_n INTEGER NOT NULL,
    end_n INTEGER NOT NULL
) RETURNS (
    n INTEGER NOT NULL
)
    EXTERNAL NAME 'udrcpp_example!gen_rows'
    ENGINE udr;
```

См. также:

[CREATE OR ALTER PROCEDURE](#), [ALTER PROCEDURE](#), [RECREATE PROCEDURE](#), [DROP PROCEDURE](#).

5.8.2. ALTER PROCEDURE

Назначение

Изменение существующей хранимой процедуры.

Доступно в

DSQL, ESQL

Синтаксис

```
ALTER PROCEDURE прогнаме [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
<routine body>
```

Подробнее см. [CREATE PROCEDURE](#).

Оператор `ALTER PROCEDURE` позволяет изменять состав и характеристики входных и выходных параметров, локальных переменных, именованных курсоров и тело хранимой процедуры. Для внешних процедур (UDR) вы можете изменить точку входа и имя движка. После выполнения существующие привилегии и зависимости сохраняются.



Будьте осторожны при изменении количества и типов входных и выходных параметров хранимых процедур. Существующий код приложения может стать неработоспособным из-за того, что формат вызова процедуры несовместим с новым описанием параметров. Кроме того, PSQL модули, использующие изменённую хранимую процедуру, могут стать некорректными. Информация о том, как это обнаружить, находится в приложении [Поле RDB\\$VALID_BLR](#).

Кто может изменить хранимую процедуру?

Выполнить оператор ALTER PROCEDURE могут:

- Администраторы
- Владелец хранимой процедуры;
- Пользователи с привилегией ALTER ANY PROCEDURE.

Примеры

Пример 170. Изменение хранимой процедуры

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

См. также:

[CREATE PROCEDURE](#), [CREATE OR ALTER PROCEDURE](#), [RECREATE PROCEDURE](#), [DROP PROCEDURE](#).

5.8.3. CREATE OR ALTER PROCEDURE

Назначение

Создание новой или изменение существующей хранимой процедуры.

Доступно в

DSQL, ESQL

Синтаксис

```
CREATE OR ALTER PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
<routine body>
```

Подробнее см. [CREATE PROCEDURE](#).

Оператор CREATE OR ALTER PROCEDURE создаёт новую или изменяет существующую хранимую процедуру. Если хранимая процедура не существует, то она будет создана с использованием предложения CREATE PROCEDURE. Если она уже существует, то она будет изменена и откомпилирована, при этом существующие привилегии и зависимости сохраняются.

Примеры

Пример 171. Создание или изменение хранимой процедуры

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

См. также:

[CREATE PROCEDURE](#), [ALTER PROCEDURE](#), [RECREATE PROCEDURE](#), [DROP PROCEDURE](#).

5.8.4. DROP PROCEDURE

Назначение

Удаление существующей хранимой процедуры.

Доступно в

DSQL, ESQL

Синтаксис

```
DROP PROCEDURE procname
```

Таблица 46. Параметры оператора DROP PROCEDURE

Параметр	Описание
<i>procname</i>	Имя хранимой процедуры.

Оператор DROP PROCEDURE удаляет существующую хранимую процедуру. Если от хранимой процедуры существуют зависимости, то при попытке удаления такой процедуру будет выдана соответствующая ошибка.

Кто может удалить хранимую процедуру?

Выполнить оператор DROP PROCEDURE могут:

- Администраторы
- Владелец хранимой процедуры;
- Пользователи с привилегией DROP ANY PROCEDURE.

Примеры

Пример 172. Удаление хранимой процедуры

```
DROP PROCEDURE GET_EMP_PROJ;
```

См. также:

CREATE PROCEDURE, RECREATE PROCEDURE.

5.8.5. RECREATE PROCEDURE*Назначение*

Создание новой или пересоздание существующей хранимой процедуры.

Доступно в

DSQL, ESQL

Синтаксис

```
RECREATE PROCEDURE procname [(<inparam> [, <inparam> ...])]  
RETURNS (<outparam> [, <outparam> ...])  
<routine body>
```

Подробнее см. CREATE PROCEDURE.

Оператор RECREATE PROCEDURE создаёт новую или пересоздаёт существующую хранимую процедуру. Если процедура с таким именем уже существует, то оператор попытается удалить её и создать новую процедуру. Операция закончится неудачей при подтверждении транзакции, если процедура имеет зависимости.



Имейте в виду, что ошибки зависимостей не обнаруживаются до фазы подтверждения транзакции.

После пересоздания процедуры привилегии на выполнение хранимой процедуры и привилегии самой хранимой процедуры не сохраняются.

Примеры

Пример 173. Создание новой или пересоздание существующей хранимой процедуры

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

См. также:

[CREATE PROCEDURE](#), [CREATE OR ALTER PROCEDURE](#), [DROP PROCEDURE](#).

5.9. FUNCTION

Хранимая функция является программой, хранящейся в области метаданных базы данных и выполняющейся на стороне сервера. К хранимой функции могут обращаться хранимые процедуры, хранимые функции (в том числе и сама к себе), триггеры и клиентские программы. При обращении хранимой функции самой к себе такая хранимая функция называется рекурсивной.

В отличие от хранимых процедур хранимые функции всегда возвращают одно скалярное значение. Для возврата значения из хранимой функции используется оператор RETURN, который немедленно прекращает выполнение функции.

5.9.1. CREATE FUNCTION

Назначение

Создание новой хранимой функции.

Доступно в

DSQL

Синтаксис

```

CREATE FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation]
[DETERMINISTIC]
<routine body>

<inparam> ::= <param_decl> [= | DEFAULT] <value>
<value> ::= {literal | NULL | context_var}

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

<type> ::= <datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col

<datatype> ::=
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов

<routine body> ::=
    <SQL routine spec>
    | <external body reference>

<SQL routine spec> ::=
    [<rights clause>] <SQL routine body>

<rights clause> ::=
    SQL SECURITY {DEFINER | INVOKER}

<SQL routine body> ::=
    AS
    [<declarations>]
    BEGIN

```

```
[<PSQL_statements>]
END

<declarations> ::= <declare_item> [<declare_item> ...]

<declare_item> ::=
  <declare_var>;
| <declare_cursor>;
| <subroutine declaration>;
| <subroutine implementation>

<subroutine declaration> ::= <subfunc_decl> | <subproc_decl>

<subroutine implementation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=
  EXTERNAL NAME '<extname>' ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

Таблица 47. Параметры оператора *CREATE FUNCTION*

Параметр	Описание
funcname	Имя хранимой функции. Может содержать до 63 символов.
inparam	Описание входного параметра.
declarations	Секция объявления локальных переменных, именованных курсоров, подпроцедур и подфункций.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
subfunc_decl	Объявление подпрограммы – функции.
subproc_decl	Объявление подпрограммы – процедуры.
subfunc_impl	Реализация подпрограммы – функции.
subproc_impl	Реализация подпрограммы – процедуры.
literal	Литерал, совместимый по типу с параметром.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного параметра функции. Может содержать до 63 символов. Имя параметра должно быть уникальным среди входных параметров функции, а также её локальных переменных.
module name	Имя внешнего модуля, в котором расположена функция.
routine name	Внутреннее имя функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.

Параметр	Описание
extbody	Тело внешней функции. Строковый литерал который может использоваться UDR для различных целей.
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain_name	Имя домена.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.

Оператор `CREATE FUNCTION` создаёт новую хранимую функцию. Имя хранимой функции должно быть уникальным среди имён всех хранимых функций и внешних (UDF) функций. Если только это не внутренняя функция (“подпрограмма”). Для внутренних функций достаточно уникальности только в рамках модулей, которые их “охватывают”.



Желательно также, чтобы имя хранимой функции было уникальным и среди имён функций расположенных в PSQL пакетах (package), хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноименная функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

`CREATE FUNCTION` является составным оператором, состоящий из заголовка и тела. Заголовок определяет имя хранимой функции, объявляет входные параметры и тип возвращаемого значения.

Тело функции состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешнем блоке, который начинается с ключевого слова `BEGIN`, и завершается ключевым словом `END`. Объявления локальных переменных и именованных курсоров, а также внутренние операторы должны завершаться точкой с запятой (;).

Терминатор оператора

Некоторые редакторы SQL-операторов — в частности утилита `isql`, которая идёт в комплекте с Firebird, и возможно некоторые сторонние редакторы — используют внутреннее соглашение, которое требует, чтобы все операторы были завершены с точкой с запятой.

Это создает конфликт с синтаксисом PSQL при кодировании в этих средах. Если вы не знакомы с этой проблемой и её решением, пожалуйста, изучите детали в главе PSQL в разделе, озаглавленном [Изменение терминатора в `isql`](#).

Входные параметры

Входные параметры заключаются в скобки после имени хранимой функции. Они передаются в функцию по значению, то есть любые изменения входных параметров внутри функции никак не повлияет на значения этих параметров в вызывающей программе.

У каждого параметра указывается тип данных. Кроме того, для параметра можно указать ограничение NOT NULL, тем самым запретив передавать в него значение NULL.

Для параметра строкового типа существует возможность задать порядок сортировки с помощью предложения COLLATE.

Входные параметры могут иметь значение по умолчанию. Параметры, для которых заданы значения, должны располагаться в конце списка параметров.

Использование доменов при объявлении параметров

В качестве типа параметра можно указать имя домена. В этом случае параметр будет наследовать все характеристики домена.

Если перед назначением 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.

Использование типа столбца при объявлении параметров

Входные и выходные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение TYPE OF COLUMN, после которого указывается имя таблицы или представления и через точку имя столбца.

При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё и набор символов, и порядок сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Возвращаемое значение

Предложение RETURNS задаёт тип возвращаемого значения хранимой функции. Если функция возвращает значение строкового типа, то существует возможность задать порядок сортировки с помощью предложения COLLATE. В качестве типа выходного значения можно указать имя домена, ссылку на его тип (с помощью предложения TYPE OF) или ссылку на тип столбца таблицы (с помощью предложения TYPE OF COLUMN).

Детерминированные функции

Необязательное предложение DETERMINISTIC указывает, что функция детерминированная. Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот

же набор входных значений. Если для функции указано, что она является детерминированной, то такая функция не вычисляется заново, если она уже была вычислена однажды с данным набором входных аргументов, а берет свои значения из кэша метаданных (если они там есть).

На самом деле в текущей версии Firebird, не существует кэша хранимых функций с маппингом входных аргументов на выходные значения.

Указание инструкции DETERMINISTIC на самом деле нечто вроде “обещания”, что код функции будет возвращать одно и то же. В данный момент детерминистическая функция считается инвариантом и работает по тем же принципам, что и другие инварианты. Т.е. вычисляется и кэшируется на уровне текущего выполнения данного запроса.

Это легко демонстрируется таким примером:



```

CREATE FUNCTION FN_T
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
    RETURN rand();
END

-- функция будет вычислена дважды и вернёт 2 разных значения
SELECT fn_t() FROM rdb$database
UNION ALL
SELECT fn_t() FROM rdb$database

-- функция будет вычислена единожды и вернёт 2 одинаковых значения
WITH t(n) AS (
    SELECT 1 FROM rdb$database
    UNION ALL
    SELECT 2 FROM rdb$database
)
SELECT n, fn_t() FROM t

```

Привилегии выполнения

Необязательное предложение SQL SECURITY позволяет задать с какими привилегиями выполняется хранимая функция. Если выбрана опция INVOKER, то хранимая функция выполняется с привилегиями вызывающего пользователя. Если выбрана опция DEFINER, то хранимая функция выполняется с привилегиями определяющего пользователя (владельца функции). Эти привилегии будут дополнены привилегиями выданные самой хранимой функции с помощью оператора GRANT. По умолчанию хранимая функция выполняется с привилегиями вызывающего пользователя.

Привилегии выполнения по умолчанию для вновь создаваемых объектов метаданных можно изменить с помощью оператора



```
ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

Тело хранимой функции

После ключевого слова AS следует тело хранимой функции.

Объявление локальных переменных, курсоров и подпрограмм

В необязательной секции declarations описаны локальные переменные функции, именованные курсоры и подпрограммы (подпроцедуры и подфункции). Локальные переменные подчиняются тем же правилам, что и входные параметры функции в отношении спецификации типа данных. Подробности вы можете посмотреть в главе “Процедурный язык PSQL” в разделах [DECLARE VARIABLE](#) и [DECLARE CURSOR, DECLARE PROCEDURE, DECLARE FUNCTION](#).

После необязательной секции деклараций обязательно следует составной оператор. Составной оператор состоит из одного или нескольких PSQL операторов, заключенных между ключевыми словами BEGIN и END. Составной оператор может содержать один или несколько других составных операторов. Вложенность ограничена 512 уровнями. Любой из BEGIN … END блоков может быть пустым, в том числе и главный блок.

Внешние функции

Хранимая функция может быть расположена во внешнем модуле. В этом случае вместо тела функции указывается место расположения функции во внешнем модуле с помощью предложения EXTERNAL NAME. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя функции внутри модуля и определённая пользователем информация. В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок UDR. После ключевого слова AS может быть указан строковый литерал — “тело” внешней функции, оно может быть использовано внешним модулем для различных целей.



Не следует путать внешние функции, объявленные как `DECLARE EXTERNAL FUNCTION`, так же известные как UDF, с функциями расположенными во внешних модулях объявленных как `CREATE FUNCTION ... EXTERNAL NAME`, называемых UDR (User Defined Routine). Первые являются унаследованными (Legacy) из предыдущих версий Firebird. Их возможности существенно уступают возможностям нового типа внешних функций. В Firebird 4.0 UDF объявлены устаревшими.

Кто может создать функцию?

Выполнить оператор `CREATE FUNCTION` могут:

- Администраторы
- Пользователи с привилегией CREATE FUNCTION.

Пользователь, создавший хранимую функцию, становится её владельцем.

Примеры

Пример 174. Создание хранимой функции

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A+B;
END
```

Вызов в запросе:

```
SELECT ADD_INT(2, 3) AS R FROM RDB$DATABASE
```

Вызов внутри PSQL кода, второй необязательный параметр не указан:

```
MY_VAR = ADD_INT(A);
```

Пример 175. Создание детерминистической хранимой функции

```
CREATE FUNCTION FN_E()
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
    RETURN EXP(1);
END
```

Пример 176. Создание хранимой функции с параметрами типа столбца таблицы

Функция, возвращающая имя мнемоники по имени столбца и значения мнемоники.

```
CREATE FUNCTION GET_MNEMONIC (
    AFIELD_NAME TYPE OF COLUMN RDB$TYPES.RDB$FIELD_NAME,
    ATYPE TYPE OF COLUMN RDB$TYPES.RDB$TYPE)
RETURNS TYPE OF COLUMN RDB$TYPES.RDB$TYPE_NAME
AS
BEGIN
    RETURN (SELECT RDB$TYPE_NAME
            FROM RDB$TYPES
           WHERE RDB$FIELD_NAME = :AFIELD_NAME
             AND RDB$TYPE = :ATYPE);
END
```

То же самое, но хранимая функция будет выполняться с привилегиями определяющего пользователя (владельца функции).

```
CREATE FUNCTION GET_MNEMONIC (
    AFIELD_NAME TYPE OF COLUMN RDB$TYPES.RDB$FIELD_NAME,
    ATYPE TYPE OF COLUMN RDB$TYPES.RDB$TYPE)
RETURNS TYPE OF COLUMN RDB$TYPES.RDB$TYPE_NAME
SQL SECURITY DEFINER
AS
BEGIN
    RETURN (SELECT RDB$TYPE_NAME
            FROM RDB$TYPES
           WHERE RDB$FIELD_NAME = :AFIELD_NAME
             AND RDB$TYPE = :ATYPE);
END
```

Пример 177. Создание внешней хранимой функции

Создание функции находящейся во внешнем модуле (UDR). Реализация функции расположена во внешнем модуле `udrcpp_example`. Имя функции внутри модуля — `wait_event`.

```
CREATE FUNCTION wait_event (
    event_name varchar(63) CHARACTER SET ascii
) RETURNS INTEGER
EXTERNAL NAME 'udrcpp_example!wait_event'
ENGINE udr
```

Пример 178. Создание хранимой функции содержащую подфункцию

Создание функции для перевода числа в 16-ричный формат.

```

CREATE FUNCTION INT_TO_HEX (
    ANumber BIGINT,
    AByte_Per_Number SMALLINT = 8)
RETURNS CHAR(66)
AS
DECLARE VARIABLE xMod SMALLINT;
DECLARE VARIABLE xResult VARCHAR(64);
DECLARE FUNCTION TO_HEX(ANum SMALLINT) RETURNS CHAR
AS
BEGIN
    RETURN CASE ANum
        WHEN 0 THEN '0'
        WHEN 1 THEN '1'
        WHEN 2 THEN '2'
        WHEN 3 THEN '3'
        WHEN 4 THEN '4'
        WHEN 5 THEN '5'
        WHEN 6 THEN '6'
        WHEN 7 THEN '7'
        WHEN 8 THEN '8'
        WHEN 9 THEN '9'
        WHEN 10 THEN 'A'
        WHEN 11 THEN 'B'
        WHEN 12 THEN 'C'
        WHEN 13 THEN 'D'
        WHEN 14 THEN 'E'
        WHEN 15 THEN 'F'
        ELSE NULL
    END;
END
BEGIN
    xMod = MOD(ANumber, 16);
    ANumber = ANumber / 16;
    xResult = TO_HEX(xMod);
    WHILE (ANUMBER > 0) DO
        BEGIN
            xMod = MOD(ANumber, 16);
            ANumber = ANumber / 16;
            xResult = TO_HEX(xMod) || xResult;
        END
    RETURN '0x' || LPAD(xResult, AByte_Per_Number * 2, '0');
END

```

См. также:

CREATE OR ALTER FUNCTION, ALTER FUNCTION, RECREATE FUNCTION, DROP FUNCTION.**5.9.2. ALTER FUNCTION***Назначение*

Изменение существующей хранимой функции.

Доступно в

DSQL

Синтаксис

```
ALTER FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation]
[DETERMINISTIC]
<routine body>
```

Подробнее см. [CREATE FUNCTION](#).

Оператор ALTER FUNCTION позволяет изменять состав и характеристики входных параметров, типа выходного значения, локальных переменных, именованных курсоров, подпрограмм и тело хранимой функции. Для внешних функций (UDR) вы можете изменить точку входа и имя движка. Внешние функции, объявленные как DECLARE EXTERNAL FUNCTION, так же известные как UDF, невозможно преобразовать в PSQL функции и наоборот. После выполнения существующие привилегии и зависимости сохраняются.

 Будьте осторожны при изменении количества и типов входных параметров хранимых функций. Существующий код приложения может стать неработоспособным из-за того, что формат вызова функции несовместим с новым описанием параметров. Кроме того, PSQL модули, использующие изменённую хранимую функцию, могут стать некорректными. Информация о том, как это обнаружить, находится в приложении [Поле RDB\\$VALID_BLR](#).

 Если у вас уже есть внешняя функция в Legacy стиле (DECLARE EXTERNAL FUNCTION), то оператор ALTER FUNCTION изменит её на обычную функцию без всяких предупреждений. Это было сделано умышлено для облегчения миграции на новый стиль написания внешних функций известных как UDR.

Кто может изменить функцию?

Выполнить оператор ALTER FUNCTION могут:

- Администраторы
- Владелец хранимой функции;
- Пользователи с привилегией ALTER ANY FUNCTION.

Примеры

Пример 179. Изменение хранимой функции

```
ALTER FUNCTION ADD_INT(A INT, B INT, C INT)
RETURNS INT
AS
BEGIN
    RETURN A+B+C;
END
```

См. также:

[CREATE FUNCTION](#), [CREATE OR ALTER FUNCTION](#), [DROP FUNCTION](#).

5.9.3. CREATE OR ALTER FUNCTION

Назначение

Создание новой или изменение существующей хранимой функции.

Доступно в

DSQL

Синтаксис

```
CREATE OR ALTER FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation]
[DETERMINISTIC]
<routine body>
```

Подробнее см. [CREATE FUNCTION](#).

Оператор CREATE OR ALTER FUNCTION создаёт новую или изменяет существующую хранимую функцию. Если хранимая функция не существует, то она будет создана с использованием предложения CREATE FUNCTION. Если она уже существует, то она будет изменена и перекомпилирована, при этом существующие привилегии и зависимости сохраняются.



Если у вас уже есть внешняя функция в Legacy стиле (DECLARE EXTERNAL FUNCTION), то оператор CREATE OR ALTER FUNCTION изменит её на обычную функцию без всяких предупреждений. Это было сделано умышлено для облегчения миграции на новый стиль написания внешних функций известных как UDR.

Примеры

Пример 180. Создание новой или изменение существующей хранимой функции

```
CREATE OR ALTER FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A+B;
END
```

См. также:

[CREATE FUNCTION, ALTER FUNCTION.](#)

5.9.4. DROP FUNCTION

Назначение

Удаление хранимой функции.

Доступно в

DSQL

Синтаксис

```
DROP FUNCTION funcname
```

Таблица 48. Параметры оператора DROP FUNCTION

Параметр	Описание
funcname	Имя хранимой функции.

Оператор `DROP FUNCTION` удаляет существующую хранимую функцию. Если от хранимой функции существуют зависимости, то при попытке удаления такой функции будет выдана соответствующая ошибка.

Кто может удалить функцию?

Выполнить оператор `DROP FUNCTION` могут:

- Администраторы
- Владелец хранимой функции;
- Пользователи с привилегией `DROP ANY FUNCTION`.

Примеры

Пример 181. Удаление хранимой функции

```
DROP FUNCTION ADD_INT;
```

См. также:

[CREATE FUNCTION](#).

5.9.5. RECREATE FUNCTION

Назначение

Создание новой или пересоздание существующей хранимой функции.

Доступно в

DSQL

Синтаксис

```
RECREATE FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation]
[DETERMINISTIC]
<routine body>
```

Подробнее см. [CREATE FUNCTION](#)

Оператор RECREATE FUNCTION создаёт новую или пересоздаёт существующую хранимую функцию. Если функция с таким именем уже существует, то оператор попытается удалить её и создать новую функцию. Операция закончится неудачей при подтверждении транзакции, если функция имеет зависимости.



Имейте в виду, что ошибки зависимостей не обнаруживаются до фазы подтверждения транзакции.

После пересоздания функции привилегии на выполнение хранимой функции и привилегии самой хранимой функции не сохраняются.

Примеры

Пример 182. Создание или пересоздание хранимой функции

```
RECREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A+B;
END
```

См. также:

[CREATE FUNCTION](#), [DROP FUNCTION](#).

5.10. PACKAGE

Пакет — группа процедур и функций, которая представляет собой один объект базы данных.

Пакеты Firebird состоят из двух частей: заголовка (ключевое слово PACKAGE) и тела (ключевые слова PACKAGE BODY). Такое разделение очень сильно напоминает модули Delphi, заголовок соответствует интерфейсной части, а тело — части реализации.

Сначала создаётся заголовок (CREATE PACKAGE), а затем — тело ([CREATE PACKAGE BODY](#)).

5.10.1. CREATE PACKAGE

Назначение

Создание заголовка пакета.

Доступно в

DSQL

Синтаксис

```

CREATE PACKAGE package_name
[<rights clause>]
AS
BEGIN
    [<package_item> ...]
END

<rights clause> ::= 
    SQL SECURITY {DEFINER | INVOKER}

<package_item> ::= 
    <function_decl>;
    | <procedure_decl>;

<function_decl> ::= 
    FUNCTION func_name [(<in_params>)]
    RETURNS <type> [COLLATE collation]
    [DETERMINISTIC]

<procedure_decl> ::= 
    PROCEDURE proc_name [(<in_params>)]
    [RETURNS (<out_params>)]

<in_params> ::= <inparam> [, <inparam> ...]

<inparam> ::= <param_decl> [= | DEFAULT] <value>

<value> ::= {literal | NULL | context_var}

<out_params> ::= <outparam> [, <outparam> ...]

<outparam> ::= <param_decl>

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<type> ::= <datatype> | [TYPE OF] domain_name | TYPE OF COLUMN rel.col

<datatype> ::= 
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов

```

Таблица 49. Параметры оператора CREATE PACKAGE

Параметр	Описание
package_name	Имя пакета. Может содержать до 63 символов.
function_decl	Объявление функции.
procedure_decl	Объявление процедуры.
proc_name	Имя процедуры. Может содержать до 63 символов.
func_name	Имя функции. Может содержать до 63 символов.
inparam	Описание входного параметра.
outparam	Описание выходного параметра.
literal	Литерал, совместимый по типу с параметром.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного или выходного параметра процедуры/функции. Может содержать до 63 символов. Имя параметра должно быть уникальным среди входных и выходных параметров процедуры/функции, а также её локальных переменных.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain_name	Имя домена.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.

Оператор CREATE PACKAGE создаёт новый заголовок пакета. Имя пакета должно быть уникальным среди имён всех пакетов.

Процедуры и функции, объявленные в заголовке пакета, доступны вне тела пакета через полный идентификатор имён процедур и функций (`package_name.procedure_name` и `package_name.function_name`). Процедуры и функции, определенные в теле пакета, но не объявленные в заголовке пакета, не видны вне тела пакета.

Имена процедур и функций, объявленные в заголовке пакета, должны быть уникальны среди имён процедур и функций, объявленных в заголовке и теле пакета.



Желательно чтобы имена хранимых процедур и функций пакета не пересекались с именами хранимых процедур и функций из глобального пространства имен, хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноименная функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

Привилегии выполнения

Необязательное предложение SQL SECURITY позволяет задать с какими привилегиями выполняется процедуры и функции пакета. Если выбрана опция INVOKER, то процедуры и функции пакета выполняются с привилегиями вызывающего пользователя. Если выбрана опция DEFINER, то процедуры и функции пакета выполняются с привилегиями определяющего пользователя (владельца пакета). Эти привилегии будут дополнены привилегиями выданные самому пакету с помощью оператора GRANT. По умолчанию процедуры и функции пакета выполняются с привилегиями вызывающего пользователя. Переопределять привилегии выполнения для процедур и функций пакета запрещено.

Привилегии выполнения по умолчанию для вновь создаваемых объектов метаданных можно изменить с помощью оператора



```
ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

Терминатор оператора

Некоторые редакторы SQL-операторов — в частности утилита `iSQL`, которая идёт в комплекте с Firebird, и возможно некоторые сторонние редакторы — используют внутреннее соглашение, которое требует, чтобы все операторы были завершены с точкой с запятой.

Это создает конфликт с синтаксисом PSQL при кодировании в этих средах. Если вы не знакомы с этой проблемой и её решением, пожалуйста, изучите детали в главе PSQL в разделе, озаглавленном [Изменение терминатора в `iSQL`](#).

Параметры процедур и функций

У каждого параметра указывается тип данных. Кроме того, для параметра можно указать ограничение NOT NULL, тем самым запретив передавать в него значение NULL.

Для параметра строкового типа существует возможность задать порядок сортировки с помощью предложения COLLATE.

Входные параметры

Входные параметры заключаются в скобки после имени хранимой процедуры. Они передаются в процедуру по значению, то есть любые изменения входных параметров внутри процедуры никак не повлияет на значения этих параметров в вызывающей программе.

Входные параметры могут иметь значение по умолчанию. Параметры, для которых заданы значения, должны располагаться в конце списка параметров.

Выходные параметры

Для хранимых процедур список выходных параметров задаётся в необязательное предложение RETURNS.

Для хранимых функций в обязательном предложении RETURNS задаётся тип возвращаемого значения.

Использование доменов при объявлении параметров

В качестве типа параметра можно указать имя домена. В этом случае параметр будет наследовать все характеристики домена.

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.

Использование типа столбца при объявлении параметров

Входные и выходные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение TYPE OF COLUMN, после которого указывается имя таблицы или представления и через точку имя столбца.

При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё и набор символов, и порядок сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Детерминированные функции

Необязательное предложение DETERMINISTIC в объявлении функции указывает, что функция детерминированная. Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений. Если для функции указано, что она является детерминированной, то такая функция не вычисляется заново, если она уже была вычислена однажды с данным набором входных аргументов, а берет свои значения из кэша метаданных (если они там есть).

На самом деле в текущей версии Firebird, не существует кэша хранимых функций с маппингом входных аргументов на выходные значения.

 Указание инструкции DETERMINISTIC на самом деле нечто вроде “обещания”, что код функции будет возвращать одно и то же. В данный момент детерминистическая функция считается инвариантом и работает по тем же принципам, что и другие инварианты. Т.е. вычисляется и кэшируется на уровне текущего выполнения данного запроса.

Кто может создать пакет?

Выполнить оператор CREATE PACKAGE могут:

- Администраторы

- Пользователи с привилегией CREATE PACKAGE.

Пользователь, создавший заголовок пакета становится владельцем пакета.

Примеры

Пример 183. Создание заголовка пакета

```
CREATE PACKAGE APP_VAR
AS
BEGIN
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

То же самое, но процедуры и функции пакета будут выполняться с правами определяющего пользователя (владельца пакета).

```
CREATE PACKAGE APP_VAR
SQL SECURITY DEFINER
AS
BEGIN
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

См. также:

[CREATE PACKAGE BODY](#), [ALTER PACKAGE](#), [DROP PACKAGE](#).

5.10.2. ALTER PACKAGE

Назначение

Изменение заголовка пакета.

Доступно в

DSQL

Синтаксис

```

ALTER PACKAGE package_name
[<rights clause>]
AS
BEGIN
  [<package_item> ...]
END

<package_item> ::= 
  <function_decl>;
| <procedure_decl>;

<function_decl> ::= 
  FUNCTION func_name [(<in_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]

<procedure_decl> ::= 
  PROCEDURE proc_name [(<in_params>)]
  [RETURNS (<out_params>)]

```

Подробнее см. [CREATE PACKAGE](#)

Оператор ALTER PACKAGE изменяет заголовок пакета. Позволяется изменять количество и состав процедур и функций, их входных и выходных параметров. При этом исходный код тела пакета сохраняется. Состояние соответствия тела пакета его заголовку отображается в столбце RDB\$PACKAGES.RDB\$VALID_BODY_FLAG.

Кто может изменить заголовок пакета?

Выполнить оператор ALTER PACKAGE могут:

- Администраторы
- Владелец пакета;
- Пользователи с привилегией ALTER ANY PACKAGE.

Примеры

Пример 184. Изменение заголовка пакета

```
ALTER PACKAGE APP_VAR
AS
BEGIN
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

См. также:

[CREATE PACKAGE](#), [DROP PACKAGE](#), [RECREATE PACKAGE BODY](#).

5.10.3. CREATE OR ALTER PACKAGE

Назначение

Создание нового или изменение существующего заголовка пакета.

Доступно в

DSQL

Синтаксис

```
CREATE OR ALTER PACKAGE package_name
[<rights clause>]
AS
BEGIN
    [<package_item> ...]
END

<package_item> ::= 
    <function_decl>;
| <procedure_decl>

<function_decl> ::= 
    FUNCTION func_name [(<in_params>)]
    RETURNS <type> [COLLATE collation]
    [DETERMINISTIC]

<procedure_decl> ::= 
    PROCEDURE proc_name [(<in_params>)]
    [RETURNS (<out_params>)]
```

Подробнее см. [CREATE PACKAGE](#)

Оператор CREATE OR ALTER PACKAGE создаёт новый или изменяет существующий заголовок пакета. Если заголовок пакета не существует, то он будет создан с использованием

предложения `CREATE PACKAGE`. Если он уже существует, то он будет изменен и перекомпилирован, при этом существующие привилегии и зависимости сохраняются.

Примеры

Пример 185. Создание нового или изменение существующего заголовка пакета

```
CREATE OR ALTER PACKAGE APP_VAR
AS
BEGIN
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

См. также:

[CREATE PACKAGE](#), [ALTER PACKAGE](#), [RECREATE PACKAGE BODY](#).

5.10.4. DROP PACKAGE

Назначение

Удаление заголовка пакета.

Доступно в

DSQL

Синтаксис

```
DROP PACKAGE package_name
```

Таблица 50. Параметры оператора `DROP PACKAGE`

Параметр	Описание
package_name	Имя пакета.

Оператор `DROP PACKAGE` удаляет существующий заголовок пакета. Перед удалением заголовка пакета (`DROP PACKAGE`), необходимо выполнить удаление тела пакета (`DROP PACKAGE BODY`), иначе будет выдана ошибка. Если от заголовка пакета существуют зависимости, то при попытке удаления такого заголовка будет выдана соответствующая ошибка.

Кто может удалить заголовок пакета?

Выполнить оператор `DROP PACKAGE` могут:

- Администраторы
- Владелец пакета;

- Пользователи с привилегией DROP ANY PACKAGE.

Примеры

Пример 186. Удаление заголовка пакета

```
DROP PACKAGE APP_VAR;
```

См. также:

CREATE PACKAGE, ALTER PACKAGE, DROP PACKAGE BODY.

5.10.5. RECREATE PACKAGE

Назначение

Создание нового или пересоздание существующего заголовка пакета.

Доступно в

DSQL

Синтаксис

```
RECREATE PACKAGE package_name
[<rights clause>]
AS
BEGIN
    [<package_item> ...]
END

<package_item> ::= 
    <function_decl>;
| <procedure_decl>;

<function_decl> ::= 
    FUNCTION func_name [(<in_params>)]
    RETURNS <type> [COLLATE collation]
    [DETERMINISTIC]

<procedure_decl> ::= 
    PROCEDURE proc_name [(<in_params>)]
    [RETURNS (<out_params>)]
```

Подробнее см. CREATE PACKAGE

Оператор RECREATE PACKAGE создаёт новый или пересоздаёт существующий заголовок пакета. Если заголовок пакета с таким именем уже существует, то оператор попытается удалить его и создать новый заголовок пакета. Пересоздать заголовок пакета невозможно, если у существующей заголовка пакета имеются зависимости или существует тело этого пакета.

После пересоздания заголовка пакета привилегии на выполнение подпрограмм пакета и привилегии самого пакета не сохраняются.

Примеры

Пример 187. Создание нового или пересоздание существующего заголовка пакета

```
RECREATE PACKAGE APP_VAR
AS
BEGIN
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

См. также:

CREATE PACKAGE, DROP PACKAGE, RECREATE PACKAGE BODY.

5.11. PACKAGE BODY

5.11.1. CREATE PACKAGE BODY

Назначение

Создание тела пакета.

Доступно в

DSQL

Синтаксис

```
CREATE PACKAGE BODY package_name
AS
BEGIN
    [<package_item> ...]
    [<package_body_item> ...]
END

<package_item> ::= 
    <function_decl>;
    | <procedure_decl>;

<function_decl> ::= 
    FUNCTION func_name [(<in_params>)]
    RETURNS <type> [COLLATE collation]
    [DETERMINISTIC]

<procedure_decl> ::=
```

```

PROCEDURE proc_name [(<in_params>)]
[RETURNS (<out_params>)]

<package_body_item> ::= 
    <function_impl>
  | <procedure_impl>

<function_impl> ::= 
    FUNCTION func_name [(<in_impl_params>)]
    RETURNS <type> [COLLATE collation]
    [DETERMINISTIC]
    <routine body>

<procedure_impl> ::= 
    PROCEDURE proc_name [(<in_impl_params>)]
    [RETURNS (<out_params>)]
    <routine body>

<in_params> ::= <inparam> [, <inparam> ...]

<inparam> ::= <param_decl> [= | DEFAULT] <value>

<in_impl_params> ::= <param_decl> [, <param_decl> ...]

<value> ::= {literal | NULL | context_var}

<out_params> ::= <outparam> [, <outparam> ...]

<outparam> ::= <param_decl>

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<type> ::= <datatype> | [TYPE OF] domain_name | TYPE OF COLUMN rel.col

<datatype> ::= 
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

<array_datatype> ::= См. Синтаксис массивов

<routine body> ::= <sql routine body> | <external body reference>

<sql routine body> ::= 
    AS
    [<declarations>]
    BEGIN
    [<PSQL_statements>]
    END

```

```

<declarations> ::= <declare_item> [<declare_item> ...]

<declare_item> ::=
  <declare_var>;
  | <declare_cursor>;
  | <subroutine declaration>;
  | <subroutine implementation>

<subroutine declaration> ::= <subfunc_decl> | <subproc_decl>

<subroutine implementation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=
  EXTERNAL NAME '<extname>' ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

```

Таблица 51. Параметры оператора CREATE PACKAGE BODY

Параметр	Описание
package_name	Имя пакета. Может содержать до 63 символов.
function_decl	Объявление функции.
procedure_decl	Объявление процедуры.
function_impl	Реализация функции.
procedure_impl	Реализация процедуры.
proc_name	Имя процедуры. Может содержать до 63 символов.
func_name	Имя функции. Может содержать до 63 символов.
inparam	Описание входного параметра.
outparam	Описание выходного параметра.
declarations	Секция объявления локальных переменных, именованных курсоров, подпроцедур и подфункций.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
subfunc_decl	Объявление подпрограммы–функции.
subproc_decl	Объявление подпрограммы–процедуры.
subfunc_impl	Реализация подпрограммы–функции.
subproc_impl	Реализация подпрограммы–процедуры.
module name	Имя внешнего модуля, в котором расположена процедура/функция.
routine name	Внутреннее имя процедуры/функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.

Параметр	Описание
extbody	Тело внешней процедуры или функции. Строковый литерал который может использоваться UDR для различных целей.
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
literal	Литерал, совместимый по типу с параметром.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного или выходного параметра процедуры/функции. Может содержать до 63 символов. Имя параметра должно быть уникальным среди входных и выходных параметров процедуры/функции, а также её локальных переменных.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain_name	Имя домена.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.

Оператор CREATE PACKAGE BODY создаёт новое тело пакета. Тело пакета может быть создано только после того как будет создан заголовок пакета. Если заголовка пакета с именем package_name не существует, то будет выдана соответствующая ошибка.

Все процедуры и функции, объявленные в заголовке пакета, должны быть реализованы в теле пакета. Кроме того, должны быть реализованы и все процедуры и функции, объявленные в теле пакета. Процедуры и функции, определенные в теле пакета, но не объявленные в заголовке пакета, не видны вне тела пакета.

Имена процедур и функций, объявленные в теле пакета, должны быть уникальны среди имён процедур и функций, объявленных в заголовке и теле пакета.

 Желательно чтобы имена хранимых процедур и функций пакета не пересекались с именами хранимых процедур и функций из глобального пространства имен, хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноименная функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

Правила:

- В теле пакеты должны быть реализованы все подпрограммы, стоящие по сигнатуре, что и объявленные в заголовке и в начале тела пакета.
- Значения по умолчанию для параметров процедур не могут быть переопределены (которые указываются в <package_item>). Это означает, что они могут быть в

<package_body_item> только для частных процедур, которые не были объявлены.



UDF деклараций (DECLARE внешняя функция) в настоящее время не поддерживается внутри пакетов.

Кто может создать тело пакета?

Выполнить оператор CREATE PACKAGE BODY могут:

- Администраторы
- Владелец пакета;
- Пользователи с привилегией ALTER ANY PACKAGE.

Примеры

Пример 188. Создание тела пакета

```
CREATE PACKAGE BODY APP_VAR
AS
BEGIN
    -- Возвращает дату начала периода
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
    AS
    BEGIN
        RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
    END
    -- Возвращает дату окончания периода
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
    AS
    BEGIN
        RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
    END
    -- Устанавливает диапазон дат рабочего периода
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
    AS
    BEGIN
        RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
        RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
    END
END
```

См. также:

DROP PACKAGE BODY, CREATE PACKAGE.

5.11.2. DROP PACKAGE BODY

Назначение

Удаление тела пакета.

Доступно в
DSQL

Синтаксис

```
DROP PACKAGE BODY package_name
```

Таблица 52. Параметры оператора DROP PACKAGE BODY

Параметр	Описание
package_name	Имя пакета.

Оператор `DROP PACKAGE BODY` удаляет тело пакета.

Кто может удалить тело пакета?

Выполнить оператор `DROP PACKAGE BODY` могут:

- Администраторы
- Владелец пакета;
- Пользователи с привилегией `ALTER ANY PACKAGE`.

Примеры

Пример 189. Удаление тела пакета

```
DROP PACKAGE BODY APP_VAR;
```

См. также:

`CREATE PACKAGE BODY`, `DROP PACKAGE`.

5.11.3. RECREATE PACKAGE BODY

Назначение

Создание нового и пересоздание существующего тела пакета.

Доступно в
DSQL

Синтаксис

```

RECREATE PACKAGE BODY package_name
AS
BEGIN
  [<package_item> ...]
  [<package_body_item> ...]
END

<package_item> ::= 
  <function_decl>;
| <procedure_decl>

<function_decl> ::= 
  FUNCTION func_name [(<in_params>)]
  RETURNS <type> [COLLATE collation]
  [<function_options>]

<procedure_decl> ::= 
  PROCEDURE proc_name [(<in_params>)]
  [RETURNS (<out_params>)]
  [<procedure_options>]

<package_body_item> ::= 
  <function_impl>
| <procedure_impl>

<function_impl> ::= 
  FUNCTION func_name [(<in_impl_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]
  <routine body>

<procedure_impl> ::= 
  PROCEDURE proc_name [(<in_impl_params>)]
  [RETURNS (<out_params>)]
  <routine body>

```

Подробнее см. [CREATE PACKAGE BODY](#).

Оператор RECREATE PACKAGE BODY создаёт новое или пересоздаёт существующее тело пакета. Если тело пакета с таким именем уже существует, то оператор попытается удалить его и создать новое тело пакета. После пересоздания тела пакета привилегии на выполнение подпрограмм пакета и привилегии самого пакета сохраняются.

Примеры

Пример 190. Пересоздание тела пакета

```

RECREATE PACKAGE BODY APP_VAR
AS
BEGIN
    -- Возвращает дату начала периода
    FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
    AS
    BEGIN
        RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
    END
    -- Возвращает дату окончания периода
    FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
    AS
    BEGIN
        RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
    END
    -- Устанавливает диапазон дат рабочего периода
    PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
    AS
    BEGIN
        RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
        RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
    END
END

```

См. также:

[CREATE PACKAGE BODY](#), [DROP PACKAGE BODY](#).

5.12. EXTERNAL FUNCTION

Внешние функции, также известные как функции определяемые пользователем (User Defined Function) — это программы, написанные на любом языке программирования, и хранящиеся в динамических библиотеках. После того как функция объявлена в базе данных, она становится в динамических и процедурных операторах, как будто они реализованы внутри языка SQL.

Внешние функции существенно расширяют возможности SQL по обработке данных. Для того чтобы функции были доступны в базе данных, их необходимо объявить с помощью оператора **DECLARE EXTERNAL FUNCTION**.

После объявления функции, содержащая её библиотека будет загружаться при первом обращении к любой из функций, включённой в библиотеку.

Внешние функции (UDF) объявлены устаревшими в Firebird 4:

- По умолчанию для параметра конфигурации `UdfAccess` установлено значение `None`. Для того, чтобы запускать UDF, теперь потребуется явная конфигурация `UdfAccess = Restrict path-list`.
- UDF библиотеки (`ib_udf`, `fbludf`) больше не входят в состав установочных комплектов.
- Большинство функций в библиотеках, ранее распространявшихся в общих (динамических) библиотеках `ib_udf` и `fbludf`, уже заменены на встроенные функции. Несколько оставшихся UDF были заменены либо аналогичными функциями в новой UDR библиотеке под названием `udf_compat`, либо преобразованы в сохраненные функции.



Обратитесь к разделу “Прекращение поддержки внешних функций (UDF)” в главе “Совместимость” Firebird 4.0 Release Notes для получения подробной информации и инструкций по обновлению для использования безопасных функций.

- Настоятельно рекомендуется заменить UDF на UDR или сохраненные функции. См. [CREATE FUNCTION](#).



UDF принципиально небезопасны. Мы рекомендуем по возможности избегать их использования и отключать UDF в конфигурации вашей базы данных (`UdfAccess = None` в `firebird.conf`, значение по умолчанию начиная с Firebird 4.0). Если вам действительно нужно вызвать собственный код из вашей базы данных, используйте вместо этого механизм UDR.

5.12.1. DECLARE EXTERNAL FUNCTION

Назначение

Объявление в базе данных функции определённой пользователем (UDF).

Доступно в

DSQL, ESQL

Синтаксис

```

DECLARE EXTERNAL FUNCTION funcname
[<arg_type_decl> [, <arg_type_decl> ...]]
RETURNS {
    <sqltype> [BY {DESCRIPTOR | VALUE}] |
    CSTRING(length) |
    PARAMETER param_num }
[FREE_IT]
ENTRY_POINT 'entry_point' MODULE_NAME 'library_name';

```

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

<sqltype> ::= <scalar_datatype> | <blob_datatype>

<scalar_datatype> ::= См. Синтаксис скалярных типов данных

<blob_datatype> ::= См. Синтаксис типа данных BLOB

Таблица 53. Параметры оператора DECLARE EXTERNAL FUNCTION

Параметр	Описание
funcname	Имя внешней функции. Может содержать до 63 символов.
entry_point	Имя экспортруемой функции (точка входа).
library_name	Имя модуля, в котором расположена функция.
sqltype	Тип данных SQL. Не может быть массивом или элементом массива.
length	Максимальная длина нуль терминальной строки. Указывается в байтах.
param_num	Номер входного параметра, который будет возвращён функцией.

Оператор `DECLARE EXTERNAL FUNCTION` делает доступным функцию, определенную пользователем (UDF), в базе данных. Внешние функции должны быть объявлены в каждой базе данных, которая собирается их использовать. Не нужно объявлять UDF, если вы никогда не будете её использовать.

Имя внешней функции должно быть уникальным среди всех имён функций. Оно может отличаться от имени функции указанной в аргументе `ENTRY_POINT`.

Входные параметры функции перечисляются через запятую сразу после имени функции. Для каждого параметра указывается SQL тип данных. Массивы не могут использоваться в качестве параметров функций. Помимо SQL типов можно указать тип `CSTRING`. В этом случае параметр является нуль терминальной строкой с максимальной длиной `length` байт. Существует несколько механизмов передачи параметра из движка Firebird во внешнюю функцию, каждый из этих механизмов будет рассмотрен отдельно.

По умолчанию входные параметры передаются по ссылке. Не существует отдельного предложения для явного указания, что параметр передаётся по ссылке.

При передаче NULL значения по ссылке оно преобразуется в эквивалент нуля, например, число 0 или пустую строку. Если после указанного параметра указано ключевое слово NULL, то при передаче значение NULL оно попадёт в функцию в виде нулевого указателя (NULL).



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.

Если указано предложение BY DESCRIPTOR, то входной параметр передаётся по дескриптору. В этом случае параметр UDF получит указатель на внутреннюю структуру, известную как дескриптор, несущую информацию о типе данных, подтипе, точности, наборе символов и сортировке, масштабе, указателе на сами данные и некоторых флагах, в том числе NULL индикаторе. Отметим, что это объявление работает только в том случае, если внешняя функция написана с использованием дескриптора.



При передаче параметра функции по дескриптору передаваемое значение не приводится к задекларированному типу данных.

Предложение BY SCALAR_ARRAY используется при передачи массивов в качестве входных параметров. В отличие от других типов, вы не можете вернуть массив из UDF.

Обязательное предложение RETURNS описывает выходной параметр возвращаемый функцией. Функция всегда возвращает только один параметр. Выходной параметр может быть любым SQL типом (кроме массива и элемента массива) или нуль терминальной строкой (CSTRING). Выходной параметр может быть передан по ссылке, по дескриптору или по значению. По умолчанию выходной параметр передаётся по ссылке. Если указано предложение BY DESCRIPTOR, то выходной параметр передаётся по дескриптору. Если указано предложение BY VALUE, то выходной параметр передаётся по значению.

Ключевое слово PARAMETER указывает, что функция возвращает значение из параметра с номером `param_num`. Такая необходимость возникает, если необходимо возвращать значение типа BLOB.

Ключевое слово FREE_IT означает, что память, выделенная для хранения возвращаемого значения, будет освобождена после завершения выполнения функции. Применяется только в том случае, если эта память в UDF выделялась динамически. В такой UDF память должна выделяться при помощи функции `ib_util_malloc` из модуля `ib_util`. Это необходимо для совместимости функций выделения и освобождения памяти используемого в коде Firebird и коде UDF.

Предложение ENTRY_POINT указывает имя точки входа (имя экспортруемой функции) в модуле.

Предложение MODULE_NAME задаёт имя модуля, в котором находится экспортруемая функция. В ссылке на модуль может отсутствовать полный путь и расширение файла. Это позволяет легче переносить базу данных между различными платформами. По умолчанию динамические библиотеки пользовательских функций должны располагаться в папке UDF корневого каталога сервера. Параметр UDFAccess в файле *firebird.conf* позволяет изменить ограничения доступа к библиотекам внешних функций.

Кто может объявить внешнюю функцию?

Выполнить оператор DECLARE EXTERNAL FUNCTION могут:

- Администраторы
- Пользователи с привилегией CREATE FUNCTION.

Пользователь, объявивший внешнюю функцию, становится её владельцем.

Примеры

Пример 191. Объявление внешней функции с передачей входных и выходных параметров по ссылке

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

Пример 192. Объявление внешней функции с передачей входных и выходных параметров подескриптору

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

Пример 193. Объявление внешней функции с передачей входных параметров по ссылке, выходных по значению

```
DECLARE EXTERNAL FUNCTION isLeapYear
  TIMESTAMP
  RETURNS INT BY VALUE
  ENTRY_POINT 'isLeapYear' MODULE_NAME 'fbudf';
```

Пример 194. Объявление внешней функции с передачей входных и выходных параметров подескриптору. В качестве выходного параметра используется второй параметр функции.

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

См. также:

[ALTER EXTERNAL FUNCTION](#), [DROP EXTERNAL FUNCTION](#), [CREATE FUNCTION](#).

5.12.2. ALTER EXTERNAL FUNCTION

Назначение

Изменение точки входа и/или имени модуля для функции определённой пользователем (UDF).

Доступно в

DSQL

Синтаксис

```
ALTER EXTERNAL FUNCTION funcname
[ENTRY_POINT 'new_entry_point']
[MODULE_NAME 'new_library_name'];
```

Таблица 54. Параметры оператора ALTER EXTERNAL FUNCTION

Параметр	Описание
funcname	Имя внешней функции.
new_entry_point	Новое имя экспортируемой функции (точки входа).
new_library_name	Новое имя модуля, в котором расположена функция.

Оператор ALTER EXTERNAL FUNCTION изменяет точку входа и/или имя модуля для функции определённой пользователем (UDF). При этом существующие зависимости сохраняются.

Предложение ENTRY_POINT позволяет указать новую точку входа (имя экспортируемой функции).

Предложение MODULE_NAME позволяет указать новое имя модуля, в котором расположена экспортируемая функция.

Кто может изменить внешнюю функцию?

Выполнить оператор ALTER EXTERNAL FUNCTION могут:

- Администраторы
- Владелец внешней функции;
- Пользователи с привилегией ALTER ANY FUNCTION.

Примеры

Пример 195. Изменение точки входа для внешней функции

```
ALTER EXTERNAL FUNCTION invl ENTRY_POINT 'intNvl';
```

Пример 196. Изменение имени модуля для внешней функции

```
ALTER EXTERNAL FUNCTION invl MODULE_NAME 'fbudf2';
```

См. также:

DECLARE EXTERNAL FUNCTION, DROP EXTERNAL FUNCTION.

5.12.3. DROP EXTERNAL FUNCTION

Назначение

Удаление объявления функции определённой пользователем (UDF) из базы данных.

Доступно в

DSQL, ESQL.

Синтаксис

```
DROP EXTERNAL FUNCTION funcname
```

Таблица 55. Параметры оператора DROP EXTERNAL FUNCTION

Параметр	Описание
funcname	Имя внешней функции.

Оператор DROP EXTERNAL FUNCTION удаляет объявление функции определённой пользователем из базы данных. Если есть зависимости от внешней функции, то удаления не произойдёт и будет выдана соответствующая ошибка.

Кто может удалить внешнюю функцию?

Выполнить оператор DROP EXTERNAL FUNCTION могут:

- Администраторы
- Владелец внешней функции;

- Пользователи с привилегией DROP ANY FUNCTION.

Примеры

Пример 197. Удаление внешней функции

```
DROP EXTERNAL FUNCTION addDay;
```

См. также:

DECLARE EXTERNAL FUNCTION.

5.13. FILTER

BLOB фильтр — объект базы данных, являющийся, по сути, специальным видом внешних функций с единственным назначением: получение объекта BLOB одного формата и преобразования его в объект BLOB другого формата. Форматы объектов BLOB задаются с помощью подтипов BLOB.

Внешние функции для преобразования BLOB типов хранятся в динамических библиотеках и загружаются по необходимости.

Подробнее о подтипах BLOB см. в разделе [Бинарные типы данных](#).

5.13.1. DECLARE FILTER

Назначение

Объявление в базе данных BLOB фильтра.

Доступно в

DSQL, ESQL

Синтаксис

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

<sub_type> ::= number | <memonic>

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

Таблица 56. Параметры оператора DECLARE FILTER

Параметр	Описание
filtername	Имя фильтра. Может содержать до 63 символов.
sub_type	Подтип BLOB. См. Подтипы BLOB .
number	Номер подтипа BLOB. См. Подтипы BLOB .
mnemonic	Мнемоника подтипа BLOB. См. Подтипы BLOB .
function_name	Имя экспортруемой функции (точка входа).
library_name	Имя модуля, в котором расположен фильтр.
user_defined	Определяемая пользователем mnemonicика подтипа BLOB.

Оператор `DECLARE FILTER` делает доступным BLOB фильтр в базе данных. Имя BLOB фильтра должно быть уникальным среди имён BLOB фильтров.

Задание подтипов

Подтип может быть задан в виде номера подтипа или мнемоники подтипа. Пользовательские подтипы должны быть представлены отрицательными числами (от -1 до -32768). Не допускается создание 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.

Предложение `INPUT_TYPE` идентифицирует тип преобразуемого объекта (подтип BLOB).

Предложение `OUTPUT_TYPE` идентифицирует тип создаваемого объекта.

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 учёта регистра и без кавычек при объявлении фильтра.

Параметры `DECLARE FILTER`

Предложение `ENTRY_POINT` указывает имя точки входа (имя экспортруемой функции) в модуле.

Предложение `MODULE_NAME` задаёт имя модуля, в котором находится экспортруемая функция. По умолчанию модули должны располагаться в папке `UDF` корневого каталога сервера. Параметр `UDFAccess` в файле `firebird.conf` позволяет изменить ограничения доступа к библиотекам фильтров.

Кто может создать BLOB фильтр?

Выполнить оператор DECLARE FILTER могут:

- Администраторы
- Пользователи с привилегией CREATE FILTER.

Пользователь, создавший BLOB фильтр, становится его владельцем.

Примеры

Пример 198. Создание BLOB фильтра с использованием номеров подтипов

```
DECLARE FILTER DESC_FILTER
INPUT_TYPE 1
OUTPUT_TYPE -4
ENTRY_POINT 'desc_filter'
MODULE_NAME 'FILTERLIB';
```

Пример 199. Создание BLOB фильтра с использованием мнемоник подтипов

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

См. также:

[DROP FILTER](#).

5.13.2. DROP FILTER

Назначение

Удаление объявления BLOB фильтра.

Доступно в

DSQL, ESQL

Синтаксис

```
DROP FILTER filtername
```

Таблица 57. Параметры оператора DROP FILTER

Параметр	Описание
filtername	Имя BLOB фильтра.

Оператор `DROP FILTER` удаляет объявление BLOB фильтра из базы данных. Удаление BLOB фильтра из базы данных делает его не доступным из базы данных, при этом динамическая библиотека, в которой расположена функция преобразования, остаётся не тронутой.

Кто может удалить BLOB фильтр?

Выполнить оператор `DROP FILTER` могут:

- Администраторы
- Владелец BLOB фильтра;
- Пользователи с привилегией `DROP ANY FILTER`.

Примеры

Пример 200. Удаление BLOB фильтра

```
DROP FILTER DESC_FILTER;
```

См. также:

`DECLARE FILTER`.

5.14. SEQUENCE (GENERATOR)

Последовательность (sequence) или генератор (generator) — объект базы данных, предназначенный для получения уникального числового значения. Термин последовательность является SQL совместимым. Ранее в Interbase и Firebird последовательности называли генераторами.

Независимо от диалекта базы данных последовательности (или генераторы) всегда хранятся как 64-битные целые значения.

Если клиент использует 1 диалект, то сервер передаёт ему значения последовательности, усечённые до 32-битного значения. Если значение последовательности передаются в 32-разрядное поле или переменную, то до тех пор, пока текущее значение последовательности не вышло за границы для 32-битного числа, ошибок не будет. В момент выхода значения последовательности 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).

В данном разделе описываются вопросы создания, модификации (установка значения последовательности) и удаления последовательностей.



5.14.1. CREATE SEQUENCE

Назначение

Создание новой последовательности (генератора).

Доступно в

DSQL, ESQL

Синтаксис

```
CREATE {SEQUENCE | GENERATOR} seq_name
[START WITH start_value] [INCREMENT [BY] increment]
```

Таблица 58. Параметры оператора CREATE SEQUENCE

Параметр	Описание
seq_name	Имя последовательности (генератора). Может содержать до 63 символов.
start_value	Начальное значение последовательности (генератора).
increment	Шаг приращения. 4 байтовое целое число.

Оператор CREATE SEQUENCE создаёт новую последовательность. Слова SEQUENCE и GENERATOR являются синонимами. Вы можете использовать любое из них, но рекомендуется использовать SEQUENCE.

В момент создания последовательности ей устанавливается значение, указанное в необязательном предложении START WITH минус значение приращения указанное в предложении INCREMENT [BY]. Если предложение STARTING WITH отсутствует, то последовательности устанавливается значение равное 1. Таким образом, если начальное значение последовательности равно 100, а приращение 10, то первое значение выданное оператором NEXT VALUE FOR будет равно 100.



До Firebird 4.0, первое значение выданное оператором NEXT VALUE FOR было равно 110.

Необязательное предложение INCREMENT [BY] позволяет задать шаг приращения для оператора NEXT VALUES FOR. По умолчанию шаг приращения равен единице. Приращение не может быть установлено в ноль для пользовательских последовательностей. Значение последовательности изменяется также при обращении к функции GEN_ID, где в качестве параметра указывается имя последовательности и значение приращения, которое может быть отлично от указанного в предложении INCREMENT BY.

Кто может создать последовательность?

Выполнить оператор CREATE SEQUENCE (CREATE GENERATOR) могут:

- Администраторы
- Пользователи с привилегией CREATE SEQUENCE (CREATE GENERATOR).

Пользователь, создавший последовательность, становится её владельцем.

Примеры

Пример 201. Создание последовательности

Создание последовательности EMP_NO_GEN с начальным значением 0 и шагом приращения равным единице.

```
CREATE SEQUENCE EMP_NO_GEN;
```

Пример 202. Создание последовательности

Создание последовательности EMP_NO_GEN с начальным значением 5 и шагом приращения равным единице.

```
CREATE SEQUENCE EMP_NO_GEN START WITH 5;
```

Пример 203. Создание последовательности

Создание последовательности EMP_NO_GEN с начальным значением 1 и шагом приращения равным 10.

```
CREATE SEQUENCE EMP_NO_GEN INCREMENT BY 10;
```

Пример 204. Создание последовательности

Создание последовательности EMP_NO_GEN с начальным значением 5 и шагом приращения равным 10.

```
CREATE SEQUENCE EMP_NO_GEN START WITH 5 INCREMENT BY 10;
```

См. также:

`ALTER SEQUENCE, SET GENERATOR, DROP SEQUENCE, NEXT VALUE FOR, GEN_ID.`

5.14.2. ALTER SEQUENCE

Назначение

Изменение последовательности (генератора).

Доступно в

DSQL, ESQL

Синтаксис

```
ALTER {SEQUENCE | GENERATOR} seq_name
[RESTART [WITH newvalue]]
[INCREMENT [BY] increment]
```

Таблица 59. Параметры оператора ALTER SEQUENCE

Параметр	Описание
seq_name	Имя последовательности (генератора).
newvalue	Новое значение последовательности (генератора). 64 битное целое в диапазоне от -2^{63} .. $2^{63} - 1$
increment	Шаг приращения.

Оператор ALTER SEQUENCE устанавливает значение последовательности или генератора в заданное значение и/или изменяет значение приращения.

Предложение RESTART WITH позволяет установить значение последовательности. Предложение RESTART может быть использовано самостоятельно (без WITH) для перезапуска значения последовательности с того значения с которого был начат старт генерации значений или предыдущий рестарт.



Неосторожное использование оператора ALTER SEQUENCE (изменение значения последовательности или генератора) может привести к нарушению логической целостности данных.

Предложение INCREMENT [BY] позволяет изменить шаг приращения последовательности для оператора NEXT VALUES FOR.



Изменение значения приращения — это возможность, которая вступает в силу для каждого запроса, который запускается после фиксаций изменения. Процедуры, которые вызваны впервые после изменения приращения, будут использовать новое значение, если они будут содержать операторы NEXT VALUE FOR. Процедуры, которые уже работают, не будут затронуты, потому что они кэшируются. Процедуры, использующие NEXT VALUE FOR, не должны быть перекомпилированы, чтобы видеть новое приращение, но если они уже работают или загружены, то никакого эффекта не будет. Конечно процедуры, использующие gen_id(gen, <expression>), не затронут при изменении приращения.

Кто может изменить последовательность?

Выполнить оператор ALTER SEQUENCE (ALTER GENERATOR) могут:

- Администраторы

- Владелец последовательности (генератора);
- Пользователи с привилегией ALTER ANY SEQUENCE (ALTER ANY GENERATOR).

Примеры

Пример 205. Изменение последовательности

Установка для последовательности EMP_NO_GEN значения 145.

```
ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
```

Пример 206. Изменение последовательности

Сброс значения последовательности в то, которое было установлено при создании последовательности (или при предыдущей установке значения).

```
ALTER SEQUENCE EMP_NO_GEN RESTART;
```

Пример 207. Изменение последовательности

Изменение значения приращения последовательности EMP_NO_GEN.

```
ALTER SEQUENCE EMP_NO_GEN INCREMENT BY 10;
```

См. также:

[SET GENERATOR](#), [CREATE SEQUENCE](#), [DROP SEQUENCE](#), [NEXT VALUE FOR](#), [GEN_ID](#).

5.14.3. CREATE OR ALTER SEQUENCE

Назначение

Создание новой или изменение существующей последовательности (генератора).

Доступно в

DSQL, ESQL

Синтаксис

```
CREATE OR ALTER {SEQUENCE | GENERATOR} seq_name
[ {START WITH start_value | RESTART} ]
[INCREMENT [BY] increment]
```

Таблица 60. Параметры оператора CREATE OR ALTER SEQUENCE

Параметр	Описание
seq_name	Имя последовательности (генератора). Может содержать до 63 символов.
start_value	Начальное значение последовательности (генератора).
increment	Шаг приращения. 4 байтное целое число.

Если последовательности не существует, то она будет создана. Уже существующая последовательность будет изменена, при этом существующие зависимости последовательности будут сохранены.



Оператор CREATE OR ALTER SEQUENCE требует, чтобы хотя бы одно из необязательных предложений было указано.

Примеры

Пример 208. Создание новой или изменение существующей последовательности

```
CREATE OR ALTER SEQUENCE EMP_NO_GEN
START WITH 10
INCREMENT BY 1;
```

См. также:

[CREATE SEQUENCE](#), [ALTER SEQUENCE](#), [SET GENERATOR](#).

5.14.4. DROP SEQUENCE

Назначение

Удаление последовательности (генератора).

Доступно в

DSQL, ESQL

Синтаксис

```
DROP {SEQUENCE | GENERATOR} seq_name
```

Таблица 61. Параметры оператора DROP SEQUENCE

Параметр	Описание
seq_name	Имя последовательности (генератора).

Оператор DROP SEQUENCE удаляет существующую последовательность (генератор). Слова SEQUENCE и GENERATOR являются синонимами. Вы можете использовать любое из них, но рекомендуется использовать SEQUENCE. При наличии зависимостей для существующей последовательности (генератора) удаления не будет выполнено.

Кто может удалить генератор?

Выполнить оператор DROP SEQUENCE (DROP GENERATOR) могут:

- Администраторы
- Владелец последовательности (генератора);
- Пользователи с привилегией DROP ANY SEQUENCE (DROP ANY GENERATOR).

Примеры

Пример 209. Удаление последовательности

```
DROP SEQUENCE EMP_NO_GEN;
```

См. также:

[CREATE SEQUENCE](#), [ALTER SEQUENCE](#), [RECREATE SEQUENCE](#).

5.14.5. RECREATE SEQUENCE

Назначение

Создание или пересоздание последовательности (генератора).

Доступно в

DSQL, ESQL

Синтаксис

```
RECREATE {SEQUENCE | GENERATOR} seq_name
[START WITH start_value]
[INCREMENT [BY] increment];
```

Таблица 62. Параметры оператора RECREATE SEQUENCE

Параметр	Описание
seq_name	Имя последовательности (генератора). Может содержать до 63 символов.
start_value	Начальное значение последовательности (генератора).
increment	Шаг приращения. 4 байтное целое число.

Оператор RECREATE SEQUENCE создаёт или пересоздаёт последовательность (генератор). Если последовательность с таким именем уже существует, то оператор RECREATE SEQUENCE попытается удалить её и создать новую последовательность. При наличии зависимостей для существующей последовательности оператор RECREATE SEQUENCE не выполнится.

Примеры

Пример 210. Пересоздание последовательности

```
RECREATE SEQUENCE EMP_NO_GEN
START WITH 10
INCREMENT BY 1;
```

5.14.6. SET GENERATOR

Назначение

Устанавливает значение последовательности или генератора в заданное значение.

Доступно в

DSQL, ESQL

Синтаксис

```
SET GENERATOR seq_name TO new_val
```

Таблица 63. Параметры оператора SET GENERATOR

Параметр	Описание
seq_name	Имя последовательности (генератора).
new_val	Новое значение последовательности (генератора). 64 битное целое в диапазоне от $-2^{63} .. 2^{63} - 1$

Оператор SET GENERATOR устанавливает значение последовательности или генератора в заданное значение.

Оператор SET GENERATOR считается устаревшим и оставлен ради обратной совместимости. В настоящее время вместо него рекомендуется использовать стандарт-совместимый оператор ALTER SEQUENCE.



Неосторожное использование оператора SET GENERATOR (изменение значения последовательности или генератора) может привести к потере логической целостности данных.

Кто может изменить значение генератора?

Выполнить оператор SET GENERATOR могут:

- Администраторы
- Владелец последовательности (генератора);
- Пользователи с привилегией ALTER ANY SEQUENCE (ALTER ANY GENERATOR).

Примеры

Пример 211. Установка значения для последовательности

```
SET GENERATOR EMP_NO_GEN TO 145;
```

То же самое можно сделать, используя оператор **ALTER SEQUENCE**



```
ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
```

См. также:

[ALTER SEQUENCE, NEXT VALUE FOR, GEN_ID.](#)

5.15. COLLATION

В SQL текстовые строки принадлежат к сортируемым объектам. Это означает, что они подчиняются своим внутренним правилам упорядочения, например, алфавитному порядку. К таким текстовым строкам можно применять операции сравнения (например, “меньше чем” или “больше чем”), при этом значения выражения должны вычисляться согласно определённой последовательности сортировки. Например, выражение 'a'<'b' означает, что 'a' предшествует 'b' в последовательности сортировки. Под выражением 'c'>'b' имеется в виду, что в последовательности сортировки 'c' определено после 'b'. Текстовые строки, включающие больше одного символа, сортируются путём последовательного сравнения символов: сначала сравниваются первые символы двух строк, затем вторые символы и так далее, до тех пор, пока не будет найдено различие между двумя строками. Такое различие управляет порядком сортировки.

Под сравнением (сортировкой) (COLLATION) принято понимать такой объект схемы, который определяет упорядочивающую последовательность (или последовательность сортировки).

5.15.1. CREATE COLLATION

Назначение

Добавление новой сортировки (сравнения) для набора символов поддерживаемого в базе данных.

Доступно в

DSQL

Синтаксис

```

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

<specific-attributes> ::= <attribute> [; <attribute> ...]

<attribute> ::= attrname=attrvalue

```

Таблица 64. Параметры оператора CREATE COLLATION

Параметр	Описание
collname	Имя сортировки (сравнения). Максимальная длина 63 символов.
charset	Набор символов.
basecoll	Базовая сортировка (сравнение).
extname	Имя сортировки из конфигурационного файла. Чувствительно к регистру.

Оператор `CREATE COLLATION` ничего не “создаёт”, его целью является сделать сортировку известной для базы данных. Сортировка уже должна присутствовать в системе, как правило, в файле библиотеки, и должна быть зарегистрирована в файле `fbintl.conf` подкаталога `intl` корневой директории Firebird.

Необязательное предложение `FROM` указывает сортировку, на основе которой будет создана новая сортировка. Такая сортировка должна уже присутствовать в базе данных. Если указано ключевое слово `EXTERNAL`, то будет осуществлён поиск сортировки из файла `$fbroot/intl/fbintl.conf`, при этом `extname` должно в точности соответствовать имени в конфигурационном файле (чувствительно к регистру).

Если предложение `FROM` отсутствует, то Firebird ищет в конфигурационном файле `fbintl.conf` подкаталога `intl` корневой директории сервера сортировку с именем, указанным сразу после `CREATE COLLATION`. Другими словами, отсутствие предложения `FROM basecoll` эквивалентно заданию `FROM EXTERNAL ('collname')`.

При создании сортировки можно указать учитываются ли конечные пробелы при сравнении. Если указана опция `NO PAD`, то конечные пробелы при сравнении учитываются. Если указана опция `PAD SPACE`, то конечные пробелы при сравнении не учитываются.

Необязательное предложение `CASE` позволяет указать будет ли сравнение чувствительно к регистру.

Необязательное предложение `ACCENT` позволяет указать будет ли сравнение чувствительно к акцентированным буквам (например “е” и “ё”).

Специфичные атрибуты

В операторе `CREATE COLLATION` можно также указать специфичные атрибуты для сортировки. Ниже в таблице приведён список доступных специфичных атрибутов. Не все атрибуты применимы ко всем сортировкам. Если атрибут не применим к сортировке, но указан при её создании, то это не вызовет ошибки.



Имена специфичных атрибутов чувствительны к регистру.

“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 65. Список доступных специфичных атрибутов COLLATION

Имя	Значение	Валидность	Описание
DISABLE-COMPRESSIONS	0, 1	1 bpc	Отключает сжатия (иначе сокращения). Сжатия заставляют определённые 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 рассматривать определённые символы (например, лигатуры или гласные умляуты) как последовательности символов и соответственно сортировать.
ICU-VERSION	default или M.m	UNI	Задаёт версию библиотеки ICU для использования. Допустимые значения определены в соответствующих элементах <code><intl_module></code> в файле <code>intl/fbintl.conf</code> . Формат: либо строка “default” или основной и дополнительный номер версии, как “3.0” (оба без кавычек).
LOCALE	xx_YY	UNI	Задаёт 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.

Имя	Значение	Валидность	Описание
NUMERIC-SORT	0, 1	UNI	Обрабатывает непрерывные группы десятичных цифр в строке как атомарные модули и сортирует их в числовой последовательности (известна как естественная сортировка).
SPECIALS-FIRST	0, 1	1 bpc	Сортирует специальные символы (пробелы и т.д.) до буквенно-цифровых символов.



Если вы хотите добавить в базу данных новый набор символов с его умалчивающей сортировкой, то зарегистрируйте и выполните хранимую процедуру `sp_register_character_name(name, max_bytes_per_character)` из подкаталога `misc/intl.sql` установки Firebird. Для нормальной работы с набором символов, он должен присутствовать в вашей операционной системе, и зарегистрирован в файле `fbintl.conf` поддиректории `intl`.

Кто может создать сортировку?

Выполнить оператор `CREATE COLLATION` могут:

- Администраторы
- Пользователи с привилегией `CREATE COLLATION`.

Пользователь, создавший сортировку, становится её владельцем.

Примеры

Пример 212. Создание сортировки с использованием имени, найденного в файле `fbintl.conf` (регистро-чувствительно).

```
CREATE COLLATION IS08859_1_UNICODE FOR IS08859_1;
```

Создание сортировки с использованием специального (заданного пользователем) названия (“external” имя должно в

точности соответствовать имени в файле `fbintl.conf`).

```
CREATE COLLATION LAT_UNI
FOR IS08859_1
FROM EXTERNAL ('IS08859_1_UNICODE');
```

Пример 213. Создание регистрационезависимой сортировки на основе уже присутствующей в базе данных.

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

Пример 214. Создание регистрационезависимой сортировки на основе уже присутствующей в базе данных со специфичными атрибутами.

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

Пример 215. Создание регистрационезависимой сортировки по значению чисел (так называемой натуральной сортировки).

```
CREATE COLLATION nums_coll FOR UTF8
FROM UNICODE
CASE INSENSITIVE 'NUMERIC-SORT=1';

CREATE DOMAIN dm_nums AS varchar(20)
CHARACTER SET UTF8 COLLATE nums_coll; -- original (manufacturer) numbers

CREATE TABLE wares(id int primary key, articul dm_nums ...);
```

См. также:

[DROP COLLATION](#).

5.15.2. DROP COLLATION

Назначение

Удаление существующей сортировки.

Доступно в

DSQL

Синтаксис

```
DROP COLLATION collname
```

Таблица 66. Параметры оператора *DROP COLLATION*

Параметр	Описание
collname	Имя сортировки.

Оператор **DROP COLLATION** удаляет указанную сортировку. Сортировка должна присутствовать в базе данных, иначе будет выдана соответствующая ошибка.



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.

Кто может удалить сортировку?

Выполнить оператор **DROP COLLATION** могут:

- Администраторы
- Владелец сортировки;
- Пользователи с привилегией **DROP ANY COLLATION**.

Примеры

Пример 216. Удаление сортировки

```
DROP COLLATION ES_ES_NOPAD_CI;
```

См. также:

[CREATE COLLATION](#).

5.16. CHARACTER SET

5.16.1. ALTER CHARACTER SET

Назначение

Установка сортировки по умолчанию для набора символов.

Доступно в

DSQL

Синтаксис

```
ALTER CHARACTER SET charset
SET DEFAULT COLLATION collation
```

Таблица 67. Параметры оператора ALTER CHARACTER SET

Параметр	Описание
charset	Набор символов.
collation	Сортировка.

Оператор ALTER CHARACTER SET изменяет сортировку по умолчанию для указанного набора символов. Это повлияет на использование набора символов в будущем, кроме случаев, когда явно переопределена сортировка COLLATE. Сортировка существующих доменов, столбцов и переменных PSQL при этом не будет изменена.

Если сортировка по умолчанию была изменена для набора символов базы данных (тот, что был указан при создании базы данных), то также изменяется и сортировка по умолчанию для базы данных.



Если сортировка по умолчанию была изменена для набора символов, который был указан при подключении, строковые константы будут интерпретироваться в соответствии с новыми параметрами сортировки (если набор символов и/или сортировка не переопределяются).

Примеры

Пример 217. Установка сортировки UNICODE_CI_AI по умолчанию для кодировки UTF8

```
ALTER CHARACTER SET UTF8 SET DEFAULT COLLATION UNICODE_CI_AI;
```

5.17. COMMENTS

Объекты базы данных и сама база данных могут содержать примечания. Это удобное средство документирования во время разработки базы данных и её поддержки.

5.17.1. COMMENT ON

Назначение

Документирование метаданных.

Доступно в

DSQL, ESQL

Синтаксис

```

COMMENT ON <object> IS {'sometext' | NULL}
<object> ::= 
    DATABASE
  | <basic-type> objectname
  | COLUMN relationname.fieldname
  | [PROCEDURE | FUNCTION] PARAMETER
    [package_name.] routinename.paramname
  | {PROCEDURE | [EXTERNAL] FUNCTION}
    routinename

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

```

```
COMMENT ON USER <username> [USING PLUGIN <plugin_name>] IS {'sometext' | NULL}
```

Таблица 68. Параметры оператора COMMENT ON

Параметр	Описание
sometext	Текст комментария.
basic-type	Тип объекта метаданных.
objectname	Имя объекта метаданных.
relationname	Имя таблицы или представления.
fieldname	Имя поля таблицы или представления.
routinename	Имя хранимой процедуры или функции.
paramname	Имя параметра хранимой процедуры или функции.
package_name	Имя пакета.
username	Имя пользователя.
plugin_name	Имя плагина управления пользователями.

Оператор COMMENT ON добавляет комментарии для объектов базы данных (метаданных). Комментарии при этом сохраняются в текстовые поля RDB\$DESCRIPTION типа BLOB соответствующей системной таблицы (из этих полей клиентское приложение может

просмотреть комментарии).

При добавлении комментария для пользователя вы можете уточнить в каком плагине управления пользователями он находится с помощью необязательного предложения USING PLUGIN. Если это предложение отсутствует, то предполагает что пользователь создан в плагине управления по умолчанию, то есть первого плагина указанного в параметре UserManager в файле *firebird.conf* или *databases.conf*.



Если вы вводите пустой комментарий (""), то он будет сохранен в базе данных как NULL.

Кто может добавить комментарий?

Выполнить оператор COMMENT ON могут:

- Администраторы
- Владелец объекта, для которого добавляется комментарий;
- Пользователи с привилегией ALTER ANY <object_type>.

Примеры

Пример 218. Добавление комментария для текущей базы данных.

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

Пример 219. Добавление комментария для таблицы.

```
COMMENT ON TABLE METALS IS 'Справочник металлов';
```

Пример 220. Добавление комментария для поля таблицы.

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

Пример 221. Добавление комментария для параметра процедуры.

```
COMMENT ON PARAMETER ADD_EMP_PROJ.EMP_NO
IS 'Код сотрудника';
```

Пример 222. Добавление комментария для пакета, его процедур и функций, и их параметров.

```
COMMENT ON PACKAGE APP_VAR IS 'Переменные приложения';  
  
COMMENT ON FUNCTION APP_VAR.GET_DATEBEGIN  
IS 'Возвращает дату начала периода';  
  
COMMENT ON PROCEDURE APP_VAR.SET_DATERANGE  
IS 'Установка диапазона дат';  
  
COMMENT ON  
PROCEDURE PARAMETER APP_VAR.SET_DATERANGE.ADATEBEGIN  
IS 'Дата начала';
```

Пример 223. Добавление комментария для пользователя.

```
COMMENT ON USER BOB35 IS 'Это Боб из плагина по умолчанию';  
  
COMMENT ON USER JHON USING PLUGIN Legacy_UserManager  
IS 'Это Джон из плагина Legacy_UserManager';
```

Приложение A: License notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the “License”); you may only use this Documentation if you comply with the terms of this License. Copies of the License are available at <https://www.firebirdsql.org/pdfmanual/pdl.pdf> (PDF) and <https://www.firebirdsql.org/manual/pdl.html> (HTML).

The Original Documentation is titled *Firebird 4.0 Language Reference*. This Documentation was derived from *Firebird 2.5 Language Reference*.

The Initial Writers of the Original Documentation are: Paul Vinkenoog, Dmitry Yemanov, Thomas Woinke and Mark Rotteveel. Writers of text originally in Russian are Denis Simonov, Dmitry Filippov, Alexander Karpeykin, Alexey Kovyazin and Dmitry Kuzmenko.

Copyright © 2008-2021. All Rights Reserved. Initial Writers contact: paul at vinkenoog dot nl.

Writers and Editors of included PDL-licensed material are: J. Beesley, Helen Borrie, Arno Brinkman, Frank Ingermann, Vlad Khorsun, Alex Peshkov, Nickolay Samofatov, Adriano dos Santos Fernandes, Dmitry Yemanov.

Included portions are Copyright © 2001-2020 by their respective authors. All Rights Reserved.

Contributor(s): Mark Rotteveel, Denis Simonov.

Portions created by Mark Rotteveel and Denis Simonov are Copyright © 2018-2021. All Rights Reserved. (Contributor contact(s): mrotteveel at users dot sourceforge dot net). (Contributor contact(s): sim-mail at list dot dot ru).