РЕД База Данных

Версия 5

Примечания к выпуску

Содержание

1 Новое в СУБД РЕД База Данных 5 9

2 Изменения в работе сервера 10
2.1 Максимальный размер страницы увеличен до 32Кб . . . 10
2.2 Поддержка международных временных зон . . . 10
2.2.1 Часовой пояс сеанса . . . 10
2.2.2 Формат часового пояса . . . 10
2.2.3 Типы данных для поддержки временных зон . . . 10
2.2.4 Выражения и операторы для работы с временными зонами . . . 11
Оператор SET TIME ZONE . . . 11
Выражение AT . . . 11
Выражение EXTRACT . . . 11
Выражение LOCALTIME . . . 12
Выражение LOCALTIMESTAMP . . . 12
Изменения в CURRENT_TIME и CURRENT_TIMESTAMP . . . 12
2.2.5 Таблица RDB$TIME_ZONES . . . 12
2.2.6 Пакет RDB$TIME_ZONE_UTIL . . . 12
Функция DATABASE_VERSION . . . 13
Процедура TRANSITIONS . . . 13
2.2.7 Обновление базы данных временных зон . . . 14
2.3 Репликация . . . 14
2.3.1 Режимы доступа . . . 14
2.3.2 Изменения параметров конфигурации репликации . . . 14
Параметры настройки ведущего . . . 15
Параметры настройки на реплике . . . 15
Поддержка макросов . . . 16
2.3.3 Настройка системы репликации . . . 16
2.3.4 Инициализация асинхронной репликации . . . 16
Пример настройки асинхронной репликации . . . 18
2.3.5 Инициализация синхронной репликации . . . 19
Пример настройки синхронной репликации . . . 20
2.4 Пул внешних подключений . . . 20
2.4.1 Настройки пула внешних подключений . . . 22
ExtConnPoolSize . . . 22
ExtConnPoolLifeTime . . . 22
2.5 Таймаут для соединений и запросов . . . 22
2.5.1 Таймаут простоя соединения . . . 22
2.5.2 Таймаут выполнения SQL-запроса . . . 23
2.6 Согласованное чтение . . . 25
2.6.1 Определение видимости записей . . . 25
Порядок фиксации транзакций . . . 25
Специальные значения CN транзакции . . . 25
Правило определения видимости записей . . . 25
Детали реализации . . . 26
2.6.2 Согласованное чтение для запросов в транзакциях READ COMMITTED . . . 26
2.6.3 Решение проблемы несогласованного чтения . . . 27
Уровень изоляции READ COMMITTED READ CONSISTENCY . . . 27
Обработка конфликта обновлений . . . 27
Транзакции Read Committed Read-Only . . . 28
Синтаксис и конфигурация . . . 29

2.6.4 Сборка мусора . . . 29
2.7 NUMERIC и DECIMAL . . . 30
2.8 Увеличено количество форматов для представлений . . . 30
2.9 Улучшение оптимизатора для GROUP BY . . . 30
2.10 Заменена поддержка xinetd в Linux . . . 30
2.11 Виртуальная таблица RDB$CONFIG . . . 30
2.12 Введено ограничение на количество символов для UNICODE_FSS . . . 31
2.13 Табличные пространства . . . 31
2.14 Планировщик заданий . . . 33
2.14.1 Создание задания . . . 34
2.14.2 Изменение задания . . . 35
2.14.3 Удаление задания . . . 35
2.14.4 Оповещения . . . 35
2.15 Блокирование сеанса пользователя . . . 36
2.16 Использование BLR вместо SQL кода . . . 37
2.17 Поддержка многопоточной работы . . . 37
2.18 Обновление до последней минорной версии ODS . . . 38
2.19 Больше подробностей о курсоре в плане запроса . . . 38
2.20 Сжатие записей . . . 39
2.21 Кэширование скомпилированных запросов . . . 39
2.22 Поддержка двунаправленных курсоров в сетевом протоколе . . . 39
2.23 PSQL-профайлер . . . 39
2.23.1 Функция START_SESSION . . . 42
2.23.2 Процедура PAUSE_SESSION . . . 42
2.23.3 Процедура RESUME_SESSION . . . 43
2.23.4 Процедура FINISH_SESSION . . . 43
2.23.5 Процедура CANCEL_SESSION . . . 43
2.23.6 Процедура DISCARD . . . 43
2.23.7 Процедура FLUSH . . . 43
2.23.8 Процедура SET_FLUSH_INTERVAL . . . 44
2.23.9 Таблицы снапшотов . . . 44
2.23.10 Дополнительные представления . . . 46
PLG$PROF_STATEMENT_STATS_VIEW . . . 46
PLG$PROF_PSQL_STATS_VIEW . . . 47
PLG$PROF_RECORD_SOURCE_STATS_VIEW . . . 49
2.24 Пакет RDB$BLOB_UTIL . . . 51
2.24.1 Функция NEW_BLOB . . . 51
2.24.2 Функция OPEN_BLOB . . . 52
2.24.3 Функция IS_WRITABLE . . . 52
2.24.4 Функция READ_DATA . . . 53
2.24.5 Функция SEEK . . . 54
2.24.6 Процедура CANCEL_BLOB . . . 54
2.24.7 Процедура CLOSE_HANDLE . . . 55
2.25 Функция RDB$TRACE_MSG . . . 55

3 Изменения API и ODS 56
3.1 Изменения ODS . . . 56
3.1.1 Новые системные таблицы . . . 56
3.1.2 Новые столбцы в системных таблицах . . . 56
3.2 Программные интерфейсы . . . 57
3.2.1 Изменения основного API . . . 57
ResultSet . . . 57
3.2.2 Расширение метода getInfo() . . . 57
Attachment::getInfo() . . . 57

Statement::getInfo() . . . 58
Transaction::getInfo() . . . 58
3.2.3 Расширение Legacy (ISC) API . . . 58
3.2.4 Изменения в Services API . . . 58
Поддержка gstat -parallel . . . 58
Поддержка gfix -upgrade . . . 59
Поддержка nbackup -fixup . . . 59
3.2.5 Поддержка таймаута простоя соединения в API . . . 59
3.2.6 Поддержка таймаута выполнения операторов в API . . . 59
3.2.7 Поддержка READ COMMITTED READ CONSISTENCY в API . . . 60
3.2.8 Поддержка пакетных операций в API . . . 60
Создание пакета . . . 60
Создание пакетного интерфейса . . . 61
Выполнение пакетной операции . . . 61
Добавление множества сообщений за один вызов пакетной операции . . . 62
Передача встроенных BLOB в пакетных операциях . . . 62
Пакетные операции в Legacy (ISC) API . . . 65
3.2.9 Поддержка временных зон в API . . . 65
Структура . . . 65
API функции . . . 66
3.2.10 Поддержка NUMERIC и DECIMAL в API . . . 67
3.2.11 Изменения в других интерфейсах . . . 69

4 Изменения списка зарезервированных слов 71
4.1 Новые ключевые слова . . . 71
4.2 Новые зарезервированные слова . . . 71

5 Изменения параметров конфигурации 73
5.1 Новые параметры конфигурации . . . 73
5.2 Изменённые параметры конфигурации . . . 76
5.3 Удалённые параметры конфигурации . . . 77

6 Безопасность 78
6.1 Системные привилегии . . . 78
6.1.1 Системные привилегии в операторе GRANT . . . 79
6.1.2 Назначение системной привилегии на роль . . . 79
Функция RDB$SYSTEM_PRIVILEGE . . . 80
6.2 Кумулятивное действие ролей . . . 80
6.2.1 Ключевое слово DEFAULT . . . 80
6.2.2 Предложение WITH ADMIN OPTION . . . 80
6.2.3 Пример кумулятивного действия ролей . . . 81
6.2.4 Функция RDB$ROLE_IN_USE . . . 81
6.3 Функция SQL SECURITY . . . 81
6.3.1 Триггеры . . . 82
6.3.2 Примеры использования SQL SECURITY . . . 82
6.4 Встроенные криптографические функции . . . 85
6.4.1 ENCRYPT . . . 85
6.4.2 DECRYPT . . . 85
6.4.3 RSA_PRIVATE . . . 86
6.4.4 RSA_PUBLIC . . . 86
6.4.5 RSA_ENCRYPT . . . 86
6.4.6 RSA_DECRYPT . . . 87
6.4.7 RSA_SIGN . . . 87
6.4.8 RSA_VERIFY . . . 87

6.5 Управление пользователями . . . 88
6.6 Отслеживание событий до установления действующего контекста безопасности . . . 88
6.7 Трассировка события COMPILE . . . 89
6.8 Политики безопасности . . . 89

7 Операторы управления 90
7.1 Управление пулом внешних соединений . . . 90
7.1.1 ALTER EXTERNAL CONNECTIONS POOL . . . 90
ALTER EXTERNAL CONNECTIONS POOL SET SIZE . . . 90
ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME . . . 90
ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL . . . 91
ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST . . . 91
7.2 ALTER SESSION RESET . . . 91
7.3 Управление часовыми поясами . . . 91
7.3.1 SET TIME ZONE . . . 92
7.3.2 SET TIME ZONE BIND . . . 92
7.4 Управление таймаутами . . . 92
7.4.1 SET SESSION IDLE TIMEOUT . . . 92
7.4.2 SET STATEMENT TIMEOUT . . . 93
7.5 Настройка параметров DECFLOAT . . . 93
7.5.1 SET DECFLOAT BIND . . . 93
7.5.2 SET DECFLOAT TRAPS TO . . . 94
7.5.3 SET DECFLOAT ROUND . . . 94
7.6 Настройка правил приведения типов данных . . . 95
7.7 SET OPTIMIZE . . . 97

8 Язык описания данных (DDL) 98
8.1 Увеличена длина имен объектов . . . 98
8.1.1 Ограничение длины . . . 98
8.2 Новые типы данных . . . 98
8.2.1 INT128 . . . 98
8.2.2 TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE . . . 98
8.2.3 DECFLOAT . . . 99
Использование DECFLOAT . . . 99
8.3 Улучшения DDL . . . 99
8.3.1 Увеличена точность NUMERIC и DECIMAL . . . 99
8.3.2 Тип данных FLOAT соответствует стандарту . . . 100
8.3.3 Типы данных для поддержки временных зон . . . 100
8.3.4 Алиасы для бинарных строковых типов . . . 100
8.3.5 Улучшения типа IDENTITY . . . 101
Синтаксис для управления столбцами IDENTITY . . . 101
Опции GENERATED ALWAYS и BY DEFAULT . . . 101
Изменение поведения по умолчанию . . . 102
Предложение DROP IDENTITY . . . 102
Опция INCREMENT BY в столбцах IDENTITY . . . 103
Изменение значения шага увеличения . . . 103
Реализация . . . 103
8.3.6 `EXCESS` в `EXECUTE STATEMENT` . . . 103
8.3.7 Управление репликацией . . . 104
8.4 Поддержка частичных индексов . . . 105
8.5 Комментарии для отображений . . . 105

9 Язык управления данными (DML) 106
9.1 JSON . . . 106

9.1.1 Язык путей JSON . . . 106
9.1.2 Режимы: lax и strict . . . 107
9.1.3 Доступ к элементам . . . 108
Обращение к элементу объекта . . . 108
Обращение к элементу массива . . . 109
9.1.4 Методы элементов . . . 109
Метод `type()` . . . 110
Метод `size()` . . . 110
Числовые методы элементов: `double()`, `ceiling()`, `floor()`, `abs()` . . . 111
Метод `keyvalue()` . . . 112
9.1.5 Арифметические операции в пути . . . 112
Унарный плюс и минус . . . 113
Бинарные операции . . . 113
9.1.6 Фильтры . . . 114
Обработка ошибок в фильтрах . . . 115
Таблицы истиности . . . 116
Операторы сравнения . . . 116
Предикат `like_regex` . . . 117
Предикат `starts with` . . . 118
Предикат `exists` . . . 118
Предикат `is unknown` . . . 118
9.1.7 Функции SQL/JSON . . . 119
Общий синтаксис функций работы с данными . . . 119
Значение JSON . . . 120
Выражение пути . . . 120
Оператор передачи контекстных переменных . . . 121
Выходное значение . . . 121
9.1.8 Функции работы с данными . . . 121
`JSON_VALUE` . . . 122
`JSON_QUERY` . . . 122
JSON_MODIFY . . . 124
`JSON_TABLE` . . . 125
Вложенные столбцы . . . 130
Предложение `PLAN` . . . 130
9.1.9 Функции валидации . . . 134
`JSON_EXISTS` . . . 134
`IS JSON` . . . 134
9.1.10 Функции генерации данных . . . 136
Общий формат входных значений . . . 137
`JSON_OBJECT` . . . 137
`JSON_OBJECTAGG` . . . 139
`JSON_ARRAY` . . . 140
`JSON_ARRAYAGG` . . . 141
9.2 Предложение SKIP LOCKED . . . 141
9.3 Поддержка WHEN NOT MATCHED BY SOURCE в операторе MERGE . . . 143
9.4 Поддержка многострочного вывода RETURNING . . . 143
9.5 Вложенные выражения . . . 143
9.6 Поддержка `PLAN` и `ORDER BY` в операторе `MERGE` . . . 144
9.7 Поддержка `PLAN`, `ORDER BY` и `ROWS` в операторе `UPDATE OR INSERT` . . . 144
9.8 Предложение `OPTIMIZE FOR` . . . 145
9.9 Изменения в литералах . . . 145
9.9.1 Изменения синтаксиса строковых литералов . . . 145
9.9.2 Изменения синтаксиса двоичных строк . . . 146
9.9.3 Изменения синтаксиса числовых литералов . . . 146

9.10 Улучшения в IN . . . 149
9.11 Новые выражения и встроенные функции . . . 149
9.11.1 UNICODE_CHAR . . . 149
9.11.2 UNICODE_VAL . . . 149
9.11.3 QUARTER добавлен в функции EXTRACT, FIRST_DAY и LAST_DAY . . . 149
9.12 Соединение с производными таблицами . . . 150
9.13 Значение DEFAULT для добавления и обновления . . . 150
9.13.1 DEFAULT и DEFAULT VALUES . . . 151
9.14 Предложение OVERRIDING для столбцов IDENTITY . . . 151
9.15 Улучшения оконных функций . . . 152
9.15.1 Рамки для оконных функций . . . 153
Навигационные функции с фреймами . . . 154
Примеры использования рамок . . . 154
9.15.2 Именованные окна . . . 155
9.15.3 Ранжирующие функции . . . 156
9.16 Предложение FILTER для агрегатных функций . . . 157
9.17 AUTOCOMMIT для SET TRANSACTION . . . 157
9.18 Совместное использование снимков транзакций . . . 158
9.19 Выражения и встроенные функции . . . 158
9.19.1 Новые функции и выражения . . . 158
Функции и выражения для работы с часовыми поясами . . . 158
Новые функции для работы с датой и временем . . . 159
Функции безопасности . . . 159
Функции для DECFLOAT . . . 160
Функция RDB$GET_TRANSACTION_CN . . . 161
MAKE_DBKEY . . . 162
BASE64_ENCODE . . . 163
BASE64_DECODE . . . 163
HEX_ENCODE . . . 163
HEX_DECODE . . . 163
CRYPT_HASH . . . 163
BLOB_APPEND . . . 163
Функция UNLIST . . . 165
9.19.2 Изменения в встроенных функциях и выражениях . . . 166
Выражение EXTRACT . . . 166
Изменения в CURRENT_TIME и CURRENT_TIMESTAMP . . . 166
HASH . . . 166
SUBSTRING . . . 166
9.19.3 Изменения UDF . . . 167
Новая UDR GetExactTimestampUTC . . . 167
9.19.4 Новые контекстные переменные пространства имён SYSTEM . . . 167
9.20 Другие улучшения DML . . . 168
9.20.1 Уточнение сообщения об ошибке при недопустимой операции записи . . . 168
9.20.2 Уточнение сообщения об ошибке для выражений индексов . . . 168
9.20.3 Поддержка RETURNING * . . . 168

10 Процедурный SQL (PSQL) 169
10.1 Подпрограммы могут обращаться к переменным, определенным на внешнем уровне . . . 169
10.2 Рекурсия для подпрограмм . . . 169
10.3 Функция RDB$ERROR . . . 170
10.4 Операторы управления в блоках PSQL . . . 171

11 Мониторинг и утилиты командной строки 172
11.1 Мониторинг . . . 172

11.1.1 Новые таблицы мониторинга . . . 172
RDB$KEYWORDS . . . 172
MON$COMPILED_STATEMENTS . . . 172
11.1.2 Новые столбцы в таблицах мониторинга . . . 172
11.2 NBACKUP . . . 173
11.2.1 Инкрементное резервное копирование . . . 173
Создание резервной копии . . . 173
Применение изменений . . . 174
Пример резервного копирования и восстановления . . . 174
11.2.2 Восстановление и исправление реплики базы данных . . . 174
11.3 ISQL . . . 174
11.3.1 Поддержка таймаута запросов . . . 174
11.3.2 Улучшенный контроль транзакций . . . 175
11.3.3 SHOW SYSTEM . . . 176
11.3.4 Отображение BLR оператора . . . 176
11.3.5 Информация о репликации добавлена в вывод SHOW DATABASE . . . 176
11.4 GBAK . . . 176
11.4.1 Резервное копирование и восстановление с шифрованием . . . 176
Необходимые условия . . . 177
Новые ключи для зашифрованного резервного копирования и восстановления . . . 177
Примеры . . . 177
11.4.2 Повышение производительности рестора . . . 178
Изменены "-fix_fss_*" сообщения . . . 178
11.4.3 Возможность выполнения бэкапа/рестора только выбранных таблиц . . . 178
11.5 GSTAT . . . 179
11.6 GFIX . . . 179
11.6.1 Обновление минорной версии ODS . . . 179
11.6.2 Настройка и управление репликацией . . . 179
11.7 Агрегатный аудит . . . 179

12 Устаревшие функции 182
12.1 Диалект 1 . . . 182
12.2 Внешние функции (UDF) . . . 182
12.3 AUTO ADMIN MAPPING . . . 183
12.4 Уровень изоляции READ COMMITTED [NO] RECORD VERSION . . . 183
12.5 Утилита GSEC . . . 183
12.6 Опции GBAK . . . 183

13 Проблемы совместимости 184
13.1 Для транзакций READ COMMITTED по умолчанию используется Read Consistency . . . 184
13.2 Изменения в DDL и DML для поддержки часовых поясов . . . 184
13.2.1 Расширение типов данных TIMESTAMP и TIME . . . 184
13.2.2 Изменения CURRENT_TIME и CURRENT_TIMESTAMP . . . 184
13.3 Сокращенное преобразование неявных литералов даты/времени не поддерживается . . . 184
13.4 Стартовое значение последовательности . . . 185
13.5 Для INSERT ... RETURNING теперь требуется привилегия SELECT . . . 185
13.6 Ограничение количества символов UNICODE_FSS . . . 185
13.7 Многострочный вывод RETURNING . . . 185
13.8 Удалён протокол WNET . . . 186
13.9 Оператор MERGE не допускает несколько совпадающих строк . . . 186
13.10 Удалена QLI . . . 186

Глава 1

Новое в СУБД РЕД База Данных 5

- *Поддержка JSON*;
- *Табличные пространства*;
- *Планировщик заданий*;
- *Блокирование сеанса пользователя*;
- *Использование BLR вместо SQL кода*;
- *Агрегатный аудит*;
- *Многопоточное выполнение бэкапа/рестора, сборки мусора и построения индексов*;
- *Частичные индексы*;
- *Предложение SKIP LOCKED для операторов SELECT WITH LOCK, UPDATE и DELETE*;
- *Обновление до последней минорной версии ODS*;
- *Кэширование скомпилированных запросов*;
- *PSQL-профайлер*;
- *Поддержка WHEN NOT MATCHED BY SOURCE в операторе MERGE*;
- *Поддержка многострочного вывода RETURNING*;
- *Улучшено сжатие записей*;
- *Поддержка прокручиваемых курсоров на уровне сетевого протокола*;
- *Согласованное чтение на уровне запросов*;
- *Изменения в логической репликации*;
- *Максимальная длина идентификаторов увеличена до 63 символов*;
- *Числа с плавающей точкой (DECFLOAT)*;
- *Увеличена точность NUMERIC и DECIMAL*;
- *Тип данных INT128*;
- *Поддержка временных зон (TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE)*;
- *Соединение с производными таблицами*;
- *Таймаут для соединений и запросов*;
- *Пул внешних подключений*;
- *Пакетные операции в API*;
- *Встроенные функции шифрования*;
- *Встроенные функции кодирования и декодирования бинарных данных*;
- *Системные привилегии*;
- *Максимальный размер страницы увеличен до 32Кб*.

Глава 2

Изменения в работе сервера

2.1 Максимальный размер страницы увеличен до 32Кб

Максимальный размер страницы для баз данных, созданных с `ODS 13`, был увеличен с 16 КБ до 32 КБ.

2.2 Поддержка международных временных зон

Поддержка временных зон в РЕД Базе Данных 5 состоит из:

- Типов данных `TIME WITH TIME ZONE` и `TIMESTAMP WITH TIME ZONE`; Существующие типы `TIME` и `TIMESTAMP` являются неявными псевдонимами для `TIME WITHOUT TIME ZONE` и `TIMESTAMP WITHOUT TIME ZONE`;
- Выражений и операторов для работы с временными зонами;
- Преобразования между типами данных с временными зонами и без них.

Типы данных `TIME WITHOUT TIME ZONE`, `TIMESTAMP WITHOUT TIME ZONE` и `DATE` определяют использование временной зоны во время конвертации в типы `TIME WITH TIME ZONE` или `TIMESTAMP WITH TIME ZONE`. `TIME` и `TIMESTAMP` являются синонимами соответствующих типов данных `WITHOUT TIME ZONE`.

2.2.1 Часовой пояс сеанса

Часовой пояс сеанса может быть разным для каждого соединения с базой данных. По умолчанию используется часовой пояс операционной системы, в которой запущен процесс сервера. Но он может быть изменен либо тегом `DPB isc_dpb_session_time_zone`, либо параметром конфигурации `DefaultTimeZone`.

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

2.2.2 Формат часового пояса

Часовой пояс - это строка, представляющая собой либо регион часового пояса (например, `'America/Sao_Paulo'`), либо смещение от `GMT` в часах:минутах (например, `'-03:00'`).

Временная метка с часовым поясом считается равной другой временной метке с часовым поясом, если их преобразования в `UTC` эквивалентны. Например, время `'10:00 -02:00'` и время `'09:00 -03:00'` эквивалентны, так как оба они равны времени `'12:00 GMT'`.

2.2.3 Типы данных для поддержки временных зон

Синтаксис типов данных `TIMESTAMP` и `TIME` был расширен и теперь включает параметр, определяющий использование временных зон.

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

По умолчанию и TIME и TIMESTAMP используют WITHOUT TIME ZONE.

2.2.4 Выражения и операторы для работы с временными зонами

Оператор SET TIME ZONE

Оператор позволяет изменить часовой пояс сеанса (текущего подключения).

```
SET TIME ZONE {'<часовой пояс>' | LOCAL}

<часовой пояс> ::=
  <регион часового пояса>
  | [+/-] <смещение часов относительно GMT> [: <смещения минут относительно GMT>]
```

Данный SQL оператор работает вне механизма управления транзакциями и вступает в силу немедленно.

```
set time zone '-02:00';
set time zone 'America/Sao_Paulo';
set time zone local;
```

Выражение AT

Преобразует время или временную метку в указанный часовой пояс.

```
<выражение> AT {TIME ZONE '<часовой пояс>' | LOCAL}

<часовой пояс> ::=
  <регион часового пояса> | [+/-] <разница часов с GMT> [:<разница минут с GMT>]
```

Если используется ключевое слово LOCAL, то преобразование происходит в часовой пояс сессии.

```
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;
```

Выражение EXTRACT

Добавлено два новых аргумента в выражение EXTRACT:

- TIMEZONE_HOUR - возвращает смещение часов часового пояса: целое число от -23 до 23;
- TIMEZONE_MINUTE - возвращает смещение минут часового пояса: целое число от -59 до 59.

```
select extract(timezone_hour from current_time) from rdb$database;
select extract(timezone_minute from current_timestamp) from rdb$database;
```

Выражение LOCALTIME

LOCALTIME типа TIME WITHOUT TIME ZONE возвращает текущее время сервера в часовом поясе сессии. В обращении к контекстной переменной LOCALTIME можно указать количество знаков в долях секунды:

```
LOCALTIME [(<количество знаков в долях секунды>)]
```

Количество знаков может быть числом от 0 до 3. Если количество знаков не указано, предполагается 0.

Выражение LOCALTIMESTAMP

Контекстная переменная LOCALTIMESTAMP типа TIMESTAMP WITHOUT TIME ZONE возвращает текущую дату и текущее время сервера в часовом поясе сессии. В текущем времени указываются миллисекунды — три знака после десятичной точки. При обращении к этой контекстной переменной можно задавать требуемое количество долей секунды:

```
LOCALTIMESTAMP [(<количество знаков в долях секунды>)]
```

Количество знаков может быть числом от 0 до 3. Если количество знаков не указано, предполагается 3.

Изменения в CURRENT_TIME и CURRENT_TIMESTAMP

Теперь CURRENT_TIME и CURRENT_TIMESTAMP возвращают TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE соответственно. В предыдущих версиях CURRENT_TIME и CURRENT_TIMESTAMP возвращали системное время без временной зоны.

2.2.5 Таблица RDB$TIME_ZONES

Отображает список часовых поясов, поддерживаемых сервером.

Идентификатор столбца	Тип данных	Описание
RDB$TIME_ZONE_ID	INTEGER	Идентификатор часового пояса.
RDB$TIME_ZONE_NAME	CHAR(63)	Наименование часового пояса.

2.2.6 Пакет RDB$TIME_ZONE_UTIL

Пакет RDB$TIME_ZONE_UTIL содержит процедуру TRANSITIONS и функцию DATABASE_VERSION для работы с часовыми поясами.

Функция DATABASE_VERSION

Функция RDB$TIME_ZONE_UTIL.DATABASE_VERSION возвращает версию базы данных часовых поясов (из библиотеки icu) типа VARCHAR(10) CHARACTER SET ASCII.

```
SELECT rdb$time_zone_util.database_version()
FROM rdb$database;

DATABASE_VERSION
================
2018g
```

Процедура TRANSITIONS

Процедура RDB$TIME_ZONE_UTIL.TRANSITIONS возвращает набор правил для часового пояса между начальной и конечной временной меткой.

Таблица 2.2 — Входные параметры процедуры TRANSITIONS

Параметр	Тип	Описание
TIME_ZONE_NAME	CHAR(63)	Наименование часового пояса
FROM_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Начало интервала дат
TO_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Окончание интервала дат

Таблица 2.3 — Выходные параметры процедуры TRANSITIONS

Параметр	Тип	Описание
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Дата начала действия правила
END_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Дата окончания действия правила
ZONE_OFFSET	SMALLINT	Смещение времени в минутах для заданного часового пояса
DST_OFFSET	SMALLINT	Летннее смещение времени в минутах для заданного часового пояса
EFFECTIVE_OFFSET	SMALLINT	Действующее смещение, вычисляется как ZONE_OFFSET + DST_OFFSET

```
SELECT
  START_TIMESTAMP,
  END_TIMESTAMP,
  ZONE_OFFSET AS ZONE_OFF,
  DST_OFFSET AS DST_OFF,
  EFFECTIVE_OFFSET AS OFF
FROM rdb$time_zone_util.transitions(
  'America/Sao_Paulo' ,
  timestamp '2017-01-01' ,
  timestamp '2019-01-01' );
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
START_TIMESTAMP              END_TIMESTAMP                ZONE_OFF DST_OFF OFF
============================ ============================ ======== ======= ====
2016-10-16 03:00:00.0000 GMT 2017-02-19 01:59:59.9999 GMT -180     60      -120
2017-02-19 02:00:00.0000 GMT 2017-10-15 02:59:59.9999 GMT -180     0       -180
2017-10-15 03:00:00.0000 GMT 2018-02-18 01:59:59.9999 GMT -180     60      -120
2018-02-18 02:00:00.0000 GMT 2018-10-21 02:59:59.9999 GMT -180     0       -180
2018-10-21 03:00:00.0000 GMT 2019-02-17 01:59:59.9999 GMT -180     60      -120
```

2.2.7 Обновление базы данных временных зон

Часовые пояса часто меняются и, когда это происходит, желательно обновить базу данных часовых поясов как можно скорее.

РЕД База Данных хранит значения `WITH TIME ZONE`, переведенные во время `UTC`. Предположим, что значение создано с одной базой данных часовых поясов, а затем обновление этой базы изменит информацию о нашем сохраненном значении. Когда это значение будет прочитано, оно будет отличаться от того, которое было сохранено изначально.

РЕД База Данных использует базу данных часовых поясов `IANA` из библиотеки `ICU`. Библиотека `ICU`, поставляющаяся вместе с РЕД Базой Данных (для `Windows`) или установленная в операционной системе `POSIX`, иногда может содержать устаревшую базу данных часовых поясов.

Обновленную базу данных можно найти на странице FirebirdSQL. Имя файла `le.zip` означает little-endian и является необходимым для большинства архитектур (`Intel/AMD` x86 или x64), в то время как `be.zip` означает big-endian и необходим в основном для архитектур `RISC`. Содержимое `zip-файла` должно быть извлечено в подкаталог `/tzdata`, с перезаписью существующих файлов `*.res`.

`/tzdata` - это каталог, в котором РЕД База Данных по умолчанию ищет базу данных часовых поясов. Его можно переопределить с помощью переменной среды `ICU_TIMEZONE_FILES_DIR`.

2.3 Репликация

В РЕД Базе Данных 5 появилась поддержка режима реплики `read-write` и изменились параметры настройки репликации.

2.3.1 Режимы доступа

Существует два режима доступа к базам данных реплики: `read-only` и `read-write`.

- В реплике, доступной только для чтения, разрешены только запросы, не изменяющие данные. Изменения ограничиваются только процессом репликации. Глобальные временные таблицы могут быть изменены, поскольку они не реплицируются.
- Реплика с режимом доступа `read-write` позволяет выполнять любые запросы. В этом режиме доступа потенциальные конфликты должны разрешаться пользователями или администраторами базы данных.

2.3.2 Изменения параметров конфигурации репликации

У имён параметров репликации префикс `"log"` изменён на `"journal"`.

Параметры настройки ведущего

Добавлены параметры:

- `sync_reconnect_timeout` - Параметр определяет таймаут для переподключения синхронной репликации в случае потери соединения. Если параметр не задан или указанный таймаут истёк, то в случае потери соединения либо возникнет ошибка, либо данная реплика будет исключена из репликации.
- `log_errors` - Логический параметр, определяющий как должны обрабатываться ошибки при репликации. Если значение параметра `true`, то все ошибки (и предупреждения) будут записываться в `replication.log`. По умолчанию включен.
- `report_errors` - Логический параметр, определяющий нужно ли возвращать ошибки при репликации клиентскому приложению. По умолчанию выключен.

Изменены параметры:

- `replica_database` - параметр заменен на `sync_replica`.

Удалены параметры:

- `compress_records`
- `master_priority`
- `alert_command`

Параметры настройки на реплике

Добавлены параметры:

- `cascade_replication` - Добавлен новый параметр настройки на реплике. Если параметр включен, то изменения, примененные к реплике, будут подлежать дальнейшей репликации (если она настроена).
- `source_guid` - `GUID` ведущей БД в формате `{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`, с которой идет репликация. Его можно узнать из вывода `GSTAT -h`.
- `apply_idle_timeout` - Время ожидания (в секундах) перед сканированием новых сегментов репликации. Он используется для приостановки сервера репликации, когда все существующие сегменты уже применены к базе данных реплики, и в указанном каталоге нет новых сегментов.
- `apply_error_timeout` - Время ожидания (в секундах) перед повторной попыткой применения сегментов в очереди после ошибки. Он используется для приостановки сервера репликации после возникновения критической ошибки во время репликации. В этом случае сервер отключается от базы данных реплики, спит в течение указанного времени ожидания, затем снова подключается и пытается повторно применить последние сегменты с момента сбоя.
- `apply_tablespaces_ddl` - Если параметр выключен, то связанные с табличными пространствами `DDL` операторы и правила внутри `CREATE TABLE, ALTER TABLE`, `CREATE INDEX` и `ALTER INDEX` не будут применены к базе данных реплики.

Изменены параметры:

- `verbose` - изменён на `verbose_logging`.
- `log_directory` - изменён на `journal_source_directory`.

Поддержка макросов

Макросы файла конфигурации теперь поддерживаются и в `replication.conf`.

2.3.3 Настройка системы репликации

Для настройки системы репликации используется файл `replication.conf`, находящийся в корневом каталоге СУБД. Файл настраивается как на стороне ведущего сервера, так и на стороне реплики. Все параметры настройки содержатся в блоке `database{ ... }`, и по умолчанию закомментированы. Настройки можно проводить глобально сразу для всех баз данных и отдельно для каждой базы данных.

При настройке параметров на уровне базы данных в разделе database должен быть указан полный путь к базе данных. Псевдонимы и подстановочные знаки не принимаются.

```
database = /your/db.fdb
{ ... }
```

2.3.4 Инициализация асинхронной репликации

1. Остановить СУБД, убедиться что не осталось процессов сервера в системе.
2. Асинхронная репликация требует настройки механизма журналирования. Основным параметром является `journal_directory`, который определяет местоположение журнала репликации. Как только это местоположение указано, асинхронная репликация включается, и сервер РЕД Базы Данных начинает создавать сегменты журнала. В `replication.conf` на ведущем прописать блок `database = /путь/к/бд`. В нем задать обязательные параметры:
   - `journal_directory` — каталог с журналами репликации;
   - `journal_archive_directory` — каталог с архивированными журналами.

   Учесть, что пользователь, от имени которого запускается СУБД, должен иметь права на запись в оба этих каталога.

   Минимальная настройка будет выглядеть так:

   ```
   database = /data/mydb.fdb
   {
     journal_directory = /dblogs/mydb/
     journal_archive_directory = /shiplogs/mydb/
   }
   ```

   Чтобы применить измененные настройки на ведущем, все пользователи должны быть переподключены.
3. В самой базе данных репликация должна быть включена с помощью DDL команды:

   ```
   ALTER DATABASE ENABLE PUBLICATION;
   ```

   А также нужно задать таблицы, которые будут реплицироваться (набор репликации), с помощью DDL команд:

   ```
   ALTER DATABASE INCLUDE ALL TO PUBLICATION; -- включение всех таблиц в
   репликацию, в том числе и тех, которые будут созданы в процессе
   ```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
ALTER DATABASE INCLUDE TABLE T1, T2, T3 TO PUBLICATION; -- включение
конкретных таблиц в репликацию
```

Чтобы исключить таблицы из процесса репликации, выполните следующие команды:

```
ALTER DATABASE EXCLUDE ALL FROM PUBLICATION; -- исключение всех таблиц из
репликации, в том числе и новых
ALTER DATABASE EXCLUDE TABLE T1, T2, T3 FROM PUBLICATION; -- исключение
указанных таблиц из репликации
```

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

4. Копировать базу данных на реплику:
   - Заблокировать базу данных:

     ```
     nbackup [-U <пользователь> -P <пароль> [-RO <роль>]] -L <база_данных>
     ```

   - Запустить СУБД на ведущей стороне.
   - Скопировать заблокированную базу данных на реплику;
   - По окончании копирования ведущей базы данных можно разблокировать:

     ```
     nbackup [-U <пользователь> -P <пароль> [-RO <роль>]] -UN <база_данных>
     ```

5. На реплике установить СУБД РЕД База Данных (рекомендуется архитектура суперклассик).
6. На реплике в `replication.conf` прописать блок `database = /путь/к/бд` реплики. В нем задать обязательные параметры:
   - `journal_source_directory` — каталог, в котором СУБД будет искать журналы для вливания. СУБД должна иметь на него права на запись;
   - `source_guid` — GUID ведущей базы.

   Пример конфигурации выглядит так:

   ```
   database = /data/mydb.fdb
   {
     journal_source_directory = /incominglogs/
     source_guid = "{6F9619FF-8B86-D011-B42D-00CF4FC964FF}"
   }
   ```

   Чтобы применить измененные настройки на стороне реплики, сервер РЕД Базы Данных должен быть перезапущен.
7. Разблокировать базу данных на реплике (копия блокированной базы данных является так же блокированной). Но не с параметром `-UN(LOCK)`, а с параметром `-F(IXUP)`:

   ```
   nbackup -F <replica_database>
   ```

8. Для базы-реплики активируется режим репликации с помощью опции `-replica` утилиты `GFIX`:

```
gfix -replica {read_only|read_write} <replica_database>
```

Если реплика доступна только для чтения, то базу данных может изменять только процесс репликации. Это главным образом предназначено для обеспечения высокой доступности, поскольку база данных реплик гарантированно совпадает с ведущей БД и может использоваться для быстрого восстановления. Обычные пользовательские соединения могут выполнять любые операции, разрешенные для транзакций только для чтения: выборку таблиц, выполнять процедуры только для чтения, записывать в глобальные временные таблицы и т.д. Также допускается обслуживание базы данных, такое как сборка мусора, выключение, мониторинг. Реплики для чтения и записи позволяют одновременно в процессе репликации подключаться обычными пользователями. В этом режиме нет гарантии, что база данных реплики будет синхронизирована с ведущей БД. Поэтому использование реплики для чтения и записи для условий высокой доступности не рекомендуется, если только пользовательские подключения на стороне реплики не ограничены изменением только таблиц, которые исключены из репликации.

9. Обеспечить доставку архивных журналов с ведущего на реплику (например, через NFS-шару).
10. Запустить СУБД на реплике.

Перевести реплику в режим штатной работы можно командой:

```
gfix -replica none <replica_database>
```

Чтобы изменения настроек репликации, сделанные на ведущем, вступили в силу, все пользователи должны быть повторно подключены. Чтобы изменение настроек репликации на реплике были применены, сервер нужно перезапустить.

Пример настройки асинхронной репликации

Для ведущего:

```
database = /repl/primary.fdb
{
  journal_directory = /repl/primary_log
  journal_file_prefix = repl_test
  journal_segment_size = 1048576 # 1MB
  journal_segment_count = 4
  journal_archive_directory = /repl/primary_arch
  journal_archive_command = "copy $(pathname) $(archivepathname)"
  journal_archive_timeout = 0
}
```

Для реплики:

```
database = /repl/replica.fdb
{
  journal_source_directory = /repl/replica_log
  source_guid = "{6F9619FF-8B86-D011-B42D-00CF4FC964FF}"
}
```

2.3.5 Инициализация синхронной репликации

1. Остановить СУБД на ведущем, убедиться что не осталось процессов сервера в системе.
2. В `replication.conf` прописать блок `database = /путь/к/бд`. В нем задать обязательный параметр подключения к реплике в виде:

```
sync_replica = [<login>:<password>@]<database connection string>
```

   Их может быть несколько.
3. В самой базе данных репликация должна быть включена с помощью DDL команды:

```
ALTER DATABASE ENABLE PUBLICATION;
```

   А также нужно задать таблицы, которые будут реплицироваться (набор репликации), с помощью DDL команд:

```
ALTER DATABASE INCLUDE ALL TO PUBLICATION; -- включение всех таблиц в
репликацию, в том числе и тех, которые будут созданы в процессе
ALTER DATABASE INCLUDE TABLE T1, T2, T3 TO PUBLICATION; -- включение
конкретных таблиц в репликацию
```

   Чтобы исключить таблицы из процесса репликации, выполните следующие команды:

```
ALTER DATABASE EXCLUDE ALL FROM PUBLICATION; -- исключение всех таблиц из
репликации, в том числе и новых
ALTER DATABASE EXCLUDE TABLE T1, T2, T3 FROM PUBLICATION; -- исключение
указанных таблиц из репликации
```

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

4. Существует два способа инициализации реплики: из физической копии и из логической копии.
   - Для инициализации реплики из физической копии необходимо скопировать базу данных на реплику, например, командой:

     ```
     scp -c arcfour
     ```

   - Для инициализации реплики из логической копии необходимо создать резервную копию базы данных:

     ```
     gbak -B <база_данных-источник> <файл резервной копии>
     ```

     При наличии триггеров, способных рассинхронизировать ведущий и реплику (например, триггеры на отключение или подключение), нужно использовать опцию -nod при создании резервной копии.

     Далее нужно восстановить базу данных из резервной копии на стороне реплики:

     ```
     gbak -C <файл резервной копии> <база_данных>
     ```

5. На реплике установить СУБД РЕД База Данных (рекомендуется архитектура классик).
6. На реплике в `replication.conf` прописать блок `database = /путь/к/бд реплики`. В нем задать рекомендуемый параметр `source_guid` — GUID ведущей базы.
7. Для базы-реплики активируется режим репликации с помощью опции `-replica` утилиты `GFIX`:

   ```
   gfix -replica {read_only|read_write} <replica_database>
   ```

   Если реплика доступна только для чтения, то базу данных может изменять только процесс репликации. Это главным образом предназначено для обеспечения высокой доступности, поскольку база данных реплики гарантированно совпадает с ведущей БД и может использоваться для быстрого восстановления. Обычные пользовательские соединения могут выполнять любые операции, разрешенные для транзакций только для чтения: выборку таблиц, выполнять процедуры только для чтения, записывать в глобальные временные таблицы и т.д. Также допускается обслуживание базы данных, такое как сборка мусора, выключение, мониторинг. Реплики для чтения и записи позволяют одновременно в процессе репликации подключаться обычными пользователями. В этом режиме нет гарантии, что база данных реплики будет синхронизирована с ведущей БД. Поэтому использование реплики для чтения и записи для условий высокой доступности не рекомендуется, если только пользовательские подключения на стороне реплики не ограничены изменением только таблиц, которые исключены из репликации.
8. Запустить СУБД на ведущем.

При необходимости можно настроить более одной синхронной репликации.

Перевести реплику в режим штатной работы можно командой:

```
gfix -replica none <replica_database>
```

Чтобы изменения настроек репликации, сделанные на ведущем, вступили в силу, все пользователи должны быть повторно подключены. Чтобы изменение настроек репликации на реплике были применены, сервер нужно перезапустить.

Пример настройки синхронной репликации

Для ведущего:

```
database = d:\temp\repl\primary.fdb
{
  sync_replica = sysdba:masterkey@otherhost:d:\temp\repl\replica.fdb
}
```

2.4 Пул внешних подключений

Чтобы избежать задержек при частом использовании внешних соединений, подсистема внешних источников данных (`EDS`) использует пул внешних подключений. Пул сохраняет неиспользуемые внешние соединения в течение некоторого времени, что позволяет избежать затрат на подключение/отключение для часто используемых строк подключения.

Как работает пул соединений:

- каждое внешнее соединение связывается с пулом при создании;

- пул имеет два списка: неиспользуемых соединений и активных соединений;
- когда соединение становится неиспользуемым (т. е. у него нет активных запросов и нет активных транзакций), то оно сбрасывается и помещается в список ожидающих (при успешном завершении сброса) или закрывается (если при сбросе произошла ошибка). Соединение сбрасывается при помощи инструкции `ALTER SESSION RESET`;
- если пул достиг максимального размера, то самое старое бездействующее соединение закрывается;
- когда РЕД База Данных просит создать новое внешнее соединение, то пул сначала ищет кандидата в списке простаивающих соединений. Поиск основан на 4 параметрах:
  - строка подключения;
  - имя пользователя;
  - пароль;
  - роль.

  Поиск чувствителен к регистру;
- если подходящее соединение найдено, то проверятся живое ли оно;
- если соединение не прошло проверку, то оно удаляется и поиск повторяется (ошибка не возвращается пользователю);
- найденное (и живое) соединение перемещается из списка простаивающих соединений в список активных соединений и возвращается вызывающему;
- если имеется несколько подходящих соединений, то будет выбрано последнее использованное;
- если нет подходящего соединения, то создаётся новое и помещается в список активных соединений;
- когда время жизни простаивающего соединения истекло, то оно удаляется из пула и закрывается.

Основные характеристики:

- отсутствие "вечных" внешних соединений;
- ограниченное количество неактивных (простаивающих) внешних соединений в пуле;
- поддерживает быстрый поиск среди соединений (по 4 параметрам указанным выше);
- пул является общим для всех внешних баз данных;
- пул является общим для всех локальных соединений, обрабатываемых данным процессом РЕД Базы Данных.

Параметры пула внешних соединений:

- время жизни соединения: временной интервал с момента последнего использования соединения, после истечении которого он будет принудительно закрыт. Параметр `ExtConnPoolLifeTime` в `firebird.conf`. По умолчанию равен 7200 секунд;
- размер пула: максимально допустимое количество незанятых соединений в пуле. Параметр `ExtConnPoolSize` в `firebird.conf`. По умолчанию равен 0, т.е. пул внешних соединений отключен.

Пулом внешних соединений, а также его параметрами можно управлять с помощью специальных операторов:

- `ALTER EXTERNAL CONNECTIONS POOL SET SIZE <размер>` – устанавливает максимальное количество бездействующих соединений;
- `ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME <значение> {SECOND | MINUTE | HOUR}` – устанавливает время жизни бездействующих соединений;
- `ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL` – закрывает все бездействующие соединения;
- `ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST` – закрывает бездействующие соединения у которых истекло время жизни.

Состояние пула внешних подключений можно узнать с помощью контекстных переменных в пространстве имен SYSTEM:

- EXT_CONN_POOL_SIZE - размер пула;
- EXT_CONN_POOL_LIFETIME - время жизни неактивных соединений;
- EXT_CONN_POOL_IDLE_COUNT- текущее количество неактивных соединений в пуле;
- EXT_CONN_POOL_ACTIVE_COUNT - текущее количество активных соединений в пуле;

2.4.1 Настройки пула внешних подключений

ExtConnPoolSize

Устанавливает максимальное количество бездействующих соединений в пуле внешних соединений. Допустимые значения от 0 до 1000. Нулевое значение означает, что пул выключен.

```
ExtConnPoolSize = 0
```

ExtConnPoolLifeTime

Устанавливает время жизни бездействующих соединений в пуле внешних соединений. Допустимые значения от 1 секунды до 24 часов (86400 секунд).

```
ExtConnPoolLifeTime = 7200
```

2.5 Таймаут для соединений и запросов

В РЕД Базе Данных существует два вида таймаутов: таймаут простоя соединения и таймаут выполнения SQL-запроса.

2.5.1 Таймаут простоя соединения

Данный функционал позволяет автоматически закрывать пользовательские подключения после определённого периода бездействия. Он может быть использован администраторами баз данных, чтобы принудительно закрывать старые неактивные соединения и освобождать связанные с ними ресурсы. Приложения и инструменты разработчика могут использовать это в качестве замены самостоятельного контроля за временем жизни подключения.

Рекомендуется устанавливать тайаут простоя в разумное большое значение, например, несколько часов. По умолчанию эта функция отключена.

Когда пользовательский вызов API завершается в ядре сервера, запускается специальный таймер, связанный с текущим подключением. Как только другой пользовательский вызов из этого подключения запускается в ядре, таймер останавливается. При превышении таймаута, сервер закроет соединение так, как будто произошла асинхронная отмена подключения:

- все активные операторы и курсоры закрываются;
- все активные транзакции откатываются;
- сетевое соединение в этот момент не закрывается. Это позволяет клиентскому приложение получить точный код ошибки при следующем вызове API. Сетевое соединение будет закрыто на стороне сервера ппосле того, как ошибка будет отправлена клиенту или, если клиентская сторона отключится по истечению таймаута сети.

Таймаут простоя соединения может быть установлен:

- На уровне базы данных. Значение параметра `ConnectionIdleTimeout` может быть установлено в `firebird.conf` (или `databases.conf`) администратором базы данных. Область действия – все пользовательские подключения, исключая системные подключения (`garbage collector`, `cache writer` и др.). Параметр `ConnectionIdleTimeout` устанавливает таймаут в минутах, по истечении которого неактивное соединение будет закрыто сервером. Ноль означает, что таймаут не установлен. Значение по умолчанию равно 0.
- На уровне подключения. Может быть установлен с использованием API (в секундах) или с помощью SQL оператора `SET SESSION IDLE TIMEOUT`. Область действия – текущее подключение.

  ```
  SET SESSION IDLE TIMEOUT <значение> [HOUR | MINUTE | SECOND]
  ```

  В качестве параметра выступает значение таймаута простоя в указанных единицах измерения времени. Если единица измерения времени не указана, то по умолчанию значение таймаута измеряется в минутах.

Действующее значение таймаута простоя соединения вычисляется каждый раз, когда пользовательский вызов API завершается в ядре сервера:

- если таймаут не установлен на уровне подключения, будет использовано значение, указанное на уровне базы данных;
- значение таймаута не может быть больше, чем значение установленное на уровне базы данных. Таким образом, значение таймаута может перекрываться разработчиком приложения для заданного подключения, но оно не может выйти за пределы, установленные в конфигурации.

Нулевой таймаут означает, что таймер ожидания не запускается.

Несмотря на то, что таймаут простоя соединения может быть установлен в секундах, абсолютная точность не гарантируется. При высокой нагрузке он может быть менее точным, но даже в таком случае таймер не сработает раньше указанного момента.

В пространстро имён `SYSTEM` добавлена контекстная переменная `SESSION_IDLE_TIMEOUT`. Она содержит текущее значение таймаута простоя соединения в секундах.

В таблицу `MON$ATTACHMENTS` добавлены новые поля:

- `MON$IDLE_TIMEOUT` - Таймаут простоя уровня соединения в секундах. Если таймаут не установлен — 0.
- `MON$IDLE_TIMER` - Значение таймера. Содержит `NULL`, если таймаут простоя соединения не установлен или таймер не запущен.

2.5.2 Таймаут выполнения SQL-запроса

Данная функция позволяет автоматически прекратить выполнение SQL оператора, если он выполняется дольше заданного значения таймаута.

Данная функция может быть полезна для:

- Администраторов баз данных, которые получают инструмент для ограничения времени выполнения запросов, которые потребляют много ресурсов;
- Разработчиков приложений, которые могут использовать таймауты SQL операторов при написании и отладке сложных запросов с неизвестным временем выполнения;
- Тестировщиков, которые могут использовать таймауты SQL операторов для обнаружения долгих запросов и обеспечения выполнения набора тестов за определённое время.

Когда начинается выполнение оператора (или открывается курсор), РЕД База Данных запускает специальный таймер. Выборка записей (`fetch`) не сбрасывает таймер. Таймер останавливается, если выполнение SQL оператора завершено или извлечена (`fetch`) последняя запись.

По истечении таймаута:

- Если выполнение SQL оператора активно, оно останавливается в ближайший подходящий момент.
- Если SQL оператор не активен в данный момент (например между выборками (`fetch`)), то он будет помечен как отменённый, следующая выборка (`fetch`) прервёт выполнение и будет возвращена ошибка

Значение таймаута может быть установлено:

- На уровне базы данных. Значение параметра `StatementTimeout` может быть установлено в `firebird.conf` (или `databases.conf`) администратором базы данных. Область действия - все операторы во всех соединениях. Параметр `StatementTimeout` устанавливает таймаут в секундах, по истечении которого выполнение SQL операторов будет отменено. Ноль означает, что таймаут не установлен. Значение по умолчанию равно 0.
- На уровне соединения. Может быть установлен с использованием API (в миллисекундах) или с помощью SQL оператора `SET STATEMENT TIMEOUT`. Область действия - текущее подключение.

```
SET STATEMENT TIMEOUT <значение> [HOUR | MINUTE | SECOND | MILLISECOND]
```

  В качестве параметра выступает значение таймаута выполнения SQL операторов в указанных единицах измерения времени. Если единица измерения времени не указана, то по умолчанию значение таймаута измеряется в секундах.
- На уровне оператора. Может быть установлен с использованием API (в миллисекундах). Область действия – текущий SQL оператор.

Действующее значение таймаута SQL оператора определяется каждый раз, когда запускается SQL оператор (открывается курсор):

- если таймаут не установлен на уровне оператора, будет использовано значение таймаута уровня соединения;
- если таймаут не установлен на уровне соединения, будет использовано значение таймаута уровня базы данных;
- значение таймаута не может быть больше, чем значение, установленное на уровне базы данных. Таким образом, значение таймаута может перекрываться на более низких уровнях, но оно не может выйти за пределы, установленные в конфигурации.

Нулевой таймаут означает, что таймер выполнения оператора не запускается.

Несмотря на то, что таймаут выполнения SQL оператора может быть установлен в миллисекундах, абсолютная точность не гарантируется. При высокой нагрузке он может быть менее точным, но даже в этом случае таймер не сработает раньше указанного времени. Клиентское приложение может ждать больше времени, чем значение таймаута, если серверу РЕД Базы Данных необходимо отменить множество действий связанных с отменой оператора.

Таймаут выполнения оператора игнорируется для всех внутренних запросов, которые используются сервером РЕД Базы Данных. Кроме того, таймаут игнорируется для DDL операторов.

В пространстро имён `SYSTEM` добавлена контекстная переменная `STATEMENT_TIMEOUT`. Она содержит текущее значение таймаута выполнения оператора в миллисекундах, который был установлен на уровне подключения.

В таблицу `MON$ATTACHMENTS` добавлено новое поле:

- `MON$STATEMENT_TIMEOUT` - Таймаут для запроса на уровне соединения в миллисекундах. Если таймаут не установлен—0.

В таблицу `MON$STATEMENTS` добавлены новые поля:

- `MON$STATEMENT_TIMEOUT` - Таймаут для текущего запроса в миллисекундах. Если таймаут не установлен—0.
- `MON$STATEMENT_TIMER` - Значение таймера SQL запроса. Содержит NULL, если таймаут не установлен или таймер не запущен.

2.6 Согласованное чтение

Традиционно транзакция SNAPSHOT при старте получает частную копию страницы инвентаризации транзакций (TIP) и использует ее для обращения к состоянию последних зафиксированных версий всех записей в базе данных, вплоть до фиксации или отката собственных изменений. Таким образом, по определению, транзакция SNAPSHOT видит состояние базы данных только таким, каким оно было в момент ее запуска.

В традиционной модели транзакция READ COMMITTED не использует стабильный снимок состояния базы данных и не хранит отдельную копию TIP. Вместо этого она запрашивает в TIP самое последнее состояние записи, зафиксированной другой транзакцией. В режиме SuperServer кэш TIP разделяется, чтобы обеспечить оптимальный доступ к нему транзакций READ COMMITTED.

2.6.1 Определение видимости записей

Используется новый подход к созданию согласованного снимка состояния базы данных, видимого для выполняющихся транзакций. Этот новый подход использует концепцию порядка коммитов.

Достаточно знать порядок коммитов, чтобы зафиксировать состояние любой транзакции на момент создания снимка.

Порядок фиксации транзакций

1. Порядковый номер коммита (Commit Number (CN)) инициализируется для каждой базы данных в момент её первого откытия.
2. При каждом подтверждении транзакции CN для базы увеличивается и новый CN связывается с определенной транзакцией.
3. Комбинация этой транзакции и CN (transaction CN) сохраняется в памяти и может быть запрошено пока база данных остается активной.
4. Снимок базы данных идентифицируется по значению, сохраненному для глобального CN в момент создания снимка базы данных.

Специальные значения CN транзакции

- CN_ACTIVE = 0 - Транзакция активна;
- CN_PREHISTORIC = 1 - Транзакция была зафиксирована до запуска базы данных (т.е. старше, чем OIT);
- CN_PREHISTORIC < CN < CN_DEAD - Транзакция была зафиксирована во время работы базы данных;
- CN_DEAD = MAX_TRA_NUM - 2 - Мертвая (dead) транзакция;
- CN_LIMBO = MAX_TRA_NUM - 1 - Транзакция в состоянии "in limbo".

Правило определения видимости записей

Если предположить, что снимок базы данных - это текущий снимок, используемый текущей транзакцией, а другая транзакция - это транзакция, создавшая данную версию записи, то правило для определения видимости версии записи работает следующим образом:

- Если состояние другой транзакции 'active', 'dead' или 'in limbo', то данная версия записи не видна текущей транзакции;
- Если состояние другой транзакции - 'committed', то видимость данной версии записи зависит от времени создания снимка базы данных, поэтому:

- если запись была зафиксирована до создания снимка базы данных, то она видна текущей транзакции;
- если она была зафиксирована после создания снимка базы данных, то она не видна текущей транзакции.

Таким образом, до тех пор, пока существует список всех известных транзакций с их `Commit Numbers`, достаточно сравнить `CN` другой транзакции с `CN` снимка базы данных, чтобы решить, должна ли данная версия записи быть видимой в рамках снимка базы данных.

Информацию о состоянии транзакции и её `CN` можно узнать с помощью новой встроенной функции `RDB$GET_TRANSACTION_CN`.

Транзакции `SNAPSHOT` теперь используют снапшот базы данных, описанный выше. Вместо того, чтобы делать копию `TIP` при запуске, они просто запоминает значение глобального `Commit Number` в этот момент.

Детали реализации

Список всех известных транзакций с соответствующими `CN` хранится в общей памяти. Он реализован в виде массива, индекс которого - идентификатор транзакции, а значение элемента - соответствующий ей `CN`.

Весь массив разбивается на блоки фиксированного размера, содержащие `CN` для всех транзакций между `OIT` и `Next Transaction`. Когда следующая транзакция выходит из области действия самого верхнего блока, выделяется новый блок. Старый блок освобождается, когда `OIT` выходит из области действия самого младшего блока.

По умолчанию размер блока кэша `TIP` составляет 4 МБ, что обеспечивает место для `512 * 1024` транзакций. Его можно настроить в файлах `firebird.conf` и `databases.conf` с помощью нового параметра `TipCacheBlockSize`.

2.6.2 Согласованное чтение для запросов в транзакциях READ COMMITTED

В реализации уровня изоляции `READ COMMITTED` существует проблема: один оператор, например `SELECT`, может видеть различные представления одних и тех же данных во время выполнения.

Например, представьте две параллельные транзакции, в которых первая вставляет 1000 строк и фиксирует, а вторая выполняет `SELECT COUNT(*)` над той же таблицей.

Если у второй транзакции уровнем изоляции является `READ COMMITTED`, то результат тяжело предсказать. Он может быть любым из перечисленных ниже:

- количество строк в таблице до старта транзакции или
- количество строк в таблице после фиксации первой транзакции или
- любое число между этими двумя значениями.

Какой из этих результатов буждет возвращён зависит от того, как эти транзакции взаимодействуют:

- Ситуация 1 произойдет, если вторая транзакция закончит подсчет до фиксации первой транзакции, поскольку незафиксированные записи в этот момент видны только первой транзакции.
- Ситуация 2 произойдет, если вторая транзакция запустится после того, как первая зафиксирует все добавленные записи.
- Ситуация 3 возникает при любых других условиях: вторая транзакция видит некоторые, но не все добавленные записи во время фиксации первой транзакции.

В третьем случае проблема возникает из-за отсутствия согласованности чтения на уровне запроса. Это важно, потому что, по определению, каждый запрос в транзакции READ COMMITTED имеет собственное представление состояния базы данных. В существующей реализации представление запроса не обязательно остается стабильным в течение всего времени его выполнения: оно может быть разным на момент начала выполнения и после завершения.

Запросы, выполняющиеся в транзакции с уровнем изоляции SNAPSHOT, не имеют такой проблемы, так как каждый запрос выполняется в согласованном представлении состояния базы данных. Кроме того, разные запросы, выполняющиеся в одной и той же транзакции READ COMMITTED, могут видеть разные представления состояния базы данных, но это не является источником несогласованности на уровне запроса.

2.6.3 Решение проблемы несогласованного чтения

Решение проблемы несогласованного чтения заключается в том, чтобы транзакция read-committed использовала стабильный снимок базы данных во время выполнения оператора. Каждый новый оператор верхнего уровня создает свой собственный снимок базы данных, в котором видны последние зафиксированные данные. С использованием снимков, основанных на порядке фиксации, это очень дешевая операция. Вложенные операторы (триггеры, вложенные хранимые процедуры и функции, динамические операторы и т. д.) используют тот же снимок уровня оператора, который был создан оператором верхнего уровня.

Уровень изоляции READ COMMITTED READ CONSISTENCY

Представлен новый уровень изоляции: READ COMMITTED READ CONSISTENCY.

Существующие опции для READ COMMITTED, такие как RECORD VERSION и NO RECORD VERSION, всё ещё поддерживаются, но они устарени и будут удалены в будущих версиях.

Доступны три варианта транзакций с уровнем изоляции READ COMMITTED:

- READ COMMITTED READ CONSISTENCY
- READ COMMITTED NO RECORD VERSION
- READ COMMITTED RECORD VERSION

Обработка конфликта обновлений

Когда оператор выполняется в транзакции с режимом изоляции READ COMMITTED READ CONSISTENCY он видит неизменное состояние базы данных (подобно транзакции SNAPSHOT). Поэтому не имеет смысла ждать подтверждения параллельной транзакции в надежде перечитать новую версию подтверждённой записи. При чтении поведение похоже на транзакцию READ COMMITTED RECORD_VERSION — оператор не ждёт завершения активной транзакции и обходит цепочку версий, в которой ищет версию записи, видимую на текущем моментальном снимке.

Для уровня изоляции READ COMMITTED READ CONSISTENCY обработка конфликтов обновлений значительно изменяется. При обнаружении конфликта обновления выполняется следующее:

1. Режим изолированности транзакции временно переключается в режим READ COMMITTED NO RECORD_VERSION.
2. Устанавливается блокировка записи на конфликтную запись.
3. Оставшиеся записи текущего курсора продолжают обрабатываться и на них также ставятся блокировки на запись
4. Когда больше нет записей для извлечения, сервер начинает отменять все действия, выполненные с момента начала выполнения оператора верхнего уровня. Он сохраняет все установленные блокировки для каждой обновлённой/удалённой/заблокированной записи, все вставленные записи удаляются.

5. Затем восстанавливается уровень изоляции транзакции `READ COMMITTED READ CONSISTENCY`, создается новый снимок уровня оператора и перезапускается выполнение оператора верхнего уровня.

Такой алгоритм гарантирует, что после перезапуска уже обновленные записи останутся заблокированными, они будут видны на новом снимке и могут быть обновлены снова без дальнейших конфликтов. Кроме того, из-за режима согласованности чтения набор измененных записей остается согласованным.

Замечания:

- Приведенный выше алгоритм перезапуска применяется к операторам `UPDATE`, `DELETE`, `SELECT WITH LOCK` и `MERGE`, с предложением `RETURNING` и без него, выполняемым непосредственно из пользовательского приложения или в составе некоторого объекта `PSQL` (хранимая процедура, функция, триггер, `EXECUTE BLOCK` и т. д.).
- Если оператор `UPDATE/DELETE` расположен на каком-либо явном курсоре (`WHERE CURRENT OF`), то пропускается шаг выше, то есть не извлекаются и не устанавливаются блокировки записи для оставшихся записей курсора.
- Если оператор верхнего уровня `SELECT` (или `EXECUTE BLOCK` возвращающий набор данных) и конфликт обновления происходит после того, как одна или несколько записей были возвращены приложению, то ошибка конфликта обновления сообщается как обычно и перезапуск не инициируется.
- Перезапуск не инициируется для операторов в автономных блоках (`IN AUTONOMOUS TRANSACTION DO ...`).
- После 10 попыток прерывается алгоритм перезапуска, снимаются все блокировки записи, восстанавливается режим изоляции транзакции как `READ COMMITTED READ CONSISTENCY` и сообщается о конфликте обновления.
- Любая необработанная ошибка на шаге выше останавливает алгоритм перезапуска и сервер продолжает работать в обычном режиме, например, ошибка может быть перехвачена и обработана `PSQL-блоком WHEN` или возвращена пользователю, если она не была обработана.
- Триггеры `UPDATE/DELETE` сработают многократно для одной и той же записи, если выполнение оператора было перезапущено и запись обновлена/удалена снова.
- По историческим причинам `isc_update_conflict` сообщается как вторичный код ошибки с основным кодом ошибки `isc_deadlock`.

Транзакции Read Committed Read-Only

В существующей реализации транзакции `READ COMMITTED` в режиме `READ ONLY` фиксируются в момент старта транзакции. Версии записей в таких транзакциях никогда не являются "интересующими", тем самым не препятствуют регулярной сборке мусора и не задерживают обновление `Oldest Snapshot Transaction`.

Транзакции `READ CONSISTENCY READ ONLY` по-прежнему запускаются как `pre-committed`, но для того, чтобы обычная сборка мусора не нарушала будущие снимки на уровне операторов, она задерживает обновление `Oldest Snapshot Transaction` так же, как это происходит для транзакций `SNAPSHOT`.

> При этом задерживается только обычная (традиционная) сборка мусора, промежуточная сборка мусора не затрагивается.

Синтаксис и конфигурация

Поддержка нового уровня изоляции READ COMMITTED READ CONSISTENCY присутствует в синтаксисе SQL, в API и в настройках конфигурации.

Если доступна функция SET TRANSACTION, новый подуровень изоляции можно установить следующим образом:

```
SET TRANSACTION READ COMMITTED READ CONSISTENCY
```

Чтобы запустить транзакцию READ COMMITTED READ CONSISTENCY через ISC API, используйте новую константу isc_tpb_read_consistency в буфере параметров транзакции (TPB).

Использование устаревших режимов READ COMMITTED (RECORD VERSION и NO RECORD VERSION) не рекомендуется, вместо них лучше использовать режим READ CONSISTENCY. Существующие приложения можно протестировать с уровнем изоляции READ COMMITTED READ CONSISTENCY, установив значение параметра конфигурации ReadConsistency.

Параметр ReadConsistency включен по умолчанию, то есть все транзакции READ COMMITTED будут выполняться в режиме READ CONSISTENCY. Отключите этот параметр, если необходимо сохранить устаревшее поведение транзакций READ COMMITTED.

2.6.4 Сборка мусора

Необходимость хранить версию записи определяется следующим образом:

- Если снимок с номером CN видит версию записи (RV_X), то все снимки с номером больше CN, тоже ее видят.
- Если все существующие снимки видят RV_X, то все её старые версии могут быть удалены
- Если самый старый активный снимок может видеть RV_X, то все её старые версии могут быть удалены.

Последняя часть правила воспроизводит традиционное правило, согласно которому все версии записей в хвосте цепочки версий начинаются с некоторой видимую всем версии записи. Правило позволяет определить эту видимую всем версию записи, чтобы можно было отсечь весь хвост после нее.

Однако при использовании снимков, основанных на порядке фиксации, цепочки версий могут быть еще более укорочены, поскольку CN позволяет идентифицировать некоторые версии записей, расположенные в промежуточных позициях цепочки версий, как мусорные. Каждая версия записи в цепочке помечается значением самого старого активного снапшота, который может ее видеть. Если несколько последовательных версий в цепочке помечены одним и тем же значением самого старого активного снапшота, то все версии, следующие за первой, могут быть удалены.

Сервер выполняет сборку мусора промежуточных версий записей во время следующих процессов:

- чистка базы данных (sweep);
- сканирование таблицы при создании индекса;
- фоновая сборка мусора в SuperServer;
- в каждом пользовательском подключении после фиксации обновленной или удаленной записи.

Чтобы это работало, сервер хранит в общей памяти массив всех активных снимков базы данных. Когда ему нужно найти самый старый активный снимок, который может видеть заданную версию записи, он просто ищет CN транзакции, которая создала эту версию записи.

По умолчанию размер этого блока общей памяти составляет 64 КБ, но при необходимости он будет автоматически увеличиваться. Размер блока можно задать в файлах firebird.conf и/или databases.conf с помощью нового параметра SnapshotsMemSize.

2.7 NUMERIC и DECIMAL

В результате реализации внутреннего 128-битного целочисленного типа данных были внесены некоторые улучшения в то, как сервер обрабатывает точность промежуточных результатов вычислений с использованием длинных типов данных NUMERIC и DECIMAL. В предыдущих версиях числа, имеющие внутренний тип данных BIGINT (т.е. с точностью от 10 до 18 десятичных цифр), умножались/делились с использованием того же типа данных BIGINT, что могло привести к ошибкам переполнения из-за ограниченной доступной точности. Теперь такие вычисления выполняются с использованием 128-битных целых чисел, что снижает вероятность непредвиденных переполнений.

2.8 Увеличено количество форматов для представлений

Представления больше не ограничены 255 форматами (версиями), прежде чем база данных потребует резервного копирования и восстановления. Новое ограничение составляет 32000 версий.

2.9 Улучшение оптимизатора для GROUP BY

Улучшение позволяет использовать индекс DESCENDING для столбца, указанного для GROUP BY.

2.10 Заменена поддержка xinetd в Linux

В Linux сервер использует один и тот же процесс сетевого слушателя для всех архитектур Для Classic главный процесс (слушатель) теперь запускается через init/systemd, подключается к порту 3050 и создает рабочий процесс для каждого соединения - аналогично тому, как это происходит в Windows.

2.11 Виртуальная таблица RDB$CONFIG

Содержит сведения о параметрах конфигурации текущей базы данных для текущего подключения. Доступна только администраторам, другие пользователи не видят записей в этой таблице.

Таблица 2.4 — RDB$CONFIG

Идентификатор столбца	Тип данных	Описание
RDB$CONFIG_ID	INTEGER	Идентификатор строки.
RDB$CONFIG_NAME	VARCHAR(63)	Название параметра конфигурации.
RDB$CONFIG_VALUE	VARCHAR(255)	Значение параметра конфигурации.
RDB$CONFIG_DEFAULT	VARCHAR(255)	Значение по умолчанию для параметра конфигурации.
RDB$CONFIG_IS_SET	BOOLEAN	TRUE, если значение настроено, FALSE, установлено значение по умолчанию.
RDB$CONFIG_SOURCE	VARCHAR(255)	Имя файла конфигурации (относительно корневого каталога), из которого был взят этот параметр, или специальное значение DPB, если параметр был указан клиентским приложением через API.

2.12 Введено ограничение на количество символов для UNICODE_FSS

Ограничение длины символов теперь применяется для UNICODE_FSS. Раньше количество символов в столбцах UNICODE_FSS не проверялось, только общая длина байт. Попытки сохранить в столбце с набором символов UNICODE_FSS значения, превышающие заявленную длину, теперь будут завершаться ошибкой "string right truncation".

2.13 Табличные пространства

Табличные пространства используются, прежде всего, для разделения и хранения определенных объектов базы данных — таких как индексы и таблицы. Табличные пространства не имеют отношения к логической структуре базы данных, а предназначены для указания места хранения данных на физических носителях. Различные объекты одной базы данных, например, индекс и таблица, могут физически храниться в разных пространствах.

Ниже приведен краткий список преимуществ использования табличных пространств:

- Позволяют расширить текущие лимиты размера базы данных, даже для небольших размеров таблицы.
- Дает возможность контролировать использование базой данных доступного места и оптимизировать быстродействие. Например, пространство, используемое для индексов, можно разместить на быстрых накопителях, а неактивную или архивную части базы данных перемещать на медленные носители большого объема.

Для создания табличного пространства понадобится оператор CREATE TABLESPACE:

```
CREATE TABLESPACE <имя табличного пространства> FILE '<путь к файлу>';
```

Права на создание табличных пространств есть только у администраторов и пользователей с привилегией CREATE TABLESPACE. Максимально доступно 254 табличных пространства.

Для создания табличного пространства необходимо указать путь к файлу. Можно указывать как абсолютный путь, так и относительный, в том числе с использованием псевдонима директории, заданного в directories.conf. В directories.conf указывается реальный путь к директории с псевдонимом <псевдоним директории>. Путь к файлу табличного пространства с использованием псевдонима директории указывается в формате: <псевдоним директории>/<файл>, например, dir/file.dat. Первый компонент пути (dir) будет обработан как псевдоним директории. Если псевдоним не будет найден в directories.conf, то путь будет обработан как относительный (относительно директории, в которой размещен основной файл базы данных). Все директории, используемые в пути, должны быть созданы заранее. В системную таблицу RDB$TABLESPACES в любом случае будет записан абсолютный путь к файлу табличного пространства.

Пример создания табличного пространства с использованием псевдонима директории:

```
CREATE TABLESPACE TS1 FILE 'dir/ts1.dat';
```

После создания табличного пространства можно перемещать таблицы или индексы в него. Чтобы создать таблицу в табличном пространстве, воспользуйтесь оператором:

```
CREATE TABLE <имя таблицы> (<столбец> [, <столбец>...]) [IN] TABLESPACE {<имя
табличного пространства> | PRIMARY};
```

Чтобы переместить существующую таблицу в табличное пространство, примените оператор:

```
ALTER TABLE <имя таблицы> SET TABLESPACE [TO] {<имя табличного пространства> |
PRIMARY};
```

Ключевое слово PRIMARY можно использовать в качестве имени табличного пространства, если необходимо ссылаться на основной файл базы данных.

Для перемещения таблицы в основной файл базы данных примените оператор:

```
ALTER TABLE <имя таблицы> SET TABLESPACE [TO] PRIMARY;
```

Операторы перемещения таблицы в табличное пространство требуют наличия единственного подключения к базе данных. Это временное ограничение, что делает процедуру перемещения более надежной.

По умолчанию индекс будет создан в том табличном пространстве, в котором располагается таблица, к которой он относится. Если вы хотели бы создать его в своем табличном пространстве, примените оператор:

```
CREATE INDEX <имя индекса> [IN] TABLESPACE {<имя табличного пространства> | PRIMARY};
```

Чтобы переместить существующий индекс в табличное пространство, примените оператор:

```
ALTER INDEX <имя индекса> SET TABLESPACE [TO] {<имя табличного пространства> |
PRIMARY};
```

Для перемещения индекса в основной файл базы данных примените оператор:

```
ALTER INDEX <имя индекса> SET TABLESPACE [TO] PRIMARY;
```

Операторы перемещения индекса в табличное пространство требуют наличия единственного подключения к базе данных. Это временное ограничение, что делает процедуру перемещения более надежной.

Можно указать табличное пространство для ограничений PK, FK, UNIQUE, созданных в CREATE TABLE и ALTER TABLE.

Для изменения путей к файлам табличных пространств используется оператор ALTER TABLESPACE:

```
ALTER TABLESPACE <имя табличного пространства> SET FILE [TO] '<путь к файлу>';
```

Эта операция не перемещает данные и не копирует файлы. Она применяется для изменения путей к табличным пространствам в случае их физического перемещения.

Для изменения путей к файлам табличных пространств необходимо указать путь к файлу. Можно указывать как абсолютный путь, так и относительный, в том числе с использованием псевдонима директории, заданного в directories.conf. В directories.conf указывается реальный путь к директории с псевдонимом <псевдоним директории>. Путь к файлу табличного пространства с использованием псевдонима директории указывается в формате: <псевдоним директории>/<файл>, например, dir/file.dat. Первый компонент пути (dir) будет обработан как псевдоним директории. Если псевдоним не будет найден в directories.conf, то путь будет обработан как относительный (относительно директории, в которой размещен основной файл базы данных). Все директории, используемые в пути,

должны быть созданы заранее. В системную таблицу RDB$TABLESPACES в любом случае будет записан абсолютный путь к файлу табличного пространства.

Права на изменение табличных пространств есть только у администраторов и пользователей с привилегией ALTER ANY TABLESPACE.

```
DROP TABLESPACE <имя табличного пространства>;
```

Если в удаляемом табличном пространстве хранятся таблицы или индексы, то будет выдано сообщение об ошибке.

Невозможно удаление табличного пространства, в котором размещено ограничение, при размещении соответствующей ему таблицы в другом табличном пространстве.

Права на удаление табличных пространств есть только у администраторов и пользователей с привилегией DROP ANY TABLESPACE.

2.14 Планировщик заданий

В СУБД РЕД Баз Данных реализован встроенный планировщик заданий для выполнения плановых работ. Планировщик имеет возможность запускать задания по расписанию (с указанием интервалов времени и дней недели) и оповещать о запуске и завершении заданий, а также об ошибках. В качестве задания может выступать блок PSQL операторов или команды операционной системы.

На данный момент планировщик заданий не работает на архитектуре Classic в Linux.

Задания сохраняются в базе данных scheduler.fdb в таблице SCH$SCHEDULED_JOBS. Путь к базе задается в файле конфигурации scheduler.conf в параметре SchedulerDatabase.

Таблица 2.5 — Описание полей таблицы SCH$SCHEDULED_JOBS

Поле	Тип	Описание
SCH$JOB_NAME	CHAR(63)	Имя задания
SCH$JOB_ID	INTEGER	ID задания
SCH$JOB_SOURCE	BLOB TEXT	Код задания в текстовом виде
SCH$JOB_BLR	BLOB BINARY	Скомпилированный BLR для задания
SCH$DESCRIPTION	BLOB TEXT	Содержит описание задания
SCH$OWNER_NAME	CHAR(63)	Имя пользователя, создавшего задание
SCH$JOB_INACTIVE	SMALLINT	Имеет два значения: 1 - задание не будет запускаться по расписанию 0 - задание будет запускаться
SCH$JOB_TYPE	SMALLINT	Имеет два значения: 0 - обычное PSQL задание 1 - задание с командой ОС
SCH$JOB_SCHEDULE	VARBINARY(64)	Строка в формате cron, задающая расписание
SCH$START_DATE	TIMESTAMP	Дата и время начала выполнения задания
SCH$END_DATE	TIMESTAMP	Дата и время окончания выполнения задания

(разрыв таблицы)

(разрыв таблицы)

Поле	Тип	Описание
SCH$DATABASE	VARCHAR(255)	Имя базы данных, для которой создано задание

События, связанные с заданиями, регистрируются в таблице SCH$JOBS_LOG (scheduler.fdb).

Таблица 2.6 — Описание полей таблицы SCH$SCHEDULED_JOBS

Поле	Тип	Описание
SCH$TIMESTAMP	TIMESTAMP	Время произошедшего события
SCH$JOB_NAME	CHAR(63)	Имя задания
SCH$JOB_ID	INTEGER	ID задания
SCH$EVENT	VARCHAR(32)	Регистрируются следующие типы событий: RUN_START - начало выполнения задания; RUN_FINISH - завершение выполнения задания; RUN_ERROR - ошибка во время выполнения задания.
SCH$MESSAGE	VARCHAR(1023)	Текст ошибки

Так как к базе данных scheduler.fdb не имеет права подключаться никто, то для всех подключений доступны соответствующие виртуальные системные таблицы RDB$JOBS и RDB$JOBS_LOG (аналогичные таблицам SCH$SCHEDULED_JOBS и SCH$JOBS_LOG). В этих таблицах обычные пользователи видят информацию только о тех заданиях, которые они создали для любой базы данных; SYSDBA видит все задания для всех баз данных.

Каждое задание выполняется в отдельном подключении от имени пользователя, который его создал. Если необходимо прервать выполнение задания, то нужно удалить соответствующую строку из системной таблицы MON$ATTACHMENTS. Сведения об имени выполняемого задания можно увидеть в поле MON$CLIENT_VERSION.

Критические ошибки, при которых работа планировщика невозможна (например, отсутствие или повреждение базы данных scheduler.fdb), записываются в firebird.log.

2.14.1 Создание задания

Оператор создания задания становится доступным после подключения к базе данных с правами администратора или привилегией CREATE JOB.

```
CREATE JOB <имя_задания>
<параметры расписания>
[ACTIVE | INACTIVE]
[START DATE '<timestamp>' | NULL]
[END DATE '<timestamp>' | NULL]
{AS
   <объявления переменных>
BEGIN
   <блок sql-операторов>
END
| COMMAND '<команда>' }
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<параметры расписания> ::= '<Минуты> <Часы> <Дни_месяца> <Месяцы> <Дни_недели>'
```

Имя задания должно быть уникальным среди всех баз данных.

Ключевое слово `ACTIVE` (по умолчанию) указывает, что задание будет запускаться по расписанию. Ключевое слово `INACTIVE` означает, что задание запускаться не будет.

`Параметры расписания` задаются в виде строки в формате, аналогичном `cron`.

Необязательные предложения `START DATE` и `END DATE` задают промежуток времени, в котором будет запускаться задание, в формате `ДД.ММ.ГГГГ ЧЧ:ММ`. Указанные значения включаются в интервал выполнения. Значение `NULL` используется, чтобы убрать интервал.

Предложение `COMMAND` дает возможность запланировать выполнение команды ОС. Такой тип заданий может создавать только `SYSDBA`. Чтобы задать список программ, которые можно использовать в команде, необходимо настроить параметр `JobCommandAccess` в `scheduler.conf`. В значении параметра можно указывать как пути к конкретным программам, так и директории с программами. По умолчанию разрешено использовать только программы из директории СУБД.

Задание выполняется от имени пользователя, который его создал.

2.14.2 Изменение задания

Синтаксис изменения задания аналогичен его созданию:

```
ALTER JOB <имя_задания>
[<параметры расписания>]
[ACTIVE | INACTIVE]
[START DATE '<timestamp>' | NULL]
[END DATE '<timestamp>' | NULL]
[AS
   <объявления переменных>
BEGIN
   <блок sql-операторов>
END
| COMMAND '<команда>' ]

<параметры расписания> ::= '<Минуты> <Часы> <Дни_месяца> <Месяцы> <Дни_недели>'
```

Изменить задание может его создатель и `SYSDBA`.

2.14.3 Удаление задания

Оператор удаления задания доступен создателю задания и `SYSDBA`.

```
DROP JOB <имя_задания>;
```

2.14.4 Оповещения

Для настройки оповещений о статусе заданий создан параметр `JobNotificationCommand` в `scheduler.conf`, в котором можно указать команду ОС. В команде можно использовать следующие переменные со сведениями о произошедшем событии:

- `$(timestamp)` – дата и время события;
- `$(job_name)` – имя задания;

- `$(job_id)` – ID задания;
- `$(event)` – тип события: `RUN_START, RUN_FINISH` или `RUN_ERROR`;
- `$(message)` – сообщение с описанием ошибки.

```
Linux: echo $(timestamp) $(job_name) $(job_id) $(event) $(message) »
/home/user/notifications.txt

Windows: echo $(timestamp) $(job_name) $(job_id) $(event) $(message) »
C:\RDB\notifications.txt
```

2.15 Блокирование сеанса пользователя

Оператор `SUSPEND` позволяет заблокировать сеансы пользователей и ролей:

```
SUSPEND [USER <список пользователей>] | [ROLE <список ролей>]
[DISCONNECT] | [PERMANENT];
<список пользователей> ::= "<имя пользователя 1>"[, "<имя пользователя 2>"...]
<список ролей> ::= "<имя роли 1>"[, "<имя роли 2>"...]
```

Заблокировать сеанс для списка пользователей или ролей может только `SYSDBA`, владелец базы данных, пользователь с ролью `RDB$ADMIN`. Заблокировать собственный сеанс может любой пользователь.

Заблокированный пользователь будет получать ошибку `"Connection suspended"` на все запросы, кроме повторного подключения к базе данных, отключения от базы данных или создания нового подключения к базе данных.

Установка параметров `DISCONNECT` или `PERMANENT` немедленно отменяет активные транзакции и завершает подключение указанных пользователей. Параметр `PERMANENT` делает указанных пользователей неактивными и без повторной активации они не смогут подключаться к базам данных.

```
SUSPEND USER TEST_USER
```

Параметр конфигурации `ConnectionSuspendTimeout` позволяет установить для всего сервера или отдельно для базы данных количество секунд, по истечении которых сессия бездействующего пользователя будет заблокирована. Изменить значение таймаута для текущего подключения можно с помощью `isc_dbp_suspend_timeout` в параметрах подключения. Установить таймаут больше определенного конфигурацией может только `SYSDBA`, владелец базы данных, пользователь с ролью `RDB$ADMIN`.

Для возобновления доступа необходимо использовать функцию `reconnect()` или `isc_reconnect()` интерфейса `IAttachment`, передав все данные, необходимые для повторной авторизации. В случае использования доверенной аутентификации указывать авторизационные данные не требуется.

Пример использования функции при возобновлении доступа в утилите `ISQL`:

```
RECONNECT [ [PASSWORD '<пароль>'] | [CERTIFICATE '<алиас_сертификата>' [PIN <пароль_
закрытого_ключа>] ] ];
```

Для возобновления выполнения транзакций/запросов заблокированный пользователь должен повторно ввести свои авторизационные данные. Это можно сделать с помощью оператора `RECONNECT`. В случае успешного ввода пароля, все незавершенные до момента блокировки транзакции будут продолжены.

Если пользовательское подключение закрыто вместо блокировки или после неё, то все незавершенные транзакции отменятся. Для возобновления работы пользователю необходимо выполнить под-

ключение к базе данных со всеми своими авторизационными данными. Это можно сделать с помощью оператора CONNECT.

2.16 Использование BLR вместо SQL кода

В РЕД Базе Данных 5 появилась возможность создать процедуру с телом в виде двоичного BLR представления, закодированного в Base64, вместо обычного исходного кода на языке SQL. В этом случае исходные SQL коды процедуры будут недоступны. Оператор создания такой процедуры будет выглядеть так:

```
CREATE PROCEDURE <имя> (<входные параметры>)
RETURNS (<выходные параметры>)
[SQL SECURITY {DEFINER | INVOKER}]
AS '<BLR код>';
```

Здесь BLR код - это BLR в его исходном двоичном виде, закодированный в Base64. Его можно получить, например, из запроса:

```
select BASE64_ENCODE(RDB$PROCEDURE_BLR)
from RDB$PROCEDURES
where RDB$PROCEDURE_NAME = '<имя>';
```

2.17 Поддержка многопоточной работы

РЕД База Данных может выполнять некоторые задачи, используя несколько потоков параллельно. Часть этих задач использует многопоточность на уровне ядра, другие реализованы непосредственно в утилитах.

Для обработки задачи с несколькими потоками сервер РЕД Базы Данных запускает дополнительные рабочие потоки и создает внутренние рабочие соединения.

По умолчанию параллельное выполнение отключено. Существует два способа включить его в пользовательском соединении:

- Установить количество параллельных рабочих процессов в DPB, используя тег isc_dpb_parallel_workers;
- Установить количество параллельных рабочих процессов по умолчанию с помощью параметра ParallelWorkers в firebird.conf.

Некоторые утилиты (gfix, gbak, gstat), поставляемые с РЕД Базой Данных, имеют ключ -parallel для установки количества параллельных рабочих потоков.

Для управления количеством рабочих потоков используйте параметры MaxParallelWorkers и ParallelWorkers.

Внутренние рабочие соединения создаются и управляются сервером. Сервер поддерживает пулы рабочих потоков для каждой базы данных. Количество потоков в каждом пуле ограничивается значением параметра MaxParallelWorkers. Пулы создаются каждым процессом РЕД Базы Данных независимо.

В архитектуре SuperServer рабочие потоки реализованы как легковесные системные подключения, а в Classic и SuperClassic они выглядят, как обычные пользовательские подключения. Все рабочие потоки встраиваются в создавший их серверный процесс. Таким образом, в архитектуре Classic нет дополнительных серверных процессов. Рабочие подключения отображаются в таблицах мониторинга. Неактивные рабочие потоки уничтожаются через 60 секунд бездействия. Также в архитектуре Classic рабочие потоки уничтожаются сразу после того, как последнее пользовательское подключение отсоединяется от базы данных.

2.18 Обновление до последней минорной версии ODS

Новая опция `gfix -upgrade` обновляет базу данных до последней минорной версии `ODS` в пределах поддерживаемой мажорной версии.

> Обновление должно быть выполнено вручную, используя команду `gfix -upgrade`. Команда должна выполняться в эксклюзивном режиме.
>
> Для выполнения требуется системная привилегия `USE_GFIX_UTILITY`. Если обновление завершится с ошибками, то все изменения будут отменены.
>
> После обновления РЕД База Данных 3 больше не сможет открыть базу данных.

```
gfix -upgrade <database>
```

2.19 Больше подробностей о курсоре в плане запроса

В выводе подробного плана теперь различаются определяемые пользователем операторы `SELECT` (сообщаемые как `select expression`), объявленные `PSQL` курсоры и подзапросы (`subquery`). И обычный и расширенный план запроса теперь включают информацию о позиции курсора (строка/столбец) внутри своего PSQL модуля.

Обычный план:

```
-- line 23, column 2
PLAN (DISTRICT INDEX (DISTRICT_PK))
-- line 28, column 2
PLAN JOIN (CUSTOMER INDEX (CUSTOMER_PK), WAREHOUSE INDEX(WAREHOUSE_PK))
```

Расширенный план:

```
Select Expression (line 23, column 2)
 -> Singularity Check
     -> Filter
         -> Table "DISTRICT" Access By ID
             -> Bitmap
                 -> Index "DISTRICT_PK" Unique Scan
Select Expression (line 28, column 2)
   -> Singularity Check
      -> Nested Loop Join (inner)
            -> Filter
               -> Table "CUSTOMER" Access By ID
                  -> Bitmap
                        -> Index "CUSTOMER_PK" Unique Scan
            -> Filter
               -> Table "WAREHOUSE" Access By ID
                  -> Bitmap
                        -> Index "WAREHOUSE_PK" Unique Scan
```

Номер строки и столбца нельзя увидеть в плане пользовательского SQL запроса, если это не запрос в `EXECUTE BLOCK`. Однако, информацию о них можно узнать в поле `MON$EXPLAINED_PLAN` таблицы `MON$STATEMENTS` или `MON$COMPILED_STATEMENTS`.

2.20 Сжатие записей

Начиная с `ODS 13.1`, сервер использует усовершенствованный метод сжатия `RLE` (со счетчиком переменной длины), который более эффективно сохраняет повторяющиеся последовательности байтов, тем самым снижая избыточные затраты на хранение. Это улучшает сжатие для длинных полей `VARCHAR` (особенно в кодировке `UTF8`), которые заполнены лишь частично.

2.21 Кэширование скомпилированных запросов

Теперь сервер поддерживает кэширование скомпилированных запросов для каждого подключения. По умолчанию кэширование включено. Максимальный размер кэша определяется параметром `MaxStatementCacheSize` в `firebird.conf`. Отключить кэширование можно установив `MaxStatementCacheSize = 0`.

Кэширование выполняется автоматически; кэшированные запросы удаляются при необходимости (обычно, когда выполняется какой-нибудь DDL оператор).

> Запрос считается одинаковым, если он совпадает с точностью до символа, то есть если у вас семантические одинаковые запросы, но они отличаются комментарием, то для кэша подготовленных запросов это разные запросы.

Помимо запросов верхнего уровня в кэш подготовленных запросов попадают также хранимые процедуры, функции и триггеры. Содержимое кэша компилированных запросов можно посмотреть с помощью новой таблицы мониторинга `MON$COMPILED_STATEMENTS`.

2.22 Поддержка двунаправленных курсоров в сетевом протоколе

До версии 5 прокручиваемые курсоры не поддерживались на уровне сетевого протокола. Это означает, что использовать API двунаправленных курсоров можно, только если подключение происходит в `embedded` режиме. Начиная с РЕД Базы Данных 5 можно использовать API прокручиваемых курсоров при подключении по сетевому протоколу, при этом клиентская библиотека `fbclient` должна быть не ниже версии 5.

2.23 PSQL-профайлер

Профайлер позволяет пользователям измерять затраты на производительность кода SQL и PSQL. Это реализовано с помощью системного пакета, передающего данные в плагин профайлера. В этом тексте части сервера и плагина рассматриваются как единое целое, так же, как будет использоваться профайлер по умолчанию (`Default_Profiler`).

Пакет `RDB$PROFILER` позволяет профилировать выполнение кода PSQL, собирая статистику о том сколько раз была выполнена каждая строка, а также ее минимальное, максимальное и общее время выполнения (с точностью до наносекунд). Также он дает доступ к статистике явных и неявных SQL-курсоров.

Чтобы собрать данные, необходимо запустить сеанс профайлера с помощью `RDB$PROFILER.START_SESSION`. Эта функция возвращает идентификатор сеанса профайлера, который позже сохраняется в таблицах снапшотов профайлера для запроса и анализа. Сеанс профайлера может быть локальным (то же подключение) или удаленным (другое подключение).

Удаленное профилирование пересылает команды удаленному подключению. Возможно одновременно профилировать несколько подключений. Также возможно, что локально или удаленно запущен-

ный сеанс профайлера содержит команды, переданные другим подключением.

Удаленно переданные команды требуют, чтобы целевое подключение находилось в состоянии ожидания, то есть не выполняло другие запросы. Когда оно не простаивает, вызов команд блокируется в ожидании этого состояния.

Если удаленное подключение происходит от другого пользователя, то вызывающий пользователь должен иметь системные привилегии `PROFILE_ANY_ATTACHMENT`.

После запуска сеанса статистика PSQL и SQL операторов начинает собираться в памяти. Обратите внимание, что сеанс профайлера собирает данные только об операторах, выполняемых в подключении, связанном с сеансом.

Данные агрегируются и сохраняются для каждого запроса. При запросе таблиц снапшотов можно выполнять дополнительную агрегацию для каждого оператора или использовать вспомогательные представления, которые делают это автоматически.

Сеанс может быть приостановлен, чтобы временно отключить сбор статистики. Он может быть возобновлен позже, чтобы вернуть сбор статистики в том же сеансе.

Новый сеанс может быть запущен, когда сеанс уже активен. В этом случае он имеет ту же семантику завершения текущего сеанса с помощью `RDB$PROFILER.FINISH_SESSION(FALSE)`, поэтому таблицы снапшотов не обновляются в то же время.

Чтобы проанализировать собранные данные, необходимо сбросить данные в таблицы снапшотов, что может быть сделано при завершении или приостановке сеанса (с параметром `FLUSH`, установленным в `TRUE`) или вызовом `RDB$PROFILER.FLUSH`. Данные удаляются с помощью автономной транзакции (транзакция, запущенная и завершенная для конкретной цели обновления данных профайлера).

Пример сессии профайлера:

1. Подготовка - создание таблиц и операций, которые будут анализироваться.

```
create table tab (
id integer not null,
val integer not null
);

set term !;

create or alter function mult(p1 integer, p2 integer) returns integer
as
begin
   return p1 * p2;
end!

create or alter procedure ins
as
   declare n integer = 1;
begin
   while (n <= 1000)
   do
   begin
      if (mod(n, 2) = 1) then
            insert into tab values (:n, mult(:n, 2));
      n = n + 1;
   end
end!

set term ;!
```

2. Старт сессии профайлера.

```
select rdb$profiler.start_session('Profile Session 1') from rdb$database;

set term !;

execute blockgroup by
as
begin
   execute procedure ins;
   delete from tab;
end!

set term ;!

execute procedure rdb$profiler.finish_session(true);

execute procedure ins;

select rdb$profiler.start_session('Profile Session 2') from rdb$database;

select mod(id, 5),
      sum(val)
from tab
where id <= 50
group by mod(id, 5)
order by sum(val);

execute procedure rdb$profiler.finish_session(true);
```

3. Анализ данных

```
set transaction read committed;

select * from plg$prof_sessions;

select * from plg$prof_psql_stats_view;

select * from plg$prof_record_source_stats_view;

select preq.*
from plg$prof_requests preq
join plg$prof_sessions pses
   on pses.profile_id = preq.profile_id and
      pses.description = 'Profile Session 1';

select pstat.*
from plg$prof_psql_stats pstat
join plg$prof_sessions pses
   on pses.profile_id = pstat.profile_id and
      pses.description = 'Profile Session 1'
order by pstat.profile_id,
         pstat.request_id,
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
          pstat.line_num,
          pstat.column_num;

select pstat.*
from plg$prof_record_source_stats pstat
join plg$prof_sessions pses
   on pses.profile_id = pstat.profile_id and
      pses.description = 'Profile Session 2'
order by pstat.profile_id,
          pstat.request_id,
          pstat.cursor_id,
          pstat.record_source_id;
```

2.23.1 Функция START_SESSION

RDB$PROFILER.START_SESSION запускает новый сеанс профайлера, превращает его в текущий сеанс (с заданным ATTACHMENT_ID) и возвращает его идентификатор.

Если значение FLUSH_INTERVAL не NULL, то автосброс настраивается таким же образом, как и при вызове RDB$PROFILER.SET_FLUSH_INTERVAL.

Если значение PLUGIN_NAME равно NULL (по умолчанию), то будет использован плагин профайлера конфигурации базы данных по умолчанию.

PLUGIN_OPTIONS – это параметры, настраиваемые для плагина, в данном случае для Default_Profiler должно быть значение NULL.

Входные параметры:

- DESCRIPTION – тип VARCHAR(255) CHARACTER SET UTF8 по умолчанию NULL;
- FLUSH_INTERVAL – тип INTEGER по умолчанию NULL;
- ATTACHMENT_ID – тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION;
- PLUGIN_NAME – тип VARCHAR(255) CHARACTER SET UTF8 по умолчанию NULL;
- PLUGIN_OPTIONS – тип VARCHAR(255) CHARACTER SET UTF8 по умолчанию NULL.

Возвращает тип BIGINT NOT NULL.

2.23.2 Процедура PAUSE_SESSION

RDB$PROFILER.PAUSE_SESSION приостанавливает текущий сеанс профайлера (с заданным ATTACHMENT_ID), поэтому статистика следующих выполненных инструкций не собирается.

Если значение FLUSH равно TRUE, таблицы снапшотов обновляются данными до текущего момента. В противном случае данные остаются только в памяти для последующего обновления.

Вызов RDB$PROFILER.PAUSE_SESSION(TRUE) имеет ту же семантику, что и вызов RDB$PROFILER.PAUSE_SESSION(FALSE), за которым следует RDB$PROFILER.FLUSH (используя тот же ATTACHMENT_ID).

Входные параметры:

- FLUSH тип BOOLEAN NOT NULL по умолчанию FALSE;
- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.3 Процедура RESUME_SESSION

RDB$PROFILER.RESUME_SESSION возобновляет текущий сеанс профайлера (с заданным ATTACHMENT_ID), если он был приостановлен, запуская снова сбор статистики.

Входные параметры:

- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.4 Процедура FINISH_SESSION

RDB$PROFILER.FINISH_SESSION завершает текущий сеанс профайлера (с заданным ATTACHMENT_ID). Если значение FLUSH равно TRUE, таблицы снапшотов обновляются данными завершенного сеанса (и старых завершенных сеансов, которые еще не присутствуют в снапшоте). В противном случае данные остаются только в памяти для последующего обновления.

Вызов RDB$PROFILER.FINISH_SESSION(TRUE) имеет ту же семантику, что и вызов RDB$PROFILER.FINISH_SESSION(FALSE), за которым следует RDB$PROFILER.FLUSH (используя тот же ATTACHMENT_ID).

Входные параметры:

- FLUSH тип BOOLEAN NOT NULL по умолчанию TRUE;
- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.5 Процедура CANCEL_SESSION

RDB$PROFILER.CANCEL_SESSION отменяет текущий сеанс профайлера (с заданным ATTACHMENT_ID). Все данные сеанса, присутствующие в плагине профайлера, удаляются и не будут сброшены. Данные, которые уже были сброшены, не удаляются автоматически.

Входные параметры:

- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.6 Процедура DISCARD

RDB$PROFILER.DISCARD удаляет все сеансы (с заданным ATTACHMENT_ID) из памяти, не сбрасывая их данные. Если есть активный сеанс, он завершается.

Входные параметры:

- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.7 Процедура FLUSH

RDB$PROFILER.FLUSH обновляет таблицы снапшотов данными из сеансов профайлера (с заданным ATTACHMENT_ID) в памяти. После обновления данные сохраняются в таблицах PLG$PROF_SESSIONS, PLG$PROF_STATEMENTS, PLG$PROF_RECORD_SOURCES, PLG$PROF_REQUESTS, PLG$PROF_PSQL_STATS и PLG$PROF_RECORD_SOURCE_STATS и могут быть прочитаны и проанализированы.

Данные обновляются с помощью автономной транзакции, поэтому, если процедура вызывается в снапшот-транзакции, данные не будут считываться в той же транзакции.

Как только происходит обновление, завершенные сеансы удаляются из памяти.

Входные параметры:

- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.8 Процедура SET_FLUSH_INTERVAL

RDB$PROFILER.SET_FLUSH_INTERVAL включает (FLUSH_INTERVAL > 0) или выключает (FLUSH_INTERVAL = 0) периодический сброс данных. FLUSH_INTERVAL – количество секунд.

Входные параметры:

- FLUSH_INTERVAL тип INTEGER NOT NULL;
- ATTACHMENT_ID тип BIGINT NOT NULL по умолчанию CURRENT_CONNECTION.

2.23.9 Таблицы снапшотов

Таблицы снапшотов (а также представления и последовательность) автоматически создаются при первом использовании профайлера. Они принадлежат текущему пользователю с правами на чтение/запись для PUBLIC. Когда сеанс завершается, связанные данные в других таблицах снапшотов профайлера также автоматически удаляются с помощью внешних ключей с опцией DELETE CASCADE.

Ниже приведен список таблиц, в которых хранятся данные профайлера.

Таблица 2.7 — PLG$PROF_SESSIONS

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
ATTACHMENT_ID	BIGINT	Идентификатор подключения
USER_NAME	CHAR(63) CHARACTER SET UTF8	Имя пользователя.
DESCRIPTION	VARCHAR(255) CHARACTER SET UTF8	Описание, переданное в RDB$PROFILER.START_SESSION.
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент запуска сессии профайлера.
FINISH_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент, когда сессия профайлера была завершена (NULL, если не завершена)

Таблица 2.8 — PLG$PROF_STATEMENTS

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
STATEMENT_ID	BIGINT	Идентификатор оператора.
PARENT_STATEMENT_ID	BIGINT	Идентификатор родительского оператора - связан с подпрограммами.
STATEMENT_TYPE	VARCHAR(20) CHARACTER SET UTF8	Блок, функция, процедура или триггер.
PACKAGE_NAME	CHAR(63) CHARACTER SET UTF8	Пакет функции или процедуры.

(разрыв таблицы)

(разрыв таблицы)

Идентификатор столбца	Тип данных	Описание
ROUTINE_NAME	CHAR(63) CHARACTER SET UTF8	Имя функции, процедуры или триггера.
SQL_TEXT	BLOB	SQL-текст для BLOCK.

Таблица 2.9 — PLG$PROF_CURSORS

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
STATEMENT_ID	BIGINT	Идентификатор оператора.
CURSOR_ID	INTEGER	Идентификатор курсора.
NAME	CHAR(63) CHARACTER SET UTF8	Имя явного курсора.
LINE_NUM	INTEGER	Номер строки оператора
COLUMN_NUM	INTEGER	Номер столбца оператора.

Таблица 2.10 — PLG$PROF_RECORD_SOURCES

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
STATEMENT_ID	BIGINT	Идентификатор оператора.
CURSOR_ID	BIGINT	Идентификатор курсора.
RECORD_SOURCE_ID	BIGINT	Идентификатор источника записи.
PARENT_RECORD_SOURCE_ID	BIGINT	Идентификатор источника родительской записи.
ACCESS_PATH	VARCHAR(255) CHARACTER SET UTF8	План запроса.

Таблица 2.11 — PLG$PROF_PSQL_STATS

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
REQUEST_ID	BIGINT	Идентификатор запроса.
LINE_NUM	INTEGER	Номер строки оператора.
COLUMN_NUM	INTEGER	Номер столбца оператора.
STATEMENT_ID	BIGINT	Идентификатор оператора.
COUNTER	BIGINT	Количество выполненных раз строки/столбца.
MIN_ELAPSED_TIME	BIGINT	Минимальное затраченное время (в наносекундах) выполнения строки/столбца.

(разрыв таблицы)

(разрыв таблицы)

Идентификатор столбца	Тип данных	Описание
MAX_ELAPSED_TIME	BIGINT	Максимальное затраченное время (в наносекундах) выполнения строки/столбца.
TOTAL_ELAPSED_TIME	BIGINT	Общее затраченное время (в наносекундах) выполнения строки/столбца.

Таблица 2.12 — PLG$PROF_RECORD_SOURCE_STATS

Идентификатор столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сеанса профайлера.
REQUEST_ID	BIGINT	Идентификатор запроса.
CURSOR_ID	BIGINT	Идентификатор курсора.
RECORD_SOURCE_ID	BIGINT	Идентификатор источника записи.
STATEMENT_ID	BIGINT	Идентификатор оператора.
OPEN_COUNTER	BIGINT	Количество раз открытия источника записи.
OPEN_MIN_ELAPSED_TIME	BIGINT	Минимальное время открытия источника записи (в наносекундах).
OPEN_MAX_ELAPSED_TIME	BIGINT	Максимальное время открытия источника записи (в наносекундах).
OPEN_TOTAL_ELAPSED_TIME	BIGINT	Общее время открытия источника записи (в наносекундах).
FETCH_COUNTER	BIGINT	Количество выборок из источника записи.
FETCH_MIN_ELAPSED_TIME	BIGINT	Минимальное время выборки из источника записи (в наносекундах).
FETCH_MAX_ELAPSED_TIME	BIGINT	Максимальное время выборки из источника записи (в наносекундах).
FETCH_TOTAL_ELAPSED_TIME	BIGINT	Общее время выборки из источника записи (в наносекундах).

2.23.10 Дополнительные представления

Эти представления помогают извлекать данные, агрегированные на уровне операторов. Они должны быть предпочтительным способом анализа собранных данных. Их также можно использовать вместе с таблицами для получения дополнительных данных, отсутствующих в представлениях.

PLG$PROF_STATEMENT_STATS_VIEW

```
select req.profile_id,
    req.statement_id,
    sta.statement_type,
    sta.package_name,
    sta.routine_name,
    sta.parent_statement_id,
    sta_parent.statement_type parent_statement_type,
    sta_parent.routine_name parent_routine_name,
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
     (select sql_text
        from plg$prof_statements
        where profile_id = req.profile_id and
              statement_id = coalesce(sta.parent_statement_id, req.statement_id)
     ) sql_text,
     count(*) counter,
     min(req.total_elapsed_time) min_elapsed_time,
     max(req.total_elapsed_time) max_elapsed_time,
     cast(sum(req.total_elapsed_time) as bigint) total_elapsed_time,
     cast(sum(req.total_elapsed_time) / count(*) as bigint) avg_elapsed_time
  from plg$prof_requests req
  join plg$prof_statements sta
     on sta.profile_id = req.profile_id and
        sta.statement_id = req.statement_id
  left join plg$prof_statements sta_parent
     on sta_parent.profile_id = sta.profile_id and
        sta_parent.statement_id = sta.parent_statement_id
  group by req.profile_id,
           req.statement_id,
           sta.statement_type,
           sta.package_name,
           sta.routine_name,
           sta.parent_statement_id,
           sta_parent.statement_type,
           sta_parent.routine_name
  order by sum(req.total_elapsed_time) des
```

PLG$PROF_PSQL_STATS_VIEW

```
select pstat.profile_id,
    pstat.statement_id,
    sta.statement_type,
    sta.package_name,
    sta.routine_name,
    sta.parent_statement_id,
    sta_parent.statement_type parent_statement_type,
    sta_parent.routine_name parent_routine_name,
    (select sql_text
       from plg$prof_statements
       where profile_id = pstat.profile_id and
             statement_id = coalesce(sta.parent_statement_id, pstat.statement_id)
    ) sql_text,
    pstat.line_num,
    pstat.column_num,
    cast(sum(pstat.counter) as bigint) counter,
    min(pstat.min_elapsed_time) min_elapsed_time,
    max(pstat.max_elapsed_time) max_elapsed_time,
    cast(sum(pstat.total_elapsed_time) as bigint) total_elapsed_time,
    cast(sum(pstat.total_elapsed_time) / nullif(sum(pstat.counter), 0) as bigint) avg_
elapsed_time
from plg$prof_psql_stats pstat
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  join plg$prof_statements sta
     on sta.profile_id = pstat.profile_id and
        sta.statement_id = pstat.statement_id
  left join plg$prof_statements sta_parent
     on sta_parent.profile_id = sta.profile_id and
        sta_parent.statement_id = sta.parent_statement_id
  group by pstat.profile_id,
           pstat.statement_id,
           sta.statement_type,
           sta.package_name,
           sta.routine_name,
           sta.parent_statement_id,
           sta_parent.statement_type,
           sta_parent.routine_name,
           pstat.line_num,
           pstat.column_num
  order by sum(pstat.total_elapsed_time) desc
```

PLG$PROF_RECORD_SOURCE_STATS_VIEW

```
select rstat.profile_id,
    rstat.statement_id,
    sta.statement_type,
    sta.package_name,
    sta.routine_name,
    sta.parent_statement_id,
    sta_parent.statement_type parent_statement_type,
    sta_parent.routine_name parent_routine_name,
    (select sql_text
       from plg$prof_statements
       where profile_id = rstat.profile_id and
             statement_id = coalesce(sta.parent_statement_id, rstat.statement_id)
    ) sql_text,
    rstat.cursor_id,
    cur.name cursor_name,
    cur.line_num cursor_line_num,
    cur.column_num cursor_column_num,
    rstat.record_source_id,
    recsrc.parent_record_source_id,
    recsrc.level,
    recsrc.access_path,
    cast(sum(rstat.open_counter) as bigint) open_counter,
    min(rstat.open_min_elapsed_time) open_min_elapsed_time,
    max(rstat.open_max_elapsed_time) open_max_elapsed_time,
    cast(sum(rstat.open_total_elapsed_time) as bigint) open_total_elapsed_time,
    cast(sum(rstat.open_total_elapsed_time) / nullif(sum(rstat.open_counter), 0) as
bigint) open_avg_elapsed_time,
    cast(sum(rstat.fetch_counter) as bigint) fetch_counter,
    min(rstat.fetch_min_elapsed_time) fetch_min_elapsed_time,
    max(rstat.fetch_max_elapsed_time) fetch_max_elapsed_time,
    cast(sum(rstat.fetch_total_elapsed_time) as bigint) fetch_total_elapsed_time,
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    cast(sum(rstat.fetch_total_elapsed_time) / nullif(sum(rstat.fetch_counter), 0) as
bigint) fetch_avg_elapsed_time,
    cast(coalesce(sum(rstat.open_total_elapsed_time), 0) + coalesce(sum(rstat.fetch_
total_elapsed_time), 0) as bigint) open_fetch_total_elapsed_time
from plg$prof_record_source_stats rstat
join plg$prof_cursors cur
   on cur.profile_id = rstat.profile_id and
      cur.statement_id = rstat.statement_id and
      cur.cursor_id = rstat.cursor_id
join plg$prof_record_sources recsrc
   on recsrc.profile_id = rstat.profile_id and
      recsrc.statement_id = rstat.statement_id and
      recsrc.cursor_id = rstat.cursor_id and
      recsrc.record_source_id = rstat.record_source_id
join plg$prof_statements sta
   on sta.profile_id = rstat.profile_id and
      sta.statement_id = rstat.statement_id
left join plg$prof_statements sta_parent
   on sta_parent.profile_id = sta.profile_id and
      sta_parent.statement_id = sta.parent_statement_id
group by rstat.profile_id,
         rstat.statement_id,
         sta.statement_type,
         sta.package_name,
         sta.routine_name,
         sta.parent_statement_id,
         sta_parent.statement_type,
         sta_parent.routine_name,
         rstat.cursor_id,
         cur.name,
         cur.line_num,
         cur.column_num,
         rstat.record_source_id,
         recsrc.parent_record_source_id,
         recsrc.level,
         recsrc.access_path
order by coalesce(sum(rstat.open_total_elapsed_time), 0) + coalesce(sum(rstat.fetch_
total_elapsed_time), 0) desc
```

2.24 Пакет RDB$BLOB_UTIL

Пакет RDB$BLOB_UTIL предназначен для обработки данных BLOB напрямую, чего не могут делать стандартные функции, такие как BLOB_APPEND и SUBSTRING.

2.24.1 Функция NEW_BLOB

Функция NEW_BLOB используется для создания нового BLOB. Она возвращает BLOB, предназначенный для добавления данных, как BLOB_APPEND. Преимущество перед BLOB_APPEND заключается в том, что можно задать опции SEGMENTED и TEMP_STORAGE. BLOB_APPEND всегда создает BLOB во временном хранилище, что может быть не лучшим подходом, если созданный BLOB будет храниться в постоянной таблице, поскольку потребуется копирование. BLOB, возвращаемый функцией NEW_BLOB, даже если

TEMP_STORAGE = FALSE, может использоваться с BLOB_APPEND для добавления данных.

Таблица 2.13 — Входные параметры функции NEW_BLOB

Параметр	Тип данных	Описание
SEGMENTED	BOOLEAN NOT NULL	Тип BLOB: true - сегментированный, false - потоковый
TEMP_STORAGE	BOOLEAN NOT NULL	Место хранения BLOB: true - во временном пространстве, false - в базе данных

Пример создания BLOB во временном пространстве и возвращение его в EXECUTE BLOCK:

```
execute block returns (b blob)
as
begin
  -- Создание дескриптора BLOB во временном пространстве.
  b = rdb$blob_util.new_blob(false, true);
  -- Добавление фрагментов данных.
  b = blob_append(b, '12345');
  b = blob_append(b, '67');
  suspend;
end
```

2.24.2 Функция OPEN_BLOB

Функция OPEN_BLOB используется для открытия существующего BLOB для чтения. Она возвращает дескриптор (целое число, связанное с транзакцией), подходящий для использования с другими функциями этого пакета, такими как SEEK, READ_DATA и CLOSE_HANDLE.

Таблица 2.14 — Входные параметры функции OPEN_BLOB

Параметр	Тип данных	Описание
BLOB	BLOB NOT NULL	Переменная типа BLOB

2.24.3 Функция IS_WRITABLE

Функция IS_WRITABLE возвращает true, если BLOB подходит для добавления данных без копирования с помощью BLOB_APPEND.

Таблица 2.15 — Входные параметры функции IS_WRITABLE

Параметр	Тип данных	Описание
BLOB	BLOB NOT NULL	Переменная типа BLOB

Пример проверки BLOB на доступность для записи:

```
create table t(b blob);

set term !;

execute block returns (bool boolean)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
as
    declare b blob;
begin
    b = blob_append(null, 'writable');
    bool = rdb$blob_util.is_writable(b);
    suspend;

    insert into t (b) values ('not writable') returning b into b;
    bool = rdb$blob_util.is_writable(b);
    suspend;
end!

set term ;!
```

2.24.4 Функция READ_DATA

Функция READ_DATA используется для чтения данных по дескриптору, открытому с помощью RDB$BLOB_UTIL.OPEN_BLOB. Возвращает NULL, когда BLOB полностью прочитан. Если LENGTH больше 0, то возвращает VARBINARY с его максимальной длиной. Если LENGTH равна NULL, то возвращает только сегмент BLOB с максимальной длиной 32765

Таблица 2.16 — Входные параметры функции READ_DATA

Параметр	Тип данных	Описание
HANDLE	INTEGER NOT NULL	Дескриптор
LENGTH	INTEGER	Длина

```
execute block returns (s varchar(10))
as
  declare b blob = '1234567';
  declare bhandle integer;
begin
  -- Открытие BLOB и получение дескриптора BLOB.
  bhandle = rdb$blob_util.open_blob(b);
  -- Получение и возвращение фрагментов данных в виде строки.
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  -- Здесь BLOB полностью прочитан, поэтому возвращается NULL.
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  -- Закрытие дескриптора BLOB.
  execute procedure rdb$blob_util.close_handle(bhandle);
end
```

2.24.5 Функция SEEK

Функция SEEK возвращает новую позицию для следующего READ_DATA. MODE может быть 0 (с начала), 1 (с текущей позиции) или 2 (с конца). Если MODE равен 2, OFFSET должен быть нулевым или отрицательным.

Таблица 2.17 — Входные параметры функции SEEK

Параметр	Тип данных	Описание
HANDLE	INTEGER NOT NULL	Дескриптор
MODE	INTEGER NOT NULL	Режим
OFFSET	INTEGER NOT NULL	Смещение

```
set term !;

execute block returns (s varchar(10))
as
  declare b blob;
  declare bhandle integer;
begin
  -- Создание дескриптора потокового BLOB
  b = rdb$blob_util.new_blob(false, true);
  -- Добавление данных.
  b = blob_append(b, '0123456789');
  -- Открытие BLOB.
  bhandle = rdb$blob_util.open_blob(b);
  -- Поиск с начала до 5.
  rdb$blob_util.seek(bhandle, 0, 5);
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  -- Поиск с начала до 2.
  rdb$blob_util.seek(bhandle, 0, 2);
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  -- Ещё на 2 вперед.
  rdb$blob_util.seek(bhandle, 1, 2);
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
  -- Поиск с конца на один элемент назад.
  rdb$blob_util.seek(bhandle, 2, -1);
  s = rdb$blob_util.read_data(bhandle, 3);
  suspend;
end!

set term ;!
```

2.24.6 Процедура CANCEL_BLOB

Процедура CANCEL_BLOB используется для немедленного освобождения временного BLOB, например, созданного с помощью BLOB_APPEND.

Обратите внимание, что если тот же BLOB будет использован после отмены, используя ту же переменную или другую переменную с той же ссылкой на BLOB id, будет выдана ошибка invalid blob id.

2.24.7 Процедура CLOSE_HANDLE

Процедура CLOSE_HANDLE используется для закрытия дескриптора BLOB, открытого с помощью OPEN_BLOB. Незакрытые дескрипторы закрываются автоматически только в конце транзакции.

Таблица 2.18 — Входные параметры функции CLOSE_HANDLE

Параметр	Тип данных	Описание
HANDLE	INTEGER NOT NULL	Дескриптор

2.25 Функция RDB$TRACE_MSG

Функция RDB$TRACE_MSG записывает указанное сообщение в лог-файл.

Для работы функции необходимо включить параметр log_message в fbtrace.conf.

```
RDB$TRACE_MSG('<сообщение>' [, <eol>])

<eol> ::= TRUE | FALSE
```

Сообщение представляет собой строковое значение, которое необходимо записать в лог-файл.

Параметр eol означает сброс буфера сообщений в лог. По умолчанию он включен (значение true), то есть каждое сообщение из буфера будет записано отдельным событием (MESSAGE) в лог-файл. Если eol выключен, то сообщение будет добавлено в буфер подключения. Сообщения из буфера будут записаны в лог, когда функция RDB$TRACE_MSG будет вызвана с включенным eol или, когда завершится текущий блок PSQL (execute block, функция, процедура).

Максимальный размер буфера составляет 1Мб. Если при вызове RDB$TRACE_MSG содержимое буфера превышает 1Мб, то будет получено сообщение об ошибке, а в лог будут записаны сообщения, добавленные в буфер до превышения лимита.

Глава 3

Изменения API и ODS

3.1 Изменения ODS

В РЕД Базе Данных 5 используется ODS версии 13.1. Определить версию ODS можно с помощью утилиты gstat, указав ключ -h: gstat -h. Обновить минорную версию ODS теперь можно с помощью утилиты gfix и ключа -upgrade: gfix -upgrade <база данных>.

Таблица 3.1 — Соответствие версий ODS версиям РЕД Базы Данных

Версия СУБД	Версия ODS
РЕД База Данных 5.0.0	13.1
РЕД База Данных 3.0.8 - РЕД База Данных 3.0.10	12.3
РЕД База Данных 3.0.0 - РЕД База Данных 3.0.7	12.0
РЕД База Данных 2.6	2.4
РЕД База Данных 2.5	24578.3
РЕД База Данных 2.1	24578.2
РЕД База Данных 2.0	11.0

3.1.1 Новые системные таблицы

RDB$KEYWORDS - Содержит информацию о ключевых словах сервера.

MON$COMPILED_STATEMENTS - Содержит информацию о скомпилированных запросах.

RDB$TIME_ZONES - Виртуальная таблица со списком часовых поясов поддерживаемых сервером.

RDB$PUBLICATIONS - Хранит публикации репликации, определенной в базе данных.

RDB$PUBLICATION_TABLES - Хранит имена таблиц, которые реплицируются в рамках публикации.

RDB$CONFIG - Содержит сведения о параметрах конфигурации текущей базы данных для текущего подключения. Доступна только администраторам, другие пользователи не видят строк этой таблицы.

3.1.2 Новые столбцы в системных таблицах

Столбцы RDB$CONDITION_SOURCE и RDB$CONDITION_BLR были добавлены в таблицу RDB$INDICES, они относятся к частичным индексам.

Столбец MON$SESSION_TIMEZONE добавлен в таблицу MON$ATTACHMENTS.

Столбец MON$COMPILED_STATEMENT_ID добавлен в таблицы MON$STATEMENTS и MON$CALL_STACK.

Столбец SEC$DESCRIPTION добавлен в таблицу SEC$GLOBAL_AUTH_MAPPING.

Столбец RDB$SYSTEM_PRIVILEGES добавлен в таблицу RDB$ROLES.

Столбец RDB$SQL_SECURITY добавлен в таблицы RDB$DATABASE, RDB$FUNCTIONS, RDB$PACKAGES, RDB$PROCEDURES, RDB$RELATIONS, RDB$TRIGGERS.

3.2 Программные интерфейсы

3.2.1 Изменения основного API

Ряд новых методов был добавлен в следующие интерфейсы.

ResultSet

```
void getInfo(Status status,
           uint itemsLength, const uchar* items,
           uint bufferLength, uchar* buffer);
```

Используется для получения информации о курсоре. В настоящее время поддерживается только `INF_RECORD_COUNT`. `INF_RECORD_COUNT` возвращает количество записей, кэшированных во время работы курсором, или -1 для однонаправленного (только вперед) курсора.

3.2.2 Расширение метода `getInfo()`

`Attachment::getInfo()`

Были добавлены следующие информационные теги:

- `fb_info_protocol_version` - Версия удаленного протокола, используемого текущим соединением;
- `fb_info_crypt_plugin` - Имя плагина шифрования базы данных;
- `fb_info_wire_crypt` - Имя плагина для шифрования соединения;
- `fb_info_statement_timeout_db` - Таймаут выполнения оператора, установленный в файле конфигурации;
- `fb_info_statement_timeout_att` - Таймаут выполнения оператора, установленный на уровне соединения;
- `fb_info_ses_idle_timeout_db` - Таймаут простоя соединения, установленный в конфигурационном файле;
- `fb_info_ses_idle_timeout_att` - Таймаут простоя соединения, установленный на уровне соединения;
- `fb_info_ses_idle_timeout_run` - Фактическое значение таймаута для текущего соединения;
- `fb_info_creation_timestamp_tz` - Время создания базы данных (с указанием часового пояса);
- `fb_info_features` - Список функций, поддерживаемых в текущем соединении;
- `fb_info_next_attachment` - Текущее значение параметра `Next attachment ID`;
- `fb_info_next_statement` - Текущее значение параметра `Next statement ID`;
- `fb_info_db_guid` - `GUID` базы данных;
- `fb_info_db_file_id` - Идентификатор файла базы данных на уровне файловой системы;
- `fb_info_replica_mode` - Режим реплики.

Возможные характеристики, возвращаемые `fb_info_features`:

- `fb_feature_multi_statements` - Несколько препарированных запросов в одном подключении;
- `fb_feature_multi_transactions` - Несколько одновременных транзакций в одном подключении;
- `fb_feature_named_parameters` - Параметры запроса могут быть именованными;

- `fb_feature_session_reset` - Поддерживается оператор `ALTER SESSION RESET`;
- `fb_feature_read_consistency` - Поддерживается уровень изоляции транзакций `Read Consistency`;
- `fb_feature_statement_timeout` - Поддерживаются таймауты выполнения операторов;
- `fb_feature_statement_long_life` - Подготовленные запросы не удаляются при завершении транзакции.

Возможные режимы реплики, возвращаемые `fb_info_replica_mode`:

- `fb_info_replica_none` - База данных не находится в состоянии реплики;
- `fb_info_replica_read_only` - База данных является репликой, доступной только для чтения;
- `fb_info_replica_read_write` - База данных является репликой, доступной для чтения и записи.

`Statement::getInfo()`

Были добавлены следующие информационные теги:

- `isc_info_sql_exec_path_blr_bytes` - Путь выполнения в виде `BLR` (бинарный формат);
- `isc_info_sql_stmt_timeout_user` - Значение таймаута для текущего оператора;
- `isc_info_sql_stmt_timeout_run` - Фактическое значение таймаута для текущего оператора;
- `isc_info_sql_stmt_blob_align` - Выравнивание потока блобов в `Batch API`;
- `isc_info_sql_exec_path_blr_text` - Путь выполнения в виде `BLR` (текстовый формат).

`Transaction::getInfo()`

Были добавлены следующие информационные теги:

- `fb_info_tra_snapshot_number` - Номер снапшота текущей транзакции.

3.2.3 Расширение Legacy (ISC) API

В ISC API было добавлено несколько функций.

```
ISC_STATUS fb_get_transaction_interface(ISC_STATUS*, void*, isc_tr_handle*);
ISC_STATUS fb_get_statement_interface(ISC_STATUS*, void*, isc_stmt_handle*);
```

Они могут быть использованы для получения объекта `OO API` из соответствующего обработчика `ISC API`.

3.2.4 Изменения в Services API

Поддержка `gstat -parallel`

Был добавлен параметр для указания количества параллельных рабочих потоков:

- `sts_par_workers <n>` - Количество параллельных рабочих потоков для сбора статистики. Включает многопоточный сбор статистики.

Пример сбора статистики в 4 потока утилитой `rdbsvcmgr` (логин и пароль были опущены для краткости):

```
rdbsvcmgr -action_db_stats <база данных> -sts_par_workers 4
```

Поддержка gfix -upgrade

Добавлен параметр для обновления минорной версии ODS:

- isc_spb_rpr_upgrade_db - обновления минорной версии ODS.

Пример обновления минорной версии ODS утилитой rdbsvcmgr (логин и пароль были опущены для краткости):

```
rdbsvcmgr -action_repair -rpr_upgrade_db <база данных>
```

Поддержка nbackup -fixup

Добавлено действие для разблокировки базы данных после самостоятельного восстановления из заблокированной резервной копии:

- isc_action_svc_nfix - разблокировка базы данных после самостоятельного восстановления из блокированной резервной копии.

Пример разблокировки базы утилитой rdbsvcmgr (логин и пароль были опущены для краткости):

```
rdbsvcmgr -action_nfix dbname /tmp/ecopy.fdb
```

3.2.5 Поддержка таймаута простоя соединения в API

Получение/установка таймаута простоя соединения в секундах:

```
interface Attachment
      uint getIdleTimeout(Status status);
      void setIdleTimeout(Status status, uint timeOut);
```

Значения таймаута простоя соединения на уровне конфигурации и соединения, а также текущий фактический таймаут можно узнать с помощью с помощью новых информационных тегов:

- fb_info_ses_idle_timeout_db - Значение, установленное на уровне конфигурации;
- fb_info_ses_idle_timeout_att - Значение, установленное на уровне данного соединения;
- fb_info_ses_idle_timeout_run - Фактическое значение таймаута для данного соединения.

3.2.6 Поддержка таймаута выполнения операторов в API

Таймаут выполнения оператора на уровне соединения в миллисекундах:

```
interface Attachment
  uint getStatementTimeout(Status status);
  void setStatementTimeout(Status status, uint timeOut);
```

Получение/установка таймаута выполнения оператора в миллисекундах:

```
interface Statement
  uint getTimeout(Status status);
  void setTimeout(Status status, uint timeOut);
```

Установка таймаута выполнения оператора с помощью ISC API:

```
ISC_STATUS ISC_EXPORT fb_dsql_set_timeout(ISC_STATUS*, isc_stmt_handle*, ISC_ULONG);
```

Узнать таймаут выполнения оператора на уровне конфигурации и/или соединения можно с помощью `isc_database_info()`, используя новые информационные теги:

- `fb_info_statement_timeout_db` - Таймаут выполнения оператора, установленный в файле конфигурации;
- `fb_info_statement_timeout_att` - Таймаут выполнения оператора, установленный на уровне соединения.

Узнать таймаут выполнения на уровне оператора можно с помощью `isc_dsql_info()`, добавив несколько новых информационных тегов:

- `isc_info_sql_stmt_timeout_user` - Значение таймаута для данного оператора;
- `isc_info_sql_stmt_timeout_run` - Фактическое значение таймаута для данного оператора. Действительно только для операторов, выполняющихся в данный момент, т. е. когда таймер действительно запущен.

3.2.7 Поддержка READ COMMITTED READ CONSISTENCY в API

Для запуска транзакции `READ COMMITTED READ CONSISTENCY` через `ISC API`, используйте параметр `isc_tpb_read_consistency` в буфере параметров транзакции.

3.2.8 Поддержка пакетных операций в API

В РЕД Базе Данных 5 появилась возможность пакетного выполнения подготовленных операторов с более чем одним набором параметров. Новый пакетный интерфейс СУБД позволяет удовлетворить требования `JDBC` по пакетной обработке подготовленных операторов, но у него есть несколько принципиальных отличий:

- Интерфейс пакетной обработки ориентирован на работу с сообщениями, а не с отдельными полями;
- Пакетный интерфейс поддерживает использование встроенных `BLOB-ов`, что особенно эффективно при работе с небольшими `BLOB-ами`.
- В результате выполнения пакетной операции возвращается не простой массив целых чисел, а специальный интерфейс, который, в зависимости от параметров создания пакета, может содержать как информацию об обновленных записях, так и флаг ошибки, дополненный подробными векторами состояния для сообщений, вызвавших ошибки выполнения.

Создание пакета

Пакет может быть создан двумя способами - с помощью интерфейса `IStatement` или `IAttachment`. В обоих случаях вызывается метод `createBatch()` соответствующего интерфейса.

В случае с `IAttachment` текст SQL-оператора, который будет выполняться в пакете, передается напрямую в метод `createBatch()`.

Настройка пакетной операции выполняется с помощью блока параметров пакета - `Batch Parameters Block (BPB)`, формат которого аналогичен `DPB v.2`: начинается с тега (`IBatch::CURRENT_VERSION`) и далее следует набор широких блоков: 1-байтовый тег, 4-байтовая длина, значение длины байт. Возможные теги описаны в интерфейсе пакета `IBatch`.

Рекомендуемый (и самый простой) способ создания BPB для создания пакетов - использовать соответствующий интерфейс `IXpbBuilder`:

```
IXpbBuilder* pb = utl->getXpbBuilder(&status, IXpbBuilder::BATCH, NULL, 0);
pb->insertInt(&status, IBatch::RECORD_COUNTS, 1);
```

Такое использование BPB позволяет пакету учитывать количество обновленных записей на основе каждого сообщения.

Создание пакетного интерфейса

Чтобы создать пакетный интерфейс с нужными параметрами, необходимо передать BPB в метод createBatch():

```
IBatch* batch = att->createBatch(&status, tra, 0, sqlStmtText, SQL_DIALECT_V6, NULL,
pb->getBufferLength(&status), pb->getBuffer(&status))
```

В этом примере пакетный интерфейс создаётся с форматом сообщения по умолчанию, поскольку вместо формата входных метаданных передается NULL.

Получение формата сообщения

Чтобы продолжить работу с созданным пакетным интерфейсом, нужно получить формат содержащихся в нем сообщений с помощью метода getMetadata():

```
IMessageMetadata* meta = batch->getMetadata(&status);
```

В интерфейс пакета может быть передан свой собственный формат сообщений.

В данном примере предполагается, что существует некая функция, которая может заполнить буфер "data" в соответствии с переданным форматом "metadata", например:

```
fillNextMessage(unsigned char* data, IMessageMetadata* metadata)
```

Буфер сообщений

Для работы с сообщениями нужен буфер для "data":

```
unsigned char* data = new unsigned char[meta->getMessageLength(&status)];
```

После создания буфера сообщений и его заполнения, в пакет можно добавлять сообщения, например:

```
fillNextMessage(data, meta);
batch->add(&status, 1, data);
fillNextMessage(data, meta);
batch->add(&status, 1, data);
```

Альтернативным способом работы с сообщениями является использование макроса FB_MESSAGE.

Выполнение пакетной операции

После подготовки пакетная операция готова к выполнению:

```
IBatchCompletionState* cs = batch->execute(&status, tra);
```

При формировании блока параметров пакета был добавлен флаг для определения количества измененных записей (вставленных, обновленных или удаленных) в каждом сообщении. Для получения статистики используется интерфейс BatchCompletionState. Общее количество сообщений, кото-

рые были обработаны в пакете, может быть меньше количества сообщений, переданных в пакет, если произошла ошибка и не была включена опция, разрешающая множественные ошибки при пакетной обработке. Получить количество обработанных сообщений можно с помощью метода `getSize()`:

```
unsigned total = cs->getSize(&status);
```

Получение статуса выполнения отдельной операции:

```
for (unsigned p = 0; p < total; ++p)
  printf("Msg %u state %d\n", p, cs->getState(&status, p));
```

Очистка

После обработки статусов пакетных операций, необходимо очистить буфер статистики:

```
cs->dispose();
```

Если по какой-то причине требуется очистить буфер пакетной операции, не выполняя его, например, готовясь к обработке новой партии сообщений, необходимо использовать `cancel()`:

```
batch->cancel(&status);
```

Для освобождения буфера пакетной операции необходимо использовать `release()`:

```
batch->release();
```

Добавление множества сообщений за один вызов пакетной операции

За одно обращение к пакету можно добавить более одного сообщения. Важно помнить, что для корректной работы этой функции сообщения должны быть соответствующим образом выровнены. Требуемое выравнивание и выровненный размер сообщения можно получить из интерфейса `MessageMetadata`, вызвав метод `getAlignedLength()`. Например:

```
unsigned aligned = meta->getAlignedLength(&status);
```

Этот размер необходим при создании массива сообщений и работе с ним:

```
unsigned char* data = new unsigned char[aligned * N];
for (int n = 0; n < N; ++n) fillNextMessage(&data[aligned * n], meta);
batch->add(&status, N, data);
```

После этого пакет может быть выполнен или к нему может быть добавлена следующая партия сообщений.

Передача встроенных BLOB в пакетных операциях

Как правило, `BLOB` несовместимы с пакетными операциями. Пакетные операции эффективны, когда необходимо передать серверу много небольших данных за одну операцию. Блобы рассматриваются как большие объекты, поэтому, как правило, нет смысла использовать их в пакетных операциях.

Тем не менее на практике встречаются `BLOB` небольшого размера. В этом случае использование традиционного `BLOB API` (создание `BLOB`, передача сегментов на сервер, закрытие `BLOB`, передача идентификатора `BLOB` в сообщении) снижает производительность, особенно при передаче по сети. Поэтому пакетная обработка в СУБД поддерживает передачу встроенного `BLOB`, вместе с другими сообщениями.

Политика использования BLOB

Чтобы использовать функцию встроенного BLOB, сначала необходимо установить политику использования BLOB в качестве опции в BPB для создаваемой пакетной операции:

```
pb->insertInt(&status, IBatch::BLOB_IDS, IBatch::BLOB_IDS_ENGINE);
```

Этот пример показывает, что для самых простых и довольно распространенных сценариев использования сервер создаёт временные идентификаторы BLOB, необходимые для поддержания связи между BLOB и сообщением, в котором он передается. Например, сообщение описано следующим образом:

```
FB_MESSAGE(Msg, ThrowStatusWrapper,
(FB_VARCHAR(5), id)
(FB_VARCHAR(10), name)
(FB_BLOB, desc)
) project(&status, master);
```

В результате на сервер будет отправлено сообщение, содержащее BLOB:

```
project->id = ++idCounter;
project->name.set(currentName);
batch->addBlob(&status, descriptionSize, descriptionText, &project->desc);
batch->add(&status, 1, project.getData());
```

BLOB больших размеров

Если какой-то BLOB оказался слишком большим, чтобы поместиться в существующий буфер пакета, то вместо пересоздания буфера можно использовать метод appendBlobData(), чтобы добавить больше данных к последнему добавленному BLOB:

```
batch->addBlob(&status, descriptionSize, descriptionText, &project->desc, bpbLength,
bpb);
```

После добавления первой части BLOB, формируется следующая часть данных в descriptionText, обновляется descriptionSize и затем выполняется метод appendBlobData():

```
batch->appendBlobData(&status, descriptionSize, descriptionText);
```

Метод может вызываться в цикле, но необходимо следить за тем, чтобы не переполнить внутренний буфер пакета. Его размер контролируется параметром BUFFER_BYTES_SIZE при создании объекта пакетного интерфейса. По умолчанию размер составляет 10 МБ, но он не может превышать 40 МБ. Если необходимо обработать слишком большой BLOB, при использовании пакетной обработки на основе данных с большим количеством маленьких BLOB, лучше использовать стандартное BLOB API и метод registerBlob() интерфейса IBatch.

Пользовательские идентификаторы BLOB

Еще один возможный параметр в политике BLOB - BLOB_IDS_USER, который позволяет предоставить временный BLOB_ID вместо того, чтобы он был сгенерирован сервером.

Использование не имеет существенных отличий. Перед вызовом addBlob() необходимо указать уникальный для каждой операции идентификатор, в памяти, на которую ссылается последний параметр. Точно такой же идентификатор должен быть передан в сообщении данных для BLOB.

Учитывая, что генерация BLOB-идентификаторов сервером происходит очень быстро, такая политика может показаться бесполезной. Но в случае, когда происходит обращение к BLOB и другим данным в относительно независимых потоках (например, блоки в файле), и в них уже присутствуют некоторые подходящие идентификаторы, использование идентификаторов BLOB может значительно

упростить код.

Потоки и сегменты

Блобы, созданные с помощью интерфейса IBatch, по умолчанию являются потоковыми, а не сегментными, как блобы, созданные с помощью createBlob(). Формат сегментированных блобов поддерживается только для обратной совместимости.

Переопределение сегментированных BLOB

Если необходимо использовать сегментированные BLOB, можно переопределить значение по умолчанию, вызвав метод setDefaultBpb() интерфейса пакета:

```
batch->setDefaultBpb(&status, bpbLength, bpb);
```

Конечно, передаваемый BPB может содержать и другие параметры создания BLOB. Также можно передать BPB непосредственно в метод addBlob(), но если большинство BLOB, которые требуется добавить, имеют одинаковый формат, отличный от формата по умолчанию, эффективнее будет использовать метод setDefaultBpb().

Вызов addBlob() добавит первый сегмент в BLOB. Последующие вызовы appendBlobData() будут добавлять новые сегменты.

Размер сегмента ограничен 64 КБ -1. Попытка передать больше данных за один вызов приведет к ошибке.

Добавление нескольких BLOB с помощью потоков

Используя метод addBlobStream(), можно за один вызов добавить в пакет более одного BLOB.

Поток блобов - это последовательность BLOB. Каждый поток начинается с BLOB-заголовка, который должен быть соответствующим образом выровнен.

Интерфейс IBatch предоставляет специальный метод getBlobAlignment() для этой цели:

```
unsigned alignment = batch->getBlobAlignment(&status);
```

Предполагается, что все компоненты BLOB-потока в пакете будут соответствующим образом сбалансированы по размеру сегментов потока, передаваемых в addBlobStream(), который должен быть кратным этому значению выравнивания.

Заголовок содержит три поля: 8-байтовый идентификатор BLOB (должен быть ненулевым), 4-байтовый общий размер BLOB и 4-байтовый размер BPB. Общий размер BLOB включает в себя вложенный BPB, т.е. следующий BLOB в потоке всегда будет найден в байтах размера BLOB после заголовка, с учетом выравнивания.

BPB присутствует, если размер BPB не равен нулю, и располагается сразу после заголовка. Далее идут данные BLOB, формат которых зависит от того, является ли BLOB потоковым или сегментированным:

- Для потокового BLOB это обычная последовательность байтов, размер которой равен (BLOB-size - BPB-size).
- Для сегментированного BLOB все немного сложнее: данные BLOB представляют собой набор сегментов, где каждый сегмент имеет формат: 2-байта для размера сегмента, выровненные по границе IBatch::BLOB_SEGHDR_ALIGN, за которыми следует столько байт, сколько приходится на этот 2-байтовый размер сегмента.

Большие BLOB в потоке

Когда в поток добавляется большой BLOB, его размер не всегда известен заранее. Чтобы избежать слишком большого буфера для этого BLOB (размер должен быть указан в заголовке BLOB, перед данными BLOB), можно использовать запись продолжения BLOB. В заголовке BLOB указывается размер BLOB в значении, известном при создании заголовка, и добавляется запись для продолжения. Формат

записи продолжения идентичен формату заголовка BLOB, за исключением того, что идентификатор BLOB и размер BPB всегда должны быть равны нулю.

Как правило, на каждый вызов addBlobStream() требуется одна запись продолжения.

Регистрация стандартного BLOB

Последний метод, используемый для работы с BLOB, отличается от первых трех, которые передают BLOB-данные вместе с остальными данными пакета. Он необходим для регистрации в пакете идентификатора BLOB, созданного с помощью стандартного BLOB API. Это может быть полезно, если в пакет необходимо передать действительно большой BLOB.

Идентификатор такого BLOB нельзя использовать в пакете напрямую, чтобы не вызвать ошибку "invalid BLOB ID" во время выполнения пакета. Вместо этого необходимо вызывать метод registerBlob():

```
batch->registerBlob(&status, &realId, &msg->desc);
```

Если политика BLOB разрешает генерировать BLOB ID, то этого кода достаточно, чтобы правильно зарегистрировать существующий BLOB в пакете. В других случаях необходимо присвоить msg->desc тот идентификатор, который будет правильным с точки зрения пакета.

Пакетные операции в Legacy (ISC) API

Для выполнения подготовленного ISC-оператора в пакетном режиме используются следующие методы ISC API: fb_get_transaction_interface и fb_get_statement_interface. Эти методы позволяют получить доступ к соответствующим интерфейсам так же, как и к существующим дескрипторам ISC.

3.2.9 Поддержка временных зон в API

Структура

```
  struct ISC_TIME_TZ
{
    ISC_TIME utc_time;
    ISC_USHORT time_zone;
};
```

```
struct ISC_TIMESTAMP_TZ
{
    ISC_TIMESTAMP utc_timestamp;
    ISC_USHORT time_zone;
};
```

```
struct ISC_TIME_TZ_EX
{
    ISC_TIME utc_time;
    ISC_USHORT time_zone;
    ISC_SHORT ext_offset;
};
```

```
struct ISC_TIMESTAMP_TZ_EX
{
    ISC_TIMESTAMP utc_timestamp;
    ISC_USHORT time_zone;
    ISC_SHORT ext_offset;
};
```

API функции

```
void decodeTimeTz(
    Status status,
    const ISC_TIME_TZ* timeTz,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```
void decodeTimeStampTz(
    Status status,
    const ISC_TIMESTAMP_TZ* timeStampTz,
    uint* year,
    uint* month,
    uint* day,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```
void encodeTimeTz(
    Status status,
    ISC_TIME_TZ* timeTz,
    uint hours,
    uint minutes,
    uint seconds,
    uint fractions,
    const string timeZone
);
```

```
void encodeTimeStampTz(
    Status status,
    ISC_TIMESTAMP_TZ* timeStampTz,
    uint year,
    uint month,
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    uint day,
    uint hours,
    uint minutes,
    uint seconds,
    uint fractions,
    const string timeZone
);
```

```
void decodeTimeTzEx(
    Status status,
    const ISC_TIME_TZ_EX* timeTzEx,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```
void decodeTimeStampTzEx(
    Status status,
    const ISC_TIMESTAMP_TZ_EX* timeStampTzEx,
    uint* year,
    uint* month,
    uint* day,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

3.2.10 Поддержка NUMERIC и DECIMAL в API

`DecFloat16` и `DecFloat34` - это вспомогательные интерфейсы, которые упрощают работу с типами данных `DECFLOAT`. В интерфейсе `DecFloat16` доступны следующие методы:

```
void toBcd(
    const FB_DEC16* from,
    int* sign,
    uchar* bcd,
    int* exp
);
```

```
void toString(
    Status status,
    const FB_DEC16* from,
    uint bufferLength,
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    string buffer
);
```

```
void fromBcd(
    int sign,
    const uchar* bcd,
    int exp,
    FB_DEC16* to
);
```

```
void fromString(
    Status status,
    const string from,
    FB_DEC16* to
);
```

Интерфейс `DecFloat34` содержит те же методы, только использует структуру `FB_DEC34`.

`Int128` - вспомогательный интерфейс для 128-битных целых чисел (используется как базовый тип для `INT128`, а также для типов данных `NUMERIC` и `DECIMAL` с точностью больше 18), содержит следующие методы:

```
void toString(
    Status status,
    const FB_I128* from,
    int scale,
    uint bufferLength,
    string buffer
);
```

```
void fromString(
    Status status,
    int scale,
    const string from,
    FB_I128* to
);
```

Структуры, используемые указанными интерфейсами:

```
struct FB_DEC16
{
  ISC_UINT64 fb_data[1];
};
```

```
struct FB_DEC34
{
  ISC_UINT64 fb_data[2];
};
```

```
struct FB_I128
{
  ISC_UINT64 fb_data[2];
};
```

Интерфейс `Util` был расширен следующими методами:

```
DecFloat16 getDecFloat16(Status status);
DecFloat34 getDecFloat34(Status status);
Int128 getInt128(Status status);
```

3.2.11 Изменения в других интерфейсах

Новые методы были добавлены в следующие интерфейсы:

Attachment:

```
uint getIdleTimeout(Status status);
void setIdleTimeout(Status status, uint timeOut);

uint getStatementTimeout(Status status);
void setStatementTimeout(Status status, uint timeOut);
Batch createBatch(
    Status status,
    Transaction transaction,
    uint stmtLength,
    const string sqlStmt,
    uint dialect,
    MessageMetadata inMetadata,
    uint parLength,
    const uchar* par
);
```

Statement:

```
uint getTimeout(Status status);
void setTimeout(Status status, uint timeout);
Batch createBatch(
    Status status,
    MessageMetadata inMetadata,
    uint parLength,
    const uchar* par
);
```

ClientBlock:

```
AuthBlock getAuthBlock(Status status);
```

Server:

```
void setDbCryptCallback(Status status, CryptKeyCallback cryptCallback);
```

MessageMetadata:

```
uint getAlignment(Status status);
uint getAlignedLength(Status status);
```

MetadataBuilder:

```
void setField(Status status, uint index, const string field);
void setRelation(Status status, uint index, const string relation);
void setOwner(Status status, uint index, const string owner);
void setAlias(Status status, uint index, const string alias);
```

FirebirdConf:

```
uint getVersion(Status status);
```

ConfigManager:

```
const string getDefaultSecurityDb();
```

Глава 4

Изменения списка зарезервированных слов

4.1 Новые ключевые слова

Таблица 4.1 — Новые ключевые слова

BASE64_DECODE	BASE64_ENCODE	BIND
CLEAR	COMPARE_DECFLOAT	CONNECTIONS
CONSISTENCY	COUNTER	CRYPT_HASH
CTR_BIG_ENDIAN	CTR_LENGTH	CTR_LITTLE_ENDIAN
CUME_DIST	DEFINER	DISABLE
ENABLE	EXCESS	EXCLUDE
EXTENDED	FIRST_DAY	FOLLOWING
HEX_DECODE	HEX_ENCODE	IDLE
INCLUDE	INVOKER	IV
LAST_DAY	LEGACY	LIFETIME
LPARAM	MAKE_DBKEY	MESSAGE
MODE	NATIVE	NORMALIZE_DECFLOAT
NTILE	NUMBER	OLDEST
OTHERS	OVERRIDING	PERCENT_RANK
POOL	PRECEDING	PRIVILEGE
QUANTIZE	RANGE	RESET
RSA_DECRYPT	RSA_ENCRYPT	RSA_PRIVATE
RSA_PUBLIC	RSA_SIGN_HASH	RSA_VERIFY_HASH
SALT_LENGTH	SECURITY	SESSION
SIGNATURE	SQL	SYSTEM
TIES	TOTALORDER	TRAPS
LOCKED	OPTIMIZE	QUARTER
TARGET	TIMEZONE_NAME	UNICODE_CHAR
UNICODE_VAL		

4.2 Новые зарезервированные слова

Таблица 4.2 — Новые зарезервированные слова

BINARY	DECFLOAT	INT128
LATERAL	LOCAL	LOCALTIME

(разрыв таблицы)

(разрыв таблицы)

LOCALTIMESTAMP	PUBLICATION	RDB$GET_TRANSACTION_CN
RDB$ERROR	RDB$ROLE_IN_USE	RDB$SYSTEM_PRIVILEGE
RESETTING	TIMEZONE_HOUR	TIMEZONE_MINUTE
UNBOUNDED	VARBINARY	WINDOW
WITHOUT		

Глава 5

Изменения параметров конфигурации

5.1 Новые параметры конфигурации

MaxStatementCacheSize

Определяет максимальный объем памяти, используемый для кэширования неиспользуемых скомпилированных операторов DSQL. Значение ноль означает, что кэширование не используется. Значение по умолчанию - 2 мегабайта.

OnDisconnectTriggerTimeout

Параметр `OnDisconnectTriggerTimeout` устанавливает число секунд, по истечении которых выполнение триггера на событие `DISCONNECT` будет прекращено. Параметр имеет целочисленный тип. Значение по умолчанию равно 180. Значение 0 означает, что время выполнения триггера на событие `DISCONNECT` не ограничено.

```
OnDisconnectTriggerTimeout = 180
```

DefaultProfilerPlugin

Определяет плагин профайлера по умолчанию, используемый для профилирования с помощью пакета `RDB$PROFILER`.

OptimizeForFirstRows

Параметр определяет, что нужно оптимизировать: получение первых строк или всех. Значение `false` соответствует стратегии `ALL ROWS`, а `true` соответствует стратегии `FIRST ROWS`.

По умолчанию оптимизатор направлен на получение всех строк.

```
OptimizeForFirstRows = false
```

OuterJoinConversion

Определяет, может ли оптимизатор преобразовывать `OUTER-соединения` в `INNER-соединения` при условии, что такое преобразование возможно с точки зрения результатов запроса.

По умолчанию включено. Может быть отключено для упрощения процесса миграции, если `OUTER-соединения` намеренно используются в `SQL-запросах` (например, в качестве подсказок оптимизатора), даже если известно, что они семантически эквивалентны `INNER-соединениям`.

StatementTimeout

Это количество секунд, по истечении которых выполнение оператора будет автоматически прекращено сервером. Ноль означает, что время ожидания не установлено. Значение по умолчанию равно 0.

```
StatementTimeout = 0
```

ConnectionIdleTimeout

Это количество минут, по истечении которых бездействующее соединение будет отключено сервером. Ноль означает, что время ожидания не установлено. Значение по умолчанию равно 0.

```
ConnectionIdleTimeout = 0
```

ExtConnPoolSize

Устанавливает максимальное количество бездействующих соединений в пуле внешних соединений. Допустимые значения от 0 до 1000. Нулевое значение означает что пул выключен.

```
ExtConnPoolSize = 0
```

ExtConnPoolLifeTime

Устанавливает время жизни бездействующих соединений в пуле внешних соединений. Допустимые значения от 1 секунды до 24 часов (86400 секунд).

```
ExtConnPoolLifeTime = 7200
```

MaxIdentifierByteLength

Максимально допустимая длина имени идентификатора в байтах. Установка этого значения для всех баз данных (включая базу данных безопасности) может вызвать проблемы.

```
MaxIdentifierByteLength = 252
```

MaxIdentifierCharLength

Максимально допустимая длина имени идентификатора в символах. Установка этого значения для всех баз данных (включая базу данных безопасности) может вызвать проблемы.

```
MaxIdentifierCharLength = 63
```

ReadConsistency

Параметр обеспечивает согласованность чтения данных для запросов в режиме `READ COMMITTED`. При запуске такого запроса для него делается снимок версий, который будет использоваться запросом для чтения данных. В этом режиме флаги транзакций `rec_version / no_rec_version` не действуют: любой `READ COMMITTED` транзакции по умолчанию назначаются режимы `read consistency record version`; значение `no_rec_version` игнорируется.

```
ReadConsistency = 1
```

TipCacheBlockSize

Количество выделяемой памяти для кэша каждого блока `TIP`. Понизьте это значения, если у вас небольшой `TIP` и вы хотите сберечь память. Увеличьте это значение, если вам нужен очень большой кэш и ограничения доступа к объектам ядра, выделенным для каждого блока (файлы, мьютексы и т. д.). Каждая кэшированная транзакция использует 8 байтов памяти.

```
TipCacheBlockSize = 4M
```

SnapshotsMemSize

Кол-во памяти для хранения снапшотов. Оно будет расти автоматически, если вы не используете экзотическую платформу, которая не является `Windows` и не поддерживает системный вызов `mmap`. Каждый активный `shapshot` использует 16 байтов памяти.

```
SnapshotsMemSize = 64K
```

ClientBatchBuffer

Определяет размер буфера, используемого клиентским подключением для пакетной передачи на сервер (при использовании пакетного API).

```
ClientBatchBuffer = 131072
```

DataTypeCompatibility

Задает уровень совместимости, определяющий, какие типы данных доступны клиентскому API. В настоящее время доступны два варианта: 3.0 и 2.5. Режим эмуляции 3.0 скрывает типы данных, появившиеся после версии 3.0, а именно `DECIMAL` и `NUMERIC` с точностью 19 и выше, `DECFLOAT, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE`. Соответствующие значения возвращаются через типы данных, поддерживаемые версией 3.0. Режим эмуляции 2.5 преобразует ещё и тип данных `BOOLEAN`. Этот параметр позволяет устаревшим клиентским приложениям работать с версией 5 без перекомпиляции для обработки новых типов данных.

DefaultTimeZone

Часовой пояс сеанса или клиента. Если параметр не установлен, то часовой пояс сеанса будет тем же, что используется операционной системой. При установке значения на сервере он определяет часовой пояс сеанса по умолчанию для подключений. Когда он установлен на клиенте, он определяет часовой пояс по умолчанию, используемый функциями API на стороне клиента.

OutputRedirectionFile

Позволяет перенаправлять потоки сервера `stdout/stderr` в пользовательский файл. По умолчанию эти потоки открываются сервером, но выходные данные отбрасываются.

```
OutputRedirectionFile =
```

Srp256 стал методом аутентификации по умолчанию

Плагин аутентификации `SRP` теперь использует алгоритм `SHA-256` для проверки клиента как на стороне сервера, так и на стороне клиента (см. настройки `AuthServer` и `AuthClient` в файле `firebird.conf`). Для обеспечения обратной совместимости клиент использовует старый плагин `Srp` (который реализует алгоритм `SHA-1`) в качестве запасного варианта. Такая настройка позволяет взаимодействовать с серверами, которые не используют `Srp256` (доступен с версии 3.0.4).

UseFileSystemCache

Параметр `UseFileSystemCache` определяет использовать ли кэш файловой системы сервером. Параметр имеет логический тип. Параметр `FileSystemCacheThreshold` учитывается, только если не задан параметр `UseFileSystemCache`.

```
UseFileSystemCache = true
```

InlineSortThreshold

Параметр `InlineSortThreshold` определяет, как обрабатываются неключевые поля во время сортировки: сохраняются внутри блока сортировки или повторно извлекаются со страниц после сортировки. Параметр имеет целочисленный тип. Значение параметра `InlineSortThreshold` определяет максимальный размер записей (в байтах), который может быть сохранен внутри блока сортировки. Если значение параметра `InlineSortThreshold` равно 0, то неключевые поля будут извлекаться со страниц после сортировки.

```
InlineSortThreshold = 1000
```

PolicyPlugin

Параметр `PolicyPlugin` определяет, использовать ли политики безопасности. По умолчанию параметр не указан, то есть политики не используются. Для включения политик безопасности укажите `PolicyPlugin = Policy`, тогда попытка пройти аутентификацию будет осуществляться для каждого плагина, указанного в `AuthServer`, а решение о предоставлении доступа будет принято плагином `Policy`.

```
PolicyPlugin = Policy
```

5.2 Изменённые параметры конфигурации

WireCryptPlugin

Добавлен новый вариант плагина `ChaCha#20`. В нем используется не 32-битный, а 64-битный внутренний счетчик. Значение по умолчанию для этого параметра теперь `ChaCha64, ChaCha, Arc4`.

5.3 Удалённые параметры конфигурации

RemotePipeName

Этот параметр был удален вместе с удалением поддержки протокола `WNET` для `Windows`.

TcpLoopbackFastPath

Этот параметр был удален, поскольку Microsoft не рекомендует использовать опцию сокета `SIO_LOOPBACK_FAST_PATH`.

Глава 6

Безопасность

6.1 Системные привилегии

Пользователю можно назначить часть прав администратора БД, делегировав ему роль с системными привилегиями.

Для этого сначала нужно создать роль с системными привилегиями:

```
CREATE ROLE <имя роли>
SET SYSTEM PRIVILEGES TO <сис.привилегия> [,<сис.привилегия> ...];
```

Таблица 6.1 — Системные привилегии

Привилегия	Описание
USER_MANAGEMENT	Управление пользователями.
READ_RAW_PAGES	Чтение страниц в сыром формате используя Attachment::getInfo()
CREATE_USER_TYPES	Создание, изменение и удаление не системных записей в таблице RDB$USER_TYPES.
USE_NBACKUP_UTILITY	Использование nbackup для создания резервных копий
CHANGE_SHUTDOWN_MODE	Закрытие базы данных (shutdown) и возвращение её в online.
TRACE_ANY_ATTACHMENT	Трассировка чужих пользовательских сессий.
MONITOR_ANY_ATTACHMENT	Мониторинг (MON$ таблицы) чужих пользовательских сессий.
ACCESS_SHUTDOWN_DATABASE	Доступ к базе данных в режиме shutdown.
CREATE_DATABASE	Создание новой базы данных (хранится в базе данных пользователей security.db).
DROP_DATABASE	Удаление текущей БД.
USE_GBAK_UTILITY	Использование утилиты или сервиса gbak.
USE_GSTAT_UTILITY	Использование утилиты или сервиса gstat.
USE_GFIX_UTILITY	Использование утилиты или сервиса gfix.
IGNORE_DB_TRIGGERS	Разрешает игнорировать триггеры на события БД.
CHANGE_HEADER_SETTINGS	Изменение параметров на заголовочной странице БД.
SELECT_ANY_OBJECT_IN_DATABASE	Выполнение оператора SELECT из всех селективных объектов (таблиц, представлений, хранимых процедур выбора).
ACCESS_ANY_OBJECT_IN_DATABASE	Доступ (любым способом) к любому объекту БД.
MODIFY_ANY_OBJECT_IN_DATABASE	Изменение любого объекта БД.
CHANGE_MAPPING_RULES	Изменение правил отображения при аутентификации.
USE_GRANTED_BY_CLAUSE	Использование GRANTED BY в операторах GRANT и REVOKE.
GRANT_REVOKE_ON_ANY_OBJECT	Выполнение операторов GRANT и REVOKE для любого объекта БД.

(разрыв таблицы)

(разрыв таблицы)

Привилегия	Описание
GRANT_REVOKE_ANY_DDL_RIGHT	Выполнение операторов GRANT и REVOKE для выдачи DDL привилегий.
CREATE_PRIVILEGED_ROLES	Создание привилегированных ролей (с использование SET SYSTEM PRIVILEGES).
GET_DBCRYPT_KEY_NAME	Получение имени ключа шифрования.
MODIFY_EXT_CONN_POOL	Управление пулом внешних соединений.
REPLICATE_INTO_DATABASE	Использование API репликации для загрузки наборов изменений в базу данных.
PROFILE_ANY_ATTACHMENT	Профилирование подключений других пользователей.

6.1.1 Системные привилегии в операторе GRANT

На более низком уровне новый тип получателя SYSTEM PRIVILEGE позволяет SYSDBA предоставлять и отбирать определенные привилегии доступа к объектам базы данных для указанной системной привилегии. Например:

```
GRANT ALL ON PLG$SRP_VIEW TO SYSTEM PRIVILEGE USER_MANAGEMENT
```

предоставляет пользователям с привилегией USER_MANAGEMENT все права на представление, которое используется в SRP плагине управления пользователями.

6.1.2 Назначение системной привилегии на роль

В синтаксисе операторов CREATE ROLE и ALTER ROLE появились новые пункты для прикрепления списка желаемых системных привилегий к новой или существующей роли.

```
CREATE ROLE name SET SYSTEM PRIVILEGES TO <сис. привилегия> {, <сис. привилегия> {, ..
.  <сис. привилегия> }}
ALTER ROLE name SET SYSTEM PRIVILEGES TO <сис. привилегия> {, <сис. привилегия> {, ...
 <сис. привилегия> }}
```

Оба оператора назначают список системных привилегий для роли. Оператор ALTER ROLE очищает привилегии, ранее назначенные роли, перед назначением нового списка.

Системные привилегии позволяют производить очень тонкую настройку, поэтому иногда вам нужно будет выдать более 1 системной привилегии для выполнения какой-либо задачи. Например, необходимо выдать IGNORE_DB_TRIGGERS совместно с USE_GSTAT_UTILITY, потому что gstat должен игнорировать триггера на события БД.

Удалить системнные привилегии, назначенные роли, можно следующим образом:

```
ALTER ROLE <имя роли> DROP SYSTEM PRIVILEGES
```

Функция RDB$SYSTEM_PRIVILEGE

Функция RDB$SYSTEM_PRIVILEGE возвращает используется ли системная привилегия текущим соединением:

```
RDB$SYSTEM_PRIVILEGE (<системная привилегия>)
```

6.2 Кумулятивное действие ролей

Ролям могут быть грантованы другие роли. В СУБД РЕД База Данных действует принцип кумулятивного действия ролей. Это значит, что, привилегии конкретной роли - это объединение привилегий, выданных этой роли, и привилегий ролей, назначенных этой роли.

Правила кумулятивного действия ролей:

- если пользователь не указывает роль при подключении к серверу, то он получает права только тех ролей, которые ему назначены с DEFAULT;
- если пользователь при подключении указал конкретную роль, то он получает только её привилегии и привилегии ролей, которые ему назначены с DEFAULT;
- пользователь может с помощью оператора SET ROLE сменить роль, указанную при подключении. В этом случае привилегии пользователя CURRENT_USER будут складываться из привилегий роли, назначенной оператором SET ROLE и привилегии ролей, которые ему назначены с DEFAULT;
- при подключении происходит проверка, что данная роль существует и назначена данному пользователю;
- циклические ссылки ролей друг на друга недопустимы.

Назначение и отбор у ролей прав других ролей происходит аналогично назначению и отбору прав у пользователей или у ролей:

```
GRANT Role1 TO Role2;
REVOKE Role2 FROM Role1;
```

6.2.1 Ключевое слово DEFAULT

Если используется ключевое слово DEFAULT, то роль (роли) будет использоваться пользователем или ролью каждый раз, даже если она не была указана явно. При подключении пользователь получит привилегии всех ролей, которые были назначены пользователю с использованием ключевого слова DEFAULT. Если пользователь укажет свою роль при подключении, то получит привилегии этой роли (если она была ему назначена) и привилегии всех ролей назначенных ему с использованием ключевого слова DEFAULT.

6.2.2 Предложение WITH ADMIN OPTION

Необязательное предложение WITH ADMIN OPTION позволяет пользователям, указанным в списке пользователей, передавать свои роли другому пользователю или роли. Полномочия роли могут быть переданы кумулятивно, только если каждая роль в последовательности ролей назначена с использованием WITH ADMIN OPTION.

6.2.3 Пример кумулятивного действия ролей

```
CREATE DATABASE 'LOCALHOST:/TMP/CUMROLES.FDB';
CREATE TABLE T(I INTEGER);
CREATE ROLE TINS;
CREATE ROLE CUMR;
GRANT INSERT ON T TO TINS;
GRANT SELECT ON T TO CUMR;
GRANT DEFAULT TINS TO USER1;
GRANT CUMR TO USER1;
CONNECT 'LOCALHOST:/TMP/CUMROLES.FDB' USER 'USER1' PASSWORD 'PAS' ROLE 'CUMR';
INSERT INTO T VALUES (1);
SELECT * FROM T;
```

6.2.4 Функция RDB$ROLE_IN_USE

Функция RDB$ROLE_IN_USE показываает используется ли роль текущим пользователем.

```
RDB$ROLE_IN_USE (<имя роли>)
```

Данная функция позволяет проверить использование любой роли: указанной явно (при входе в систему или изменённой с помощью оператора SET ROLE) и назначенной неявно (роли назначенные пользователю с использованием предложения DEFAULT).

Узнать список активных в данный момент ролей можно следующим образом:

```
SELECT * FROM RDB$ROLES WHERE RDB$ROLE_IN_USE(RDB$ROLE_NAME)
```

6.3 Функция SQL SECURITY

Эта новая возможность позволяет определять выполняемые объекты (триггеры, хранимые процедуры, хранимые функции) для запуска в контексте условия SQL SECURITY, как определено в стандартах SQL (2003, 2011).

Конструкция SQL SECURITY имеет два контекста: INVOKER и DEFINER. Контекст INVOKER соответствует привилегиям, доступным CURRENT_USER или вызывающему объекту, а DEFINER - привилегиям, доступным владельцу объекта.

Свойство SQL SECURITY - это необязательная часть определения объекта, которая может быть применена к объекту с помощью операторов DDL. Это свойство нельзя исключить из функций, процедур и пакетов, но его можно изменить с INVOKER на DEFINER и наоборот.

Это не то же самое, что привилегии SQL, которые применяются к пользователям и некоторым типам объектов базы данных, чтобы предоставить им различные виды доступа к объектам базы данных. Когда исполняемому объекту требуется доступ к таблице, представлению или другому исполняемому объекту, целевой объект будет недоступен, если вызов не обладает необходимыми привилегиями. То есть по умолчанию все исполняемые объекты имеют свойство SQL SECURITY INVOKER. Любой вызов, не обладающий необходимыми привилегиями, будет отклонен.

Если к процедуре применено свойство SQL SECURITY DEFINER, то пользователь или процедура смогут выполнить ее, если владельцу предоставлены необходимые привилегии, при этом пользователю не нужно специально предоставлять эти привилегии.

В итоге:

- Если установлено значение INVOKER, то права доступа для выполнения вызова объекта определяются путем проверки активного набора привилегий текущего пользователя.
- Если установлено значение DEFINER, то вместо этого будут применяться права доступа владельца объекта, независимо от активного набора привилегий текущего пользователя.

Синтаксис:

```
CREATE TABLE table_name (...) [SQL SECURITY {DEFINER | INVOKER}]
ALTER TABLE table_name ... [{ALTER SQL SECURITY {DEFINER | INVOKER} | DROP SQL
SECURITY}]
CREATE [OR ALTER] FUNCTION function_name ... [SQL SECURITY {DEFINER | INVOKER}] AS ...
CREATE [OR ALTER] PROCEDURE procedure_name ... [SQL SECURITY {DEFINER | INVOKER}] AS .
..
CREATE [OR ALTER] TRIGGER trigger_name ... [SQL SECURITY {DEFINER | INVOKER} | DROP
SQL SECURITY] [AS ...]
CREATE [OR ALTER] PACKAGE package_name [SQL SECURITY {DEFINER | INVOKER}] AS ...

ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

6.3.1 Триггеры

Триггеры наследуют значение свойства SQL SECURITY от таблицы, но его можно явно переопределить. Если свойство изменяется для таблицы, триггеры, которые не несут переопределенного свойства, не увидят результат изменения до следующей загрузки триггера в кэш метаданных.

Чтобы явно удалить опцию SQL SECURITY из триггера, например, с именем tr_ins, вы можете выполнить команду:

```
alter trigger tr_ins DROP SQL SECURITY;
```

Чтобы снова установить значение SQL SECURITY INVOKER, выполните:

```
alter trigger tr_ins sql security invoker;
```

6.3.2 Примеры использования SQL SECURITY

1. Если для таблицы t установлено значение DEFINER, пользователю US требуется только привилегия SELECT для t. Если бы было установлено значение INVOKER, пользователю также потребовалась бы привилегия EXECUTE для функции f.

```
set term ^;
create function f() returns int
as
begin
    return 3;
end^
set term ;^
create table t (i integer, c computed by (i + f())) SQL SECURITY DEFINER;
insert into t values (2);
grant select on table t to user us;

commit;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
connect 'localhost:/tmp/7.fdb' user us password 'pas';
select * from t;
```

2. С DEFINER, установленным для функции f, пользователю US нужна только привилегия EXECUTE для f. Если было бы установлено INVOKER, пользователю также понадобилась бы привилегия INSERT для таблицы t.

```
set term ^;
create function f (i integer) returns int SQL SECURITY DEFINER
as
begin
  insert into t values (:i);
  return i + 1;
end^
set term ;^
grant execute on function f to user us;

commit;

connect 'localhost:/tmp/59.fdb' user us password 'pas';
select f(3) from rdb$database;
```

3. Если для процедуры p установлено значение DEFINER, пользователю US требуется только привилегия EXECUTE для p. Если установлено INVOKER, пользователю или процедуре также потребовалась бы привилегия INSERT для таблицы t.

```
set term ^;
create procedure p (i integer) SQL SECURITY DEFINER
as
begin
  insert into t values (:i);
end^
set term ;^

grant execute on procedure p to user us;
commit;

connect 'localhost:/tmp/17.fdb' user us password 'pas';
execute procedure p(1);
```

4. Если для триггера tr_ins установлено значение DEFINER, пользователю US нужна только привилегия INSERT на таблице tr. Если установлено INVOKER, то либо пользователю, либо триггеру также потребовалась бы привилегия INSERT для таблицы t.

```
create table tr (i integer);
create table t (i integer);
set term ^;
create trigger tr_ins for tr after insert SQL SECURITY DEFINER
as
begin
  insert into t values (NEW.i);
end^
set term ;^
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
grant insert on table tr to user us;

commit;

connect 'localhost:/tmp/29.fdb' user us password 'pas';
insert into tr values(2);
```

Результат будет таким же, если для таблицы TR указан SQL SECURITY DEFINER:

```
create table tr (i integer) SQL SECURITY DEFINER;
create table t (i integer);
set term ^;
create trigger tr_ins for tr after insert
as
begin
  insert into t values (NEW.i);
end^
set term ;^
grant insert on table tr to user us;

commit;

connect 'localhost:/tmp/29.fdb' user us password 'pas';
insert into tr values(2);
```

5. Если для пакета pk установлено значение DEFINER, пользователю US нужна только привилегия EXECUTE на pk. Если установлено INVOKER, то пользователю или пакету также потребовалась бы привилегия INSERT для таблицы t.

```
create table t (i integer);
set term ^;
create package pk SQL SECURITY DEFINER
as
begin
    function f(i integer) returns int;
end^

create package body pk
as
begin
    function f(i integer) returns int
    as
    begin
      insert into t values (:i);
      return i + 1;
    end
end^
set term ;^
grant execute on package pk to user us;

commit;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
connect 'localhost:/tmp/69.fdb' user us password 'pas';
select pk.f(3) from rdb$database;
```

6.4 Встроенные криптографические функции

6.4.1 ENCRYPT

Функция ENCRYPT шифрует данные с использованием симметричного шифра.

```
ENCRYPT (<строка> [USING <алгоритм шифрования>] [MODE <режим шифрования>]
  KEY <ключ шифрования> [IV <вектор инициализации>]
  [ <порядок байтов счётчика>] [CTR_LENGTH <длина счётчика>]
  [COUNTER <начальное значение счётчика>])

<алгоритм шифрования> ::= { <блочные алгоритмы> | <потоковые алгоритмы> }
<блочные алгоритмы> ::= { AES | ANUBIS | BLOWFISH | KHAZAD | RC5 | RC6 | SAFER+ |
TWOFISH | XTEA }
<потоковые алгоритмы> ::= { CHACHA20 | RC4 | SOBER128 }
<режим шифрования> ::= { CBC | CFB | CTR | ECB | OFB }
<порядок байтов счётчика> ::= { CTR_BIG_ENDIAN | CTR_LITTLE_ENDIAN }
```

Тип возвращаемого результата: BLOB или VARBINARY.

Здесь:

- <строка> — выражение строкового типа или BLOB, которое необходимо зашифровать. Размеры строк передаваемых в эту функцию должны соответствовать требованиям выбранного алгоритма и режима.
- <режим шифрования> - обязателен для блочных алгоритмов шифрования.
- <вектор инициализации> - должен быть указан для всех блочных алгоритмов шифрования за исключением ECB и всех потоковых алгоритмов шифрования за исключением RC4.
- <порядок байтов счётчика> - может быть указан только в режиме CTR. По умолчанию используется CTR_LITTLE_ENDIAN.
- <длина счётчика> - (в байтах) может быть указана только в режиме CTR. По умолчанию равна длине вектора инициализации IV.
- <начальное значение счётчика> - может быть указана только для алгоритма CHACHA20. По умолчанию равно 0.

6.4.2 DECRYPT

Функция DECRYPT дешифрует данные с использованием симметричного шифра.

```
DECRYPT (<строка> [USING <алгоритм шифрования>] [MODE <режим шифрования>]
  KEY <ключ шифрования> [IV <вектор инициализации>] [ <порядок байтов счётчика>] [CTR_
LENGTH <длина счётчика>] [COUNTER <начальное значение счётчика>])

<алгоритм шифрования> ::= { <блочные алгоритмы> | <потоковые алгоритмы> }

<блочные алгоритмы> ::= { AES | ANUBIS | BLOWFISH | KHAZAD | RC5 | RC6 | SAFER+ |
TWOFISH | XTEA }
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<потоковые алгоритмы> ::= { CHACHA20 | RC4 | SOBER128 }

<режим шифрования> ::= { CBC | CFB | CTR | ECB | OFB }

<порядок байтов счётчика> ::= { CTR_BIG_ENDIAN | CTR_LITTLE_ENDIAN
```

Тип возвращаемого результата: BLOB или VARBINARY.

Здесь:

- <строка> — выражение строкового типа или BLOB, которое необходимо зашифровать. Размеры строк передаваемых в эту функцию должны соответствовать требованиям выбранного алгоритма и режима.
- <режим шифрования> - обязателен для блочных алгоритмов шифрования.
- <вектор инициализации> - должен быть указан для всех блочных алгоритмов шифрования за исключением ECB и всех потоковых алгоритмов шифрования за исключением RC4.
- <порядок байтов счётчика> - может быть указан только в режиме CTR. По умолчанию используется CTR_LITTLE_ENDIAN.
- <длина счётчика> - (в байтах) может быть указана только в режиме CTR. По умолчанию равна длине вектора инициализации IV.
- <начальное значение счётчика> - может быть указана только для алгоритма CHACHA20. По умолчанию равно 0.

6.4.3 RSA_PRIVATE

Функция RSA_PRIVATE возвращает RSA закрытый ключ заданной длины (в байтах) в PKCS#1 формате как строку VARBINARY.

```
RSA_PRIVATE (<размер ключа>)
```

Тип возвращаемого результата VARBINARY.

6.4.4 RSA_PUBLIC

Функция RSA_PUBLIC возвращает RSA открытый ключ для заданного RSA закрытого ключа. Оба ключа должны быть в PKCS#1 формате.

```
RSA_PUBLIC (<RSA закрытый ключ>)
```

Тип возвращаемого результата VARBINARY.

6.4.5 RSA_ENCRYPT

Заполняет данные, используя заполнение OAEP, и шифрует их, используя открытый ключ RSA. Обычно используется для шифрования коротких симметричных ключей, которые затем используются в блочных шифрах для шифрования сообщения.

```
RSA_ENCRYPT (<данные> KEY <открытый RSA ключ> [LPARAM <тег>] [HASH <алгоритм
хэширования>])

<алгоритм хэширования> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Здесь:

- <данные> — строка или BLOB для шифрования.
- <открытый RSA ключ> — открытый RSA ключ, который возвращает функция RSA_PUBLIC.
- <тег> — дополнительный системный тег, который можно применять для определения того, какая система закодировала сообщение. Значением по умолчанию является NULL.
- <алгоритм хэширования> по умолчанию SHA256.

Тип возвращаемого результата VARBINARY.

6.4.6 RSA_DECRYPT

Расшифровывает с использованием закрытого ключа RSA, и удаляет OAEP дополненные данные.

```
RSA_DECRYPT (<данные> KEY <закрытый RSA ключ> [LPARAM <тег>] [HASH <алгоритм
хэширования>])

<алгоритм хэширования> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Здесь:

- <данные> — строка или BLOB для шифрования.
- <закрытый RSA ключ> — закрытый RSA ключ, который возвращает функция RSA_PRIVATE.
- <тег> — дополнительный системный тег, который можно применять для определения того, какая система закодировала сообщение. Значением по умолчанию является NULL.
- <алгоритм хэширования> по умолчанию SHA256.

Тип возвращаемого результата VARCHAR.

6.4.7 RSA_SIGN

Выполняет PSS-кодирование дайджеста сообщения для подписи и подписывает его с использованием закрытого ключа RSA. Возвращает подпись сообщения.

```
RSA_SIGN (<данные> KEY <закрытый RSA ключ> [HASH <алгоритм хэширования>] [SALT_LENGTH
<длина>])

<алгоритм хэширования> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Здесь:

- <данные> — строка или BLOB для кодирования.
- <закрытый RSA ключ> — закрытый RSA ключ, который возвращает функция RSA_PRIVATE.
- <алгоритм хэширования> по умолчанию SHA256.
- <длина> указывает на длину.

Тип возвращаемого результата VARBINARY.

6.4.8 RSA_VERIFY

Выполняет PSS-кодирование дайджеста сообщения для подписи и проверяет его цифровую подпись, используя открытый ключ RSA. Возвращает результат проверки подписи.

```
RSA_VERIFY (<данные> SIGNATURE <подпись> KEY <открытый RSA ключ> [HASH <алгоритм
хэширования>] [SALT_LENGTH <длина>])

<алгоритм хэширования> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Здесь:

- `<данные>` — строка или `BLOB` для кодирования.
- `<подпись>` должно быть значением возвращаемым функцией `RSA_SIGN`.
- `<открытый RSA ключ>` — открытый `RSA` ключ, который возвращает функция `RSA_PUBLIC`.
- `<алгоритм хэширования>` по умолчанию `SHA256`.
- `<длина>` указывает на длину желаемой соли и, как правило, должен быть небольшим. Хорошее значение от 8 до 16.

Тип возвращаемого результата `BOOLEAN`.

6.5 Управление пользователями

Пользователь, созданный с правами администратора в базе данных безопасности (с помощью `GRANT ADMIN ROLE`), больше не обязан иметь роль `RDB$ADMIN`, чтобы создавать, изменять или удалять других пользователей.

6.6 Отслеживание событий до установления действующего контекста безопасности

Привилегированная сессия трейса (например, администратором или пользователем с параметром `TRACE_ANY_ATTACHMENT`) теперь может сообщать о событиях (т.е. ошибках), происходящих до проверки контекста безопасности вложения.

Пример:

Установите конфликтующее отображение для пользователя:

```
# ./isql employee
Database: employee, User: SYSDBA
SQL> create user qq password 'qq';
SQL> create global mapping z1 using * from user qq to user z1;
SQL> create global mapping z2 using * from user qq to user z2;
SQL> ^D
```

Из-за конфликтующего отображения пользователь `QQ` не может подключиться к базе данных даже с действительными логином/паролем:

```
# ./isql localhost:employee -user qq -pas qq
Statement failed, SQLSTATE = 08004
Multiple maps found for QQ
Use CONNECT or CREATE DATABASE to specify a database
SQL> ^D
```

В выводе трейса можно увидеть следующее:

```
2023-03-17T13:38:41.5240 (25380:0x7f282c10c750) FAILED ATTACH_DATABASE
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
        employee (ATT_0, QQ, NONE, TCPv4:127.0.0.1/39474)
        /opt/firebird/bin/isql:25396
```

6.7 Трассировка события COMPILE

В РЕД Базе Данных 5 появилось новое событие трассировки: парсинг хранимых модулей. Оно позволяет отслеживать моменты парсинга хранимых модулей, затраченное время и самое главное — планы запросов внутри модулей PSQL. Отслеживание плана также возможно, если модуль PSQL уже был загружен до начала сеанса трассировки; в этом случае о плане будет сообщено во время первого выполнения, замеченного сеансом трассировки.

Для отслеживания события в конфигурационный файл аудита были добавлены следующие параметры:

- `log_procedure_compile` - включает отслеживание событий парсинга процедур;
- `log_function_compile` - включает отслеживание событий парсинга функций;
- `log_trigger_compile` - включает отслеживание событий парсинга триггеров.

6.8 Политики безопасности

В РЕД Базе Данных 5 включение политик безопасности вынесено в новый параметр конфигурации - `PolicyPlugin`. По умолчанию параметр не указан, то есть политики не используются.

Для включения политик безопасности укажите `PolicyPlugin = Policy`, тогда попытка пройти аутентификацию будет осуществляться для каждого плагина, указанного в `AuthServer`, а информация об успешном прохождении будет добавлена в `AuthBlock`. Затем плагин `Policy`, указанный в `PolicyPlugin`, проверяет `AuthBlock` и решает дать ли доступ пользователю.

Если `PolicyPlugin` не указан, то аутентификация завершается при первом успешном прохождении по методу, указанному в `AuthServer`.

Глава 7

Операторы управления

7.1 Управление пулом внешних соединений

Для управление пулом внешних соединений добавлен ряд новых операторов.

Для выполнения этих операторов требуется роль с системной привилегией MODIFY_EXT_CONN_POOL.

7.1.1 ALTER EXTERNAL CONNECTIONS POOL

Оператор ALTER EXTERNAL CONNECTIONS POOL добавлен для управление пулом внешних соединений.

Синтаксис оператора:

```
ALTER EXTERNAL CONNECTIONS POOL { <параметры> }
```

При его подготовке они описываются как DDL операторы, но имеют немедленный эффект: то есть они выполняются немедленно и полностью, не дожидаясь фиксации транзакции.

Операторы могут выполняться из любого соединения, а изменения применяются к экземпляру пула в памяти текущего процесса сервера. В случае архитектуры Classic изменения, внесенные в процесс, не влияют на другие Classic-процессы.

После перезапуска сервер будет использовать настройки пула, указанные в firebird.conf с помощью ExtConnPoolSize и ExtConnPoolLifeTime.

ALTER EXTERNAL CONNECTIONS POOL SET SIZE

Оператор позволяет устанавливать максимальное количество бездействующих соединений в пуле внешних соединений.

```
ALTER EXTERNAL CONNECTIONS POOL SET SIZE <размер>
```

Допустимые значения от 0 до 1000. Нулевое значение означает что пул выключен. Значение по умолчанию определяется в firebird.conf (параметр ExtConnPoolSize).

ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME

Оператор позволяет устанавливать время жизни бездействующих соединений в пуле внешних соединений.

```
ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME <значение> {SECOND | MINUTE | HOUR}
```

Допустимые значения от 1 секунды до 24 часов. Значение по умолчанию определяется в firebird.conf (параметр ExtConnPoolLifeTime).

ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL

Оператор позволяет закрыть все бездействующие соединения в пуле внешних соединений.

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL
```

Все активные соединения будут отсоединены от пула (такие соединения будут немедленно закрыты, когда они не будут использоваться).

ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST

Оператор позволяет закрыть бездействующие соединения в пуле внешних соединений, у которых истекло время жизни.

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST
```

7.2 ALTER SESSION RESET

Оператор сбрасывает сеансовое окружение (подключения) к исходному состоянию. Эта функция полезна если сеанс используется повторно, вместо того чтобы производить отключение/подключение.

```
ALTER SESSION RESET
```

Данный оператор делает следующее:

- сбрасывает установленные параметры DECFLOAT (BIND, TRAP и ROUND) в значения по умолчанию;
- сбрасывает таймауты сессии и оператора в 0;
- удаляет все контекстные переменные из пространства имён USER_SESSION;
- очищает содержимое всех используемых глобальных таблиц уровня соединения (COMMIT PRESERVE ROWS);
- сбрасывает роль в значение переданное в DPB (указанное при подключении) и очищает кэш привелегий (если роль была изменена с помощью оператора SET ROLE);
- текущая активная транзакция откатывается и стартуется новая с теми же параметрами, после рестарта.

Если в текущей активной транзакции были произведены изменения, то будет выдано предупреждение.

Если в текущей сессии активны другие транзакции, то будет выдана ошибка. При проверке транзакций перед сбросом сессии подготовленные 2PC транзакции игнорируются.

7.3 Управление часовыми поясами

Добавлен оператор для управления часовым поясом текущего соединения.

7.3.1 SET TIME ZONE

Оператор позволяет изменить часовой пояс сеанса (текущего подключения).

```
SET TIME ZONE {'<часовой пояс>' | LOCAL}

<часовой пояс> ::=
  <регион часового пояса> |
  [+/-] <смещения часов относительно GMT> [: <смещения минут относительно GMT>]
```

Данный SQL оператор работает вне механизма управления транзакциями и вступают в силу немедленно.

```
set time zone '-02:00';
set time zone 'America/Sao_Paulo';
set time zone local;
```

7.3.2 SET TIME ZONE BIND

Оператор изменяет привязку часового пояса сеанса для совместимости со старыми клиентами.

```
SET TIME ZONE BIND { NATIVE | LEGACY }
```

Значение по умолчанию - `NATIVE`, что означает, что выражения `TIME WITH TIME ZONE` и `TIMESTAMP WITH TIME ZONE` возвращаются с новыми типами данных для клиента.

Старые клиенты могут не понимать новые типы данных, поэтому можно определить привязку к `LEGACY`, и выражения будут возвращены как `TIME WITHOUT TIME ZONE` и `TIMESTAMP WITHOUT TIME ZONE` с соответствующим преобразованием.

7.4 Управление таймаутами

Временные интервалы для таймаутов сеансов и операторов можно настроить на уровне сеанса с помощью операторов `SET SESSION IDLE TIMEOUT` и `SET STATEMENT TIMEOUT`, соответственно.

7.4.1 SET SESSION IDLE TIMEOUT

Оператор позволяет устанавливать таймаут простоя соединения на уровне текущего соединения.

```
SET SESSION IDLE TIMEOUT <значение> [HOUR | MINUTE | SECOND]
```

В качестве параметра выступает значение таймаута простоя в указанных единицах измерения времени. Если единица измерения времени не указана, то по умолчанию значение таймаута измеряется в минутах.

Данный `SQL` оператор работает вне механизма управления транзакциями и вступают в силу немедленно.

```
SET SESSION IDLE TIMEOUT 8 HOUR;
```

7.4.2 SET STATEMENT TIMEOUT

Оператор позволяет установить таймаут выполнения SQL операторов на уровне текущего соединения.

```
SET STATEMENT TIMEOUT <значение> [HOUR | MINUTE | SECOND | MILLISECOND]
```

В качестве параметра выступает значение таймаута выполнения SQL операторов в указанных единицах измерения времени. Если единица измерения времени не указана, то по умолчанию значение таймаута измеряется в секундах.

Данный SQL оператор работает вне механизма управления транзакциями и вступают в силу немедленно.

```
SET STATEMENT TIMEOUT 3 MINUTE;
```

7.5 Настройка параметров DECFLOAT

7.5.1 SET DECFLOAT BIND

Для того чтобы старые приложения умели работать с типом DECFLOAT можно настроить отображение значений DECFLOAT на другие доступные типы данных с помощью оператора SET DECFLOAT BIND.

```
SET DECFLOAT BIND <тип для привязки>

<тип для привязки> ::= NATIVE
                     | CHAR[ACTER]
                     | DOUBLE PRECISION
                     | BIGINT[, <точность>]
```

Допустимые типы для привязки:

- NATIVE — используется IEEE754 двоичное представление. Идеальная поддержка для дальнейшей обработки, но с плохой точностью;
- CHAR[ACTER] — ASCII строка. Идеальная точность, но плохая поддержка для дальнейшей обработки;
- DOUBLE PRECISION — используется 8 байтное представление с плавающей точкой, тоже самое что и для полей типа DOUBLE PRECISION. Идеальная поддержка для дальнейшей обработки, но с плохой точностью;
- BIGINT — используется целое с указанным масштабом, тоже самое что поле NUMERIC(18, <точность>). Хорошая поддержка дальнейшей обработки и требуемая точность, но диапазон значений очень ограничен.

Привязка к ASCII строке будет иметь тип CHAR(23) для DECFLOAT(16) и CHAR(42) для DECFLOAT(34).

Строковое представление зависит от значения DECFLOAT: если оно является экспоненциальным, а требования к точности позволяют отображать значение без использования научной записи, используется полностью выписанный формат.

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

7.5.2 SET DECFLOAT TRAPS TO

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

```
SET DECFLOAT TRAPS TO <ситуация>[, <ситуация>...]

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

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

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

7.5.3 SET DECFLOAT ROUND

Оператор позволяет изменять режим округления для типа DECFLOAT для текущей сессии.

```
SET DECFLOAT ROUND <режим округления>
```

Поддерживаются следующие режимы округления совместимые со стандартом IEEE:

Таблица 7.1 — Режимы округления

Режим округления	Описание
CEILING	Округление сверху. Если все отбрасываемые цифры равны нулю или знак числа отрицателен, последняя неотбрасываемая цифра не меняется. В противном случае последняя неотбрасываемая 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 неотбрасываемая цифра инкрементируется на единицу (округляется в большую сторону). В противном случае отбрасываемые значения игнорируются. Этот режим используется по умолчанию.

(разрыв таблицы)

(разрыв таблицы)

Режим округления	Описание
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 неотбрасываемая 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 неотбрасываемая 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 неотбрасываемая 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 неотбрасываемая цифра не меняется. В противном случае (знак отрицателен) последняя неотбрасываемая цифра инкрементируется на единицу.
REROUND	Округление к большему значению, если округляется 0 или 5, в противном случае округление происходит к меньшему значению.

7.6 Настройка правил приведения типов данных

Оператор SET BIND настраивает правила приведения к типу данных для текущего сеанса. Этот оператор позволяет заменять один тип данных другим при выполнении клиент-серверных взаимодействий.

```
SET BIND OF <заменяемый тип> TO <итоговый тип>

<заменяемый тип> ::=
    <scalar_datatype>
  | <blob_datatype>
  | TIME ZONE
  | VARCHAR | CHARACTER | CHAR VARYING

<итоговый тип> ::=
    <scalar_datatype>
  | <blob_datatype>
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    | VARCHAR | CHARACTER | CHAR VARYING
    | LEGACY | NATIVE | EXTENDED
    | EXTENDED TIME WITH TIME ZONE
    | EXTENDED TIMESTAMP WITH TIME ZONE
```

Если используется неполное определение типа (например, `CHAR` вместо `CHAR(n)`) в типе данных, который нужно заменить, то принудительное преобразование выполняется для всех столбцов `CHAR`. Специальный неполный тип `TIME ZONE` обозначает `TIME WITH TIME ZONE` и `TIMESTAMP WITH TIME ZONE`. Если в типе данных, к которому нужно преобразовать, используется неполное определение типа, сервер автоматически определяет недостающие сведения об этом типе на основе исходного столбца.

Изменение привязки типа данных `NUMERIC` или `DECIMAL` не влияет на базовый целочисленный тип. Напротив, изменение привязки целочисленного типа данных также влияет на соответствующие `NUMERIC` или `DECIMAL` (например, `SET BIND OF INT128 TO DOUBLE PRECISION` также будет преобразовывать `NUMERIC` и `DECIMAL` с точностью более 19, поскольку они используют `INT128` в качестве базового типа).

Специальный тип `LEGACY` используется, когда тип данных, отсутствующий в предыдущей версии СУБД, должен быть представлен способом, понятным для устаревшего клиентского программного обеспечения (возможно, с некоторой потерей данных). Правила приведения, применяемые в этом случае, описаны в таблице ниже.

Таблица 7.2 — Правила преобразования NATIVE к LEGACY

NATIVE	LEGACY
BOOLEAN	CHAR(5)
DECFLOAT	DOUBLE PRECISION
INT128	BIGINT
TIME WITH TIME ZONE	TIME WITHOUT TIME ZONE
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITHOUT TIME ZONE

Установка преобразования к `NATIVE` сбрасывает существующее правило преобразования для этого типа данных и он будет передаваться в исходном формате.

Выполнение `ALTER SESSION RESET` приведет к возвращению к правилам преобразования по умолчанию.

Пример работы оператора `SET BIND`:

```
SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;      --native

CAST
======================
123.45

SET BIND OF DECFLOAT TO DOUBLE PRECISION;
SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;      --double

CAST
======================
123.4500000000000

SET BIND OF DECFLOAT(34) TO CHAR;
SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;      --still double
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
CAST
======================
123.4500000000000

SELECT CAST('123.45' AS DECFLOAT(34)) FROM RDB$DATABASE;      --text

CAST
========================================
123.45
```

7.7 SET OPTIMIZE

Оператор определяет, что нужно оптимизировать: получение первых строк или всех.

```
SET OPTIMIZE <режим оптимизации>

<режим оптимизации> ::=
    FOR {FIRST | ALL} ROWS
  | TO DEFAULT
```

Оператор `SET OPTIMIZE` позволяет изменить стратегию оптимизатора на уровне текущей сессии.

Существует две стратегии оптимизации запросов:

- `FIRST ROWS` - оптимизатор строит план запроса так, чтобы наиболее быстро извлечь только первые строки запроса;
- `ALL ROWS` - оптимизатор строит план запроса так, чтобы наиболее быстро извлечь все строки запроса.

По умолчанию используется стратегия оптимизации указанная в параметре `OptimizeForFirstRows` конфигурационного файла `firebird.conf` или `database.conf`. `OptimizeForFirstRows = false` соответствует стратегии `ALL ROWS`, `OptimizeForFirstRows = true` соответствует стратегии `FIRST ROWS`.

Стратегия оптимизации может быть переопределена на уровне `SQL` оператора с помощью предложения `OPTIMIZE FOR`.

Глава 8

Язык описания данных (DDL)

8.1 Увеличена длина имен объектов

Максимальная длина имен объектов увеличена с 31 байта до 63 байт.

Многобайтовые идентификаторы теперь также могут быть длинными. Например, предыдущее ограничение позволяло использовать только 15 кириллических символов, а теперь их может быть до 63.

Двойные кавычки вокруг имени столбца не учитываются.

8.1.1 Ограничение длины

Если требуется ограничить максимальный размер имен объектов глобально или для отдельных баз данных, в файлах `firebird.conf` и/или `databases.conf` доступны два новых параметра конфигурации: `MaxIdentifierByteLength` и `MaxIdentifierCharLength`.

8.2 Новые типы данных

8.2.1 INT128

Тип данных `INT128` представляет собой 128-битное целое. Числа типа `INT128` находятся в диапазоне от -2^{127} до $+2^{127} - 1$.

Числа типа `INT128` могут быть заданы в шестнадцатеричном виде с 17 — 32 шестнадцатеричными цифрами.

Операции сложения и вычитания для типа данных `INT128` выполняются обычным образом. Следует только быть особенно внимательными при выполнении операций умножения и деления. В этих операциях результат будет иметь количество дробных знаков, равное сумме дробных знаков обоих операндов. То есть результатом операции деления двух целых чисел:

```
1 / 3
```

будет 0, потому что сумма дробных знаков операндов равна нулю и целая часть в результате выполнения операции деления будет равна нулю. Результат является целочисленным.

Для числовых типов данных могут использоваться агрегатные функции, определенные в `SQL` — `MIN`, `MAX`, `SUM`, `AVG` и множество других встроенных функций.

8.2.2 TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE

Тип данных `TIME WITH TIME ZONE` хранит время в часах, минутах, секундах и десятитысячных долях секунды с учётом часового пояса. Диапазон: от `00:00:00.0000` до `23:59:59.9999`.

Тип данных `TIMESTAMP WITH TIME ZONE` - комбинация (объединение) даты и времени с учётом часового пояса.

8.2.3 DECFLOAT

Тип DECFLOAT, основанный на стандарте IEEE 754-2008 и совместимый с SQL:2016, позволяет точно хранить десятичные числа с плавающей запятой. Количество значащих цифр (точность) – 16 или 34.

Если точность не указана, то по умолчанию используется точность 34 значащих цифры. Все промежуточные вычисления осуществляются с использованием 34-значной точности. Точность столбца или домена DECFLOAT указана в системной таблице RDB$FIELDS в RDB$FIELD_PRECISION.

Использование DECFLOAT

Длина литералов

Длина литералов DECFLOAT не может превышать 1024 символов. Для более длинных значений требуется научная нотация. Например, 0.0<1020 нулей>11 не может быть использовано в качестве литерала. Эквивалент в научной нотации - 1.1E-1022. Аналогично, 10<1022 нулей>0 может быть представлено как 1.0E1024.

Использование в стандартных функциях

Ряд стандартных скалярных функций можно использовать с выражениями и значениями типа DECFLOAT. К ним относятся: ABS, CEILING, EXP, FLOOR, LN, LOG, LOG10, POWER, SIGN, SQRT.

Все математические, агрегатные и статистические агрегатные функции поддерживают работу со значениями типа DECFLOAT.

Кроме этого, созданы *4 функции* специально для поддержки типа DECFLOAT: COMPARE_DECFLOAT, NORMALIZE_DECFLOAT, QUANTIZE, TOTALORDER.

Оператор *SET DECFLOAT* позволяет изменять свойства типа данных DECFLOAT на уровне подключения.

8.3 Улучшения DDL

8.3.1 Увеличена точность NUMERIC и DECIMAL

NUMERIC и DECIMAL теперь могут быть определены с точностью до 38 цифр. Любое значение с точностью выше 18 цифр будет храниться как 38-значное число. Также появился тип данных INT128, который представляет собой 128-битное целое (до 38 десятичных цифр).

Синтаксис:

```
INT128
NUMERIC (<точность>, <масштаб>)
DECIMAL (<точность>, <масштаб>)

1 <= <точность> <= 38
1 <= <масштаб> <= 38
```

Точность указывает количество хранящихся знаков. Масштаб задаёт количество знаков после разделителя. Масштаб должен быть меньше или равен точности.

Примеры:

- Объявление переменной из 25 цифр, чтобы она воспринималась как целое число:

```
DECLARE VARIABLE VAR1 DECIMAL(25);
```

- Столбец для хранения до 38 цифр с 19 знаками после запятой:

```
CREATE TABLE TABLE1 (FIELD1 NUMERIC(38, 19));
```

- Определение процедуры с входным параметром, заданным как 128-битное целое число:

```
CREATE PROCEDURE PROC1 (PAR1 INT128) AS BEGIN END;
```

8.3.2 Тип данных FLOAT соответствует стандарту

Тип данных FLOAT был усовершенствован для соответствия стандарту SQL:2016. Поддерживаются следующие приблизительные числовые типы: 32-битной одинарной точности и 64-битной двойной точности. Эти типы доступны со следующими именами типов по стандарту SQL:

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

Дополнительные типы:

- LONG FLOAT — 64-битный тип двойной точности (синоним типа DOUBLE PRECISION);
- LONG FLOAT(p), где p — точность в двоичных числах. 1 <= p <= 53 — 64-битное двойной точности (синоним типа DOUBLE PRECISION)

Точность 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.

8.3.3 Типы данных для поддержки временных зон

Синтаксис типов данных TIMESTAMP и TIME был расширен аргументами, определяющими наличие часового пояса:

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

По умолчанию TIME и TIMESTAMP используют WITHOUT TIME ZONE.

8.3.4 Алиасы для бинарных строковых типов

Типы данных BINARY(n), VARBINARY(n) и BINARY VARYING(n) были добавлены в качестве псевдонимов для определения строковых столбцов в CHARACTER SET OCTETS.

BINARY(n) алиас для CHAR(n) CHARACTER SET OCTETS, а VARBINARY(n) и BINARY VARYING(n) алиасы для VARCHAR(n) CHARACTER SET OCTETS и друг для друга.

8.3.5 Улучшения типа IDENTITY

Столбец идентификации представляет собой столбец, связанный с внутренним генератором последовательностей, значение которого автоматически устанавливается, когда он запущен в операторе INSERT.

В версии 5 IDENTITY был расширен реализацией DROP IDENTITY, GENERATED ALWAYS и OVERRIDE, а также опцией INCREMENT BY.

Синтаксис для управления столбцами IDENTITY

```
<определение столбца> ::=
  <имя столбца> <тип данных> GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( <опции
автоинкремента>... ) ] <ограничения столбца>

<опции автоинкремента> ::=
  START WITH <значение> |  INCREMENT [ BY ] <значение>

<alter column definition> ::=
    <название столбца> <генерация автоинкремента> [ <alter identity column option>...
]
  | <название столбца> <alter identity column option>...
  | <название столбца> DROP IDENTITY

<генерация автоинкремента> ::=
  SET GENERATED { ALWAYS | BY DEFAULT }

<alter identity column option> ::=
  RESTART [ WITH <значение> ] | SET INCREMENT [ BY ] <значение>
```

Правила:

- Тип данных столбца идентификации должен быть целым числом с нулевым масштабом. Допустимыми типами являются SMALLINT, INTEGER, BIGINT, NUMERIC(p,0) и DECIMAL(p,0), где 1 <= p <= 18.
- 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.

Опции GENERATED ALWAYS и BY DEFAULT

Предыдущая реализация автоматически генерировала целочисленные ключи, если столбец был опущен в списке столбцов операции вставки. Если столбец не был указан, генератор IDENTITY выдавал значение.

Предложение GENERATED BY является обязательным. Директива GENERATED BY DEFAULT, присутствующая в синтаксисе РЕД Базы Данных 3, реализовала это поведение формально, без альтернативной опции GENERATED ALWAYS:

```
create table objects (
  id integer generated BY DEFAULT as
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    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 order by id;

commit;

ID  NAME
=== =========
1   Table
2   Book
10  Computer
```

Директива `GENERATED ALWAYS` вводит альтернативное поведение, которое принудительно использует генератор идентификаторов независимо от того, предоставляет ли пользователь значение или нет.

Переопределение поведения по умолчанию

Для разовых случаев это поведение можно отменить в `DML`, включив предложение `OVERRIDING SYSTEM VALUE`.

С другой стороны, для разовых случаев, когда вы хотите отменить определенное действие для столбца, заданного с помощью директивы `GENERATED BY DEFAULT`, чтобы он вел себя так, как если бы был определен как `GENERATED ALWAYS` и игнорировал любое значение, предоставленное `DML`, можно воспользоваться предложением `OVERRIDING USER VALUE`.

Изменение поведения по умолчанию

В предложении `ALTER COLUM` оператора `ALTER TABLE` теперь есть синтаксис для изменения поведения `GENERATED` с `BY DEFAULT` на `ALWAYS` или наоборот:

```
alter table objects
  alter id
  SET GENERATED ALWAYS;
```

Предложение DROP IDENTITY

В ситуации, когда вы хотите удалить свойство `IDENTITY` для столбца, но сохранить данные, в операторе `ALTER TABLE` есть `DROP IDENTITY`:

```
alter table objects
  alter id
  DROP IDENTITY;
```

Опция INCREMENT BY в столбцах IDENTITY

По умолчанию столбцы идентичности начинаются с 1 и увеличиваются на 1. Опция INCREMENT BY теперь может использоваться для установки положительного или отрицательного шага увеличения:

```
create table objects (
  id integer generated BY DEFAULT as
    identity (START WITH 10000 INCREMENT BY 10)
    primary key,
  name varchar(15)
);
```

Изменение значения шага увеличения

Для изменения значения шага последовательности, создаваемой генератором IDENTITY, в синтаксисе оператора ALTER TABLE есть SET INCREMENT:

```
alter table objects
  alter id SET INCREMENT BY 5;
```

Изменение значения шага не влияет на существующие данные.

Нет необходимости указывать SET INCREMENT BY 1 для нового столбца или для столбца, который ранее не изменялся, так как шаг по умолчанию равен 1.

Реализация

В RDB$RELATION_FIELDS добавлены два столбца: RDB$GENERATOR_NAME и RDB$IDENTITY_TYPE. RDB$GENERATOR_NAME хранит автоматически созданный генератор для столбца.

В RDB$GENERATORS значение RDB$SYSTEM_FLAG этого генератора будет равно 6. RDB$IDENTITY_TYPE хранит значение 0 для GENERATED ALWAYS, 1 для GENERATED BY DEFAULT и NULL для столбцов, которые не являются IDENTITY.

8.3.6 EXCESS в EXECUTE STATEMENT

Входные параметры команды EXECUTE STATEMENT могут быть дополнены ключевым словом EXCESS. Если указано EXCESS, то данный параметр может быть опущен в тексте запроса.

```
CREATE PROCEDURE P_EXCESS (A_ID INT, A_TRAN INT = NULL, A_CONN INT = NULL)
  RETURNS (ID INT, TRAN INT, CONN INT)
AS
DECLARE S VARCHAR(255);
DECLARE W VARCHAR(255) = '';
BEGIN
  S = 'SELECT * FROM TTT WHERE ID = :ID';

  IF (A_TRAN IS NOT NULL)
  THEN W = W || ' AND TRAN = :a';

  IF (A_CONN IS NOT NULL)
  THEN W = W || ' AND CONN = :b';

  IF (W <> '')
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  THEN S = S || W;

  -- could raise error if TRAN or CONN is null
  -- FOR EXECUTE STATEMENT (:S) (a := :A_TRAN, b := A_CONN, id := A_ID)

  -- OK in all cases
  FOR EXECUTE STATEMENT (:S) (EXCESS a := :A_TRAN, EXCESS b := A_CONN, id := A_ID)
    INTO :ID, :TRAN, :CONN
      DO SUSPEND;
END
```

8.3.7 Управление репликацией

После настройки репликации в конфигурационном файле `replication.conf` ее можно включить-/выключить во время выполнения с помощью оператора `ALTER DATABASE`. Кроме того, набор репликации (т. е. таблицы, которые будут реплицироваться) можно настроить с помощью операторов `ALTER DATABASE` и `CREATE/ALTER TABLE`.

Синтаксис управления репликацией:

```
ALTER DATABASE ... [<управление репликацией>]

CREATE TABLE <имя таблицы> ... [<состояние репликации>]
ALTER TABLE <имя таблицы> ... [<состояние репликации>]

<управление репликацией> ::=
  <состояние репликации> |
  INCLUDE <набор репликации> TO PUBLICATION |
  EXCLUDE <набор репликации> FROM PUBLICATION

<состояние репликации> ::=
  ENABLE PUBLICATION |
  DISABLE PUBLICATION

<набор репликации> ::=
  ALL |
  TABLE <имя таблицы> [, <имя таблицы> ...]
```

Все команды управления репликацией являются `DDL` операторами и поэтому выполняются во время фиксации транзакции.

`ALTER DATABASE ENABLE PUBLICATION` позволяет начать (или продолжить) репликацию со следующей транзакцией, начатой после фиксации текущей транзакции.

`ALTER DATABASE DISABLE PUBLICATION` отключает репликацию сразу после фиксации.

Если используется предложение `INCLUDE ALL TO PUBLICATION`, то все таблицы, созданные после этого, также будут реплицированы, если это не отменено явно оператором `CREATE TABLE`.

Если используется условие `EXCLUDE ALL FROM PUBLICATION`, то все созданные после этого таблицы не будут реплицированы, если это не отменено явно оператором `CREATE TABLE`.

8.4 Поддержка частичных индексов

При создании индекса появилась возможность указать необязательное предложение WHERE, которое определяет условие поиска, ограничивающее подмножество записей таблицы для индексирования. Такие индексы называются частичными индексами. Условие поиска должно содержать один или несколько столбцов таблицы.

Определение частичного индекса также может включать предложение COMPUTED BY.

```
CREATE [UNIQUE] [{ASC[ENDING] | DESC[ENDING]}] INDEX <индекс> ON <таблица>
{ (<список столбцов>) | COMPUTED [BY] ( <выражение> ) }
WHERE <условие поиска>
```

Примеры:

```
-- 1.
CREATE INDEX IT1_COL ON T1 (COL) WHERE COL < 100;
SELECT * FROM T1 WHERE COL < 100;
-- PLAN (T1 INDEX (IT1_COL))

-- 2.
CREATE INDEX IT1_COL2 ON T1 (COL) WHERE COL IS NOT NULL;
SELECT * FROM T1 WHERE COL > 100;
-- PLAN (T1 INDEX IT1_COL2)

-- 3.
CREATE INDEX IT1_COL3 ON T1 (COL) WHERE COL = 1 OR COL = 2;
SELECT * FROM T1 WHERE COL = 2;
-- PLAN (T1 INDEX IT1_COL3)
```

Определение частичного индекса может включать определение UNIQUE. В этом случае каждый ключ в индексе должен быть уникальным. Это позволяет обеспечить уникальность для некоторого подмножества строк таблицы.

Частичные индексы полезны только в следующих случаях:

- Условие WHERE содержит в точности такое же логическое выражение, как и определённое для индекса;
- Условие поиска, заданное для индекса, содержит логические выражения, объединенные по принципу OR, и одно из них явно включено в условие WHERE;
- Условие поиска, определенное для индекса, определяет IS NOT NULL, а условие WHERE включает выражение для того же поля, которое, как известно, игнорирует NULL.

8.5 Комментарии для отображений

Оператор COMMENT ON позволяет добавить комментарий для MAPPING.

```
COMMENT ON [GLOBAL] MAPPING <отображение> IS {<комментарий> | NULL};
```

Глава 9

Язык управления данными (DML)

9.1 JSON

JSON — это текстовый формат данных, который позволяет хранить скалярные значения (число, строку, логическое значение), значения null, массивы и объекты (набор элементов в формате ключ:значение). В качестве элементов массива и значений объектов можно использовать другие массивы и объекты.

JSON хорошо подходит для сериализации. Можно взять несколько значений SQL, например:

```
1, "SQL", true
```

Далее записать их в виде JSON объекта и сохранить его в строку:

```
{"ID":1, "LANG":"SQL", "ACTIVE":true}
```

Получившуюся строку можно поместить в текстовое поле или вывести.

Для получения данных из объекта в JSON SQL используется Язык путей JSON. Там указывается, к какому именно элементу нужно получить доступ. Например, '$.ID' обратится к элементу JSON с именем ID.

9.1.1 Язык путей JSON

Язык путей SQL/JSON чувствителен к регистру как в идентификаторах, так и в ключевых словах и не имеет автоматического преобразования идентификаторов в верхний регистр. Выражение пути является строкой, поэтому его необходимо заключить в одинарные кавычки. Внутри этой строки можно написать строковый литерал, заключив его в двойные кавычки. Апостроф внутри двойных кавычек нужно экранировать. Это можно сделать, используя либо соглашение SQL о написании символа дважды, либо экранирование методом JavaScript

Экранирование методом написания дважды:

```
'$."Name''"'
```

Кавычки в примере интерпретируются следующим образом:

- Внешние одинарные кавычки содержат выражение пути;
- Двойные кавычки содержат символьную строку на языке пути;
- Внутренний апостроф экранируется, поскольку содержится в строковом литерале. Фактически обозначает один символ.

Экранирование по методу JavaScript, то есть \':

```
'$."Name\'"'
```

Экранирование методом записывания символа в unicode, например:

```
'$."Name\u0027"'
```

Экранирование двойных кавычек:

```
'$."\"Name"'
```

9.1.2 Режимы: lax и strict

Механизм пути имеет два режима: `lax` и `strict`.

На данный момент режим `strict` не проверяет соответствие `JSON` указанному `пути JSON`.

Объявление режима:

```
<выражение пути> ::=
  <режим> <путь>

<режим> ::=
  strict
| lax
```

По умолчанию используется режим `lax`.

Режим `lax` преобразует ошибки в пустые последовательности `SQL/JSON`.

Режимы пути регулируют три аспекта: разворачивание массивов, оборачивание элементов в массив и обработку ошибок.

В режиме `lax` массив, содержащий только один элемент, является взаимозаменяемым с этим значением, например, `["hello"]` эквивалентно `"Hello"`. Это подкрепляется следующими правилами:

- Если для операции требуется массив, но операнд не является массивом, то операнд оборачивается в массив;
- Если для операции требуется не массив, но операнд является массивом, то операнд разворачивается в последовательность.

Разворачивание массивов:

- В режиме `lax` – массивы автоматически разворачиваются перед выполнением операции, это означает, что `[*]` можно опустить ;
- В режиме `strict` – массивы не разворачиваются автоматически (в таком случае нужно написать `[*]`, чтобы развернуть массив).

Оборачивание элементов в массив:

- В режиме `lax` – операции пути, применяющиеся к индексу, такие как `$[0]` или `$[*]`, могут быть применены к значению, которое не является массивом, но для этого значение неявно оборачивается в массив перед применением операции;
- В режиме `strict` – автоматическое оборачивание перед выполнением операций, применяющихся к индексу, не выполняется.

Обработка ошибок:

- В режиме `lax` – многие ошибки, связанные с тем, являются ли данные массивом или скаляром, обрабатываются функциями автоматического разворачивания и оборачивания. Остальные ошибки классифицируются либо как структурные, либо как неструктурные. Примером структурной ошибки является `$.name`, если в `$` нет элемента, ключом которого является `name`. Структурные ошибки преобразуются в пустые последовательности `SQL/JSON`. Примером неструктурной ошибки является деление на ноль;
- В режиме `strict` – ошибки строго определены во всех случаях.

9.1.3 Доступ к элементам

Синтаксис обращения к элементам:

```
<выражение пути> ::=
  <режим> <путь>

<путь> ::=
  <доступ к элементу>
| [<метод элемента>]
| [<выражение фильтра>]

<доступ к элементу> ::=
  <контекстная переменная>[<способ доступа>...]

<способ доступа> ::=
  <доступ к элементу объекта>
| <обращение к элементу объекта подстановочным знаком>
| <доступ к элементу массива>
| <обращение к элементу массива подстановочным знаком>
```

Обращение к элементу объекта

Синтаксис обращения к элементу объекта:

```
<доступ к элементу объекта> ::=
  .<ключ>
| .<строка>
| .<*>
```

Есть три способа определения ключа. Первый способ – написание имени ключа открытым текстом. Применяется, если имя ключа не начинается со знака доллара ($) и удовлетворяет правилам идентификатора JavaScript.

Например:

```
$.name
$.firstName
$.Phone
```

Второй способ – написание имени ключа строкой. Это нужно, когда имя ключа начинается со знака доллара ($) или содержит специальные символы.

Например:

```
$."name"
$."$price"
$."home address"
```

Третий способ – обращение к элементу объекта с помощью подстановочного символа.

Например, * запросит все поля объекта:

```
$.*
```

Обращение к элементу массива

```
<доступ к элементу массива> ::=
   [ <список индексов> ]

<список индексов> ::=
   <индекс> [ {, <индекс> }... ]

<индекс> ::=
   <диапазон индексов>
 | <индекс> to <индекс>
 | [ <*> ]

<индекс> ::=
   <число> | last
```

Квадратные скобки содержат список индексов через запятую. Список индексов может быть определён двумя способами: числом или диапазоном между двумя числами, указанным с ключевым словом `to`. Также к элементу массива можно обратиться с помощью подстановочного знака `[*]`. Например, `$[*]` запросит все элементы массива. В режиме `strict` операнд должен быть массивом. В режиме `lax` операнд, не являющийся массивом, будет обёрнут в массив.

Индексы рассчитываются с 0. Таким образом, `[0]` – это первый элемент в массиве.

Для доступа к последнему элементу массива неизвестного размера можно использовать ключевое слово `last` в качестве индекса. Например, `$[last]` обратится к последнему элементу массива, а `$[last-1 to last]` вернёт два последних элемента массива.

Например:

```
$[0, last-1 to last, 5]
```

В данном случае обратятся к первому элементу массива, двум последним и шестому.

В режиме `lax` выражение `$[*]` аналогично `$[0 to last]`. В режиме `strict` между такими обращениями есть различие: `$[0 to last]` требует, чтобы массив содержал хотя бы один элемент, тогда как `$[*]` не вернёт ошибку, если `$` является пустым массивом.

К элементам массива можно обращаться по отрицательным индексам. Например, `$[-1]` обратится к последнему элементу массива, а `$[-2]` к предпоследнему.

Индексы могут быть указаны в любой последовательности и могут повторяться. В режиме `strict` индекс должен быть числовым значением в диапазоне между 0 и `last`. В режиме `lax` индексы, выходящие за границы массива игнорируются. Нечисловые значения индексов в любом режиме будут считаться ошибкой.

9.1.4 Методы элементов

Методы элементов — это методы, которые можно использовать в `пути JSON`. Синтаксис методов элементов:

```
<выражение пути> ::=
   <режим> <путь>

<путь> ::=
   <доступ к элементу>
 | [<метод элемента>]
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
 | [<выражение фильтра>]

<метод элемента> ::=
   .<метод>

<метод> ::=
   type()
   | size()
   | double()
   | ceiling()
   | floor()
   | abs()
   | keyvalue()
```

Методы элементов разворачивают массив в режиме `lax`. Исключение – методы `type()` и `size()`, иначе было бы невозможно узнать их тип или размер.

Метод `type()`

Метод `type()` возвращает строку с именем типа элемента `SQL/JSON`:

- Если элемент – `null`, то вернёт "null";
- Если элемент – `true` или `false`, то вернёт "boolean";
- Если элемент – число, то вернёт "number";
- Если элемент – строка, то вернёт "string";
- Если элемент – массив, то вернёт "array";
- Если элемент – объект, то вернёт "object".

Например, можно вывести только строки:

```
SELECT JSON_QUERY('{"data": [123, "123", "words", false, true, null, [], {}]}', '$.* ?
(@.type() == "string")' RETURNING VARCHAR(100) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

================================================
["123","words"]
```

Или вывести типы элементов:

```
SELECT JSON_QUERY('{"data": [123, "123", "words", false, true, null, [], {}]}', '$.
data[*].type()' RETURNING VARCHAR(100) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

================================================
["number","string","string","boolean","boolean","null","array","object"]
```

Метод `size()`

Метод `size()` возвращает размер элемента `SQL/JSON`:

- Размер массива равен количеству элементов в массиве;
- Размер объекта или скаляра равен 1.

Например, можно вывести массивы, размер которых больше 1:

```
SELECT JSON_QUERY('[ [1, 2, 3], [1], [1, 2]]', '$ ? (@.type() == "array"
&& @.size() > 1)' RETURNING VARCHAR(100) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

====================
[[1,2,3],[1,2]]
```

Можно посчитать количество элементов в массиве:

```
SELECT JSON_QUERY('{"data":[1, 2, 3, 4, 5, 6, 7, 8, 9]}', '$.data.size()'
RETURNING VARCHAR(100) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

====================
[9]
```

Числовые методы элементов: double(), ceiling(), floor(), abs()

Числовые методы элементов выполняют общие числовые функции:

- `double()` преобразует строку в дробное числовое значение;
- `ceiling()`, `floor()` и `abs()` выполняют те же операции, что и `CEILING`, `FLOOR` и `ABS` в `SQL`.

Все эти функции при обработке значения `null` возвращают `null`. Если числовые методы применяются не к числу (например, к объекту), то результатом будет ошибка, но если присутствует оператор сравнения, то результатом будет `Unknown`.

Пример применения метода `double()`:

```
SELECT JSON_VALUE('{"numbers": "555"}', '$.numbers.double()')
FROM RDB$DATABASE;

==================
555
```

Пример применения метода `abs()`:

```
SELECT JSON_VALUE('{"numbers": -555.25}', '$.numbers.abs()')
FROM RDB$DATABASE;

==================
555.25
```

Пример применения метода `ceiling()`:

```
SELECT JSON_VALUE('{"numbers": 555.25}', '$.numbers.ceiling()')
FROM RDB$DATABASE;

==================
556
```

Пример применения метода `floor()`:

```
SELECT JSON_VALUE('{"numbers": 555.25}', '$.numbers.floor()')
FROM RDB$DATABASE;

====================
555
```

Метод keyvalue()

Метод keyvalue() преобразовывает поля JSON-объекта в последовательность объектов, описывающих поля исходного.

```
SELECT JSON_QUERY('{ "who": "Fred", "what": 64 }', '$.keyvalue()' RETURNING
VARCHAR WITH ARRAY WRAPPER ERROR ON ERROR)
FROM RDB$DATABASE;

=======================
[{"name":"who","value":Fred,"id":1},{"name":"what","value":64,"id":1}]
```

В примере выше на вход подается один JSON-объект с двумя полями, а на выходе получается JSON-последовательность из двух объектов с тремя полями у каждого. В результирующей последовательности ключами являются:

- name - имя ключа во входном объекте;
- value - значение ключа;
- id - целое число, являющееся уникальным идентификатором входного JSON-объекта.

В результирующей JSON-последовательности объекты будут в том же порядке, что и поля в исходном объекте, из которых они преобразованы.

В режиме lax вывод keyvalue() разворачивается:

```
SELECT JSON_QUERY(' [ { "who": "Fred", "what": 64 }, { "who": "Fred", "what": 64 } ]
', 'lax $.keyvalue()'
RETURNING VARCHAR WITH ARRAY WRAPPER ERROR ON ERROR)
FROM RDB$DATABASE;

=======================
[ { name: "who", value: "Fred", id: 1 }, { name: "what", value: 64, id: 1 },
{ name: "who", value: "Moe", id: 2 }, { name: "how", value: 22, id: 2 } ]
```

9.1.5 Арифметические операции в пути

Язык путей JSON поддерживает следующие арифметические операции:

- Унарные: + и -;
- Бинарные: +, -, *, /, %.

Модуль, обозначаемый символом %, использует тот же алгоритм, что и функция MOD в SQL.

Унарный плюс и минус

Унарные операции плюс и минус выполняют итерации по последовательности JSON. Каждый элемент последовательности должен быть числом (в противном случае, даже в режиме lax, будет получена ошибка). В остальном единственной ошибкой является переполнение при выполнении унарного минуса некоторых чисел на границах их диапазона.

Унарные операции выполняются после получения значений JSON и выполнения метода. Например:

```
$ = { readings: [15.2, -22.3, 45.9] }
```

Тогда выражение 'lax -$.readings.floor()' эквивалентно lax -($.readings.floor()). В режиме lax унарные операции разворачивают массив.

Таблица 9.1 — Вычисление lax -$.readings.floor()

Шаг	Выражение	Значение
1	$	{ readings: [15.2, -22.3, 45.9]}
2	$.readings	[15.2, -22.3, 45.9]
3	$.readings.floor()	15, -23, 45
4	-$.readings.floor()	-15, 23, -45

Для получения другого порядка вычислений необходимо использовать скобки: lax (-$.readings).floor().

Таблица 9.2 — Вычисление lax (-$.readings).floor()

Шаг	Выражение	Значение
1	$	{ readings: [15.2, -22.3, 45.9] }
2	$.readings	[15.2, -22.3, 45.9]
3	-$.readings	-15.2, 22.3, -45.9
4	(-$.readings)	-15.2, 22.3, -45.9
5	(-$.readings).floor()	-16, 22, -46

В режиме strict эти примеры требуют явного [*] для раскрытия массива:

```
strict -$.readings[*].floor()
```

или

```
strict (-$.readings[*]).floor()
```

Бинарные операции

Бинарные операции не выполняют итерации по последовательности JSON. Они требуют, чтобы их операнд был числом, иначе результатом будет ошибка, даже в режиме lax. Бинарные операторы имеют тот же приоритет, что и в SQL.

Пример сложения:

```
SELECT JSON_QUERY('{"digits": [15.2, -22, 45, 0]}', '$.digits[*]-5.1' RETURNING
VARCHAR(50) WITH ARRAY WRAPPER ERROR ON ERROR) FROM RDB$DATABASE;

JSON_QUERY
==================================================
[10.1,-27.1,39.9,-5.1]
```

Для изменения порядка выполнения нужно использовать скобки:

```
SELECT JSON_VALUE('{"value": 15}', '(-$.value)+2*3-15/5%2' RETURNING VARCHAR(20)
ERROR ON ERROR)
FROM RDB$DATABASE;

==============
-10
```

```
SELECT JSON_VALUE('{"value": 15}', '-($.value+2*3-15/5%2)' RETURNING VARCHAR(20)
ERROR ON ERROR)
FROM RDB$DATABASE;

==============
-20
```

9.1.6 Фильтры

Выражение фильтра является аналогом предложения `WHERE` в `SQL` – оно используется для нахождения элементов, удовлетворяющих определённому условию.

Синтаксис выражения фильтра:

```
<выражение фильтра> ::=
   ? ( <предикат> ) [ <путь> ? ( <предикат> ) ... ]

<предикат> ::=
   <предикат exists>
 | ( <предикат> )
 | <предикат сравнения>
 | <предикат like_regex>
 | <предикат starts with>
 | <предикат is unknown>
```

При использовании фильтра выполняются следующие действия:

- В режиме `lax` массивы в операнде разворачиваются;
- Предикат вычисляется для каждого элемента в последовательности;
- В результат попадают элементы, для которых предикат принял значение `True`.

Переменная `@` в фильтре используется для обозначения текущего элемента в последовательности.

Язык путей содержит следующие предикаты:

- `exists`, чтобы проверить, получает ли выражение пути непустой результат;
- Операторы сравнения `==`, `!=`, `<>`, `<`, `<=`, `>`, и `>=`;
- `like_regex` для сопоставления значения выражения с регулярным выражением;

- `starts with` чтобы проверить, является ли второй операнд началом первого операнда;
- `is unknown`, чтобы проверить, является ли результат сравнения значением `unknown`. Предикат возвращает `true` или `false`.

Выражение пути, указанное после фильтра, будет проверяться, только если выполняется условие предиката в фильтре. Фильтры в выражении пути можно использовать неограниченное количество раз.

Примеры выражения пути без промеждуточного фильтра:

```
SELECT JSON_QUERY('{"root": {"name": "first", "content": "some text"}}', '$.root ?(@.
name=="first")' RETURNING VARCHAR(200) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

========================================
[{"name":"first","content":"some text"}]
```

```
SELECT JSON_QUERY('[{"name": "first", "content": "some text"}, {"name": "second",
"content": "some text"}]', '$ ?(@.content=="some text")' RETURNING VARCHAR(200) WITH
ARRAY WRAPPER ERROR ON ERROR)
FROM RDB$DATABASE;

========================================
[{"name":"first","content":"some text"},{"name":"second","content":"some text"}]
```

Пример применения промежуточного фильтра к объекту JSON:

```
SELECT JSON_QUERY('{"root": {"name": "first", "content": "some text"}}', '$.root ?(@.
name=="first").content' RETURNING VARCHAR(200) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

======================
["some text"]
```

Пример применения промежуточного фильтра к массиву JSON:

```
SELECT JSON_QUERY('[{"name": "first", "content": "some text"}, {"name": "second",
"content": "some text"}]', '$ ?(@.content=="some text").name' RETURNING VARCHAR(200)
WITH ARRAY WRAPPER ERROR ON ERROR)
FROM RDB$DATABASE;

======================
["first","second"]
```

Обработка ошибок в фильтрах

Ошибки могут возникать в выражениях фильтра по двум причинам:

- При вычислении выражений режим `lax` преобразует структурные ошибки в пустую последовательность. Неструктурные ошибки рассматриваются как необработанные ошибки. В режиме `strict` все ошибки являются необработанными;
- Если после вычисления операнда предикат обнаружил, что значение не подходит. Например, `10 == "ten"`, в операндах нет ошибок, но они несопоставимы, поэтому в этом предикате все еще есть ошибка.

При любом типе ошибки предикат возвращает значение `unknown`.

Таблицы истиности

Операнды `&&` (логическое И) будут вычисляться в любом случае.
Результаты `&&` представлены в таблице:

	True	**False**	**Unknown**
`True`	`True`	`False`	`Unknown`
`False`	`False`	`False`	`False`
`Unknown`	`Unknown`	`False`	`Unknown`

Аналогично, операнды `||` (логическое ИЛИ) будут вычисляться в любом случае.
Возможные результаты `||` представлены в таблице:

	True	**False**	**Unknown**
`True`	`True`	`True`	`True`
`False`	`True`	`False`	`Unknown`
`Unknown`	`True`	`Unknown`	`Unknown`

Результаты `!` (логическое отрицание):

P	**NOT P**
`True`	`False`
`False`	`True`
`Unknown`	`Unknown`

Операторы сравнения

Операторы сравнения – это `==, !=, <, <=, >, >=, <>`.
В режиме `lax` операторы сравнения автоматически разворачивают операнды.
В следующей таблице приведены возможные варианты сравнения:

	null	**скаляр**	**массив**	**объект**
null	сравниваемые	сравниваемые	несравниваемые	несравниваемые
скаляр	сравниваемые	сравниваться могут строка со строкой, число с числом, логическое значение с логическим значением	несравниваемые	несравниваемые
массив	несравниваемые	несравниваемые	несравниваемые	несравниваемые
объект	несравниваемые	несравниваемые	несравниваемые	несравниваемые

Сравнение объектов с чем-либо, даже с самими собой, не поддерживается.

Сравнение выполняется с использованием семантики SQL с дополнительным правилом: SQL/JSON null равен SQL/JSON null, не больше и не меньше чего-либо.

Операнды могут быть последовательностями. В таком случае формируется перекрестное вычисление. Каждый элемент в одной последовательности сравнивается с каждым элементом в другой последовательности. Результат unknown, если какая-либо пара элементов несопоставима. Результат true, если любая пара элементов сравнивается и результат сравнения удовлетворяет условию оператора. Во всех остальных случаях результат false. В режиме lax обработчик пути досрочно прекращает вычисление, если обнаруживает ошибочный или успешный результат. В режиме strict обработчик пути должен проверять все сравнения, и если какое-либо из этих сравнений несопоставимо, то результатом будет unknown.

Предикат like_regex

Предикат сравнивает выражение символьного типа с регулярным выражением.

Синтаксис предиката like_regex:

```
<предикат like_regex> ::=
   <предикат> like_regex <регулярное выражение>
      [ FLAGS "<флаг>[ <флаг>...]" ]

<регулярное выражение> ::=
   <строковое значение>

<флаги> ::=
     i
   | m
   | s
   | u
   | t
```

Для регулярных выражений используется синтаксис google re2.

Флаги можно указывать как в верхнем, так и в нижнем регистре. Описание флагов:

- i - чувствительность к регистру (по умолчание не чувствительны);
- m - многострочный режим: ^ и $ соответствуют началу и концу строки (по умолчанию выключено);
- s - символ точки (.) соответствует \n (по умолчанию выключено);
- u - меняет местами значения x* и x*?, x+ и x+? и т.д. (по умолчанию выключено);
- t - убрать пробелы справа.

Например:

```
SELECT JSON_EXISTS('{"name": "Isaac Asimov"}', '$ ? (@.name like_regex "Asimov")')
FROM RDB$DATABASE;

==================
<true>
```

Предикат starts with

Предикат **starts with** проверяет, является ли второй операнд началом первого операнда.

Синтаксис предиката **starts with**:

```
<предикат starts with> ::=
   <текстовое значение> starts with <искомая строка>

<текстовое значение> ::=
   <переменная пути>
 | <переменная passing>
 | <скалярное значение>

<искомая строка> ::=
   <текстовое значение>
```

Например:

```
SELECT JSON_EXISTS('{"name": "Isaac Asimov"}', '$ ? (@.name starts with "Isa")')
FROM RDB$DATABASE;

==================
<true>
```

Предикат exists

Предикат **exists** проверяет, содержит ли результат выражения пути хотя бы один элемент.

Синтаксис предиката **exists**:

```
<предикат exists> ::=
   exists <левая скобка> <путь> <правая скобка>
```

Если результат выражения пути – ошибка, то предикат **exists** вернёт **Unknown**. Если результат выражения пути – пустая последовательность, то предикат **exists** вернёт **False**. В любом другом случае результатом будет **True**.

Например:

```
SELECT JSON_QUERY('{"data": [1, 2, 3]}', '$ ? (exists (@.data))')
FROM RDB$DATABASE;

==================
{"data":[1,2,3]}
```

Предикат is unknown

Предикат **is unknown** проверяет, является ли результат выражения типом **Unknown**.

Синтаксис предиката **is unknown**:

```
<предикат is unknown> ::=
   <правая скобка> <путь> <левая скобка> is unknown
```

Например:

```
SELECT JSON_EXISTS('{"digits": [1, 2, 3, 4, 5]}', '$.digits ? ((@ < 2)
is unknown)')
FROM RDB$DATABASE;

==================
<false>
```

```
SELECT JSON_EXISTS('{"digits": [1, 2, 3, 4, 5]}', '$.digits ? (("hi" > 42) is unknown)
') FROM RDB$DATABASE;

JSON_EXISTS
===========
<true>
```

9.1.7 Функции SQL/JSON

Функции SQL/JSON разделены на три группы: функции генерации данных (JSON_OBJECT, JSON_OBJECTTAG, JSON_ARRAY и JSON_ARRAYAGG), функции работы с данными (JSON_VALUE, JSON_TABLE, JSON_QUERY и JSON_MODIFY) и функции валидации данных (JSON_EXISTS и IS JSON). Функции генерации данных принимают значения SQL и создают объекты или массивы JSON, которые сохраняются либо в строке, либо в текстовом блобе. Функции работы с данными используются для извлечения значений JSON по выражению пути. Из результата убираются пробелы, переносы на новую строку и символы табуляции. Функции валидации проверяют существующий JSON на соответствие указанным критериям.

Общий синтаксис функций работы с данными

Всем функциям работы с данными требуется выражение пути, значение JSON, а также необязательные значения параметров.

Функции работы с данными используют общий синтаксис:

```
<общий синтаксис> ::=
   <значение JSON>, <выражение пути> [<оператор передачи контекстных переменных>] [
<выходное значение>]

<значение JSON> ::=
   <значение> [FORMAT <формат>]

<формат> ::=
   AUTO
 | SQL
 | JSON

<выражение пути> ::=
   <режим> <путь>

<путь> ::=
   <доступ к элементу>
  |[<метод элемента>]
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  |[<выражение фильтра>]

<оператор передачи контекстных переменных> ::=
   PASSING <параметр JSON> [ {, <параметр JSON> } ]

<параметр JSON> ::=
   <значение и формат> AS <идентификатор>

<выходное значение> ::=
   RETURNING <тип данных>
```

Значение JSON

`Значение JSON` может быть любым типом данных.

```
<значение JSON> ::=
   <значение> [FORMAT <формат>]

<формат> ::=
   AUTO
 | SQL
 | JSON
```

Формат может принимать три значения: `AUTO`, `SQL` и `JSON`. По умолчанию используется `FORMAT AUTO`.

`FORMAT AUTO` – это формат, при котором все значения, созданные функциями генерации данных (`JSON_OBJECT`, `JSON_OBJECTAGG`, `JSON_ARRAY`, `JSON_ARRAYAGG`) или являющиеся результатом работы функции `JSON_QUERY`, будут обрабатываться как `JSON`. Все остальные значения будут преобразованы в `JSON скаляр`. Двойные кавычки и слеши будут экранированы.

`FORMAT SQL` – формат, при котором указанное значение будет преобразовано в `JSON скаляр`, даже если фактически значение является `JSON`.

`FORMAT JSON` – формат, при котором указанное значение будет обрабатываться как JSON, даже если фактически значение является строкой.

Выражение пути

`Выражение пути` определяет, какие значения из исходных данных нужно обработать.

```
<выражение пути> ::=
   <режим> <путь>

<путь> ::=
   <доступ к элементу>
  |[<метод элемента>]
  |[<выражение фильтра>]
```

Например, можно использовать значения, которые больше 4:

```
SELECT JSON_QUERY('[{"value":4}, {"value":6}, {"value":42}]', 'lax $.value ? (@ > 4)'
WITH ARRAY WRAPPER)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
FROM RDB$DATABASE;

==================
[6,42]
```

Оператор передачи контекстных переменных

Оператор PASSING используется для передачи параметров в выражение пути. Синтаксически оператор PASSING представляет собой список значений с псевдонимом для каждого:

```
<параметр JSON> ::=
   <значение и формат> AS <идентификатор>
```

Идентификатор указывает имя переменной, с помощью которого на значение можно ссылаться в выражении пути.

```
SELECT JSON_QUERY('[{"value":4}, {"value":6}, {"value":42}]', 'lax $.value ? (@ > $TR)
' PASSING 5 AS TR RETURNING VARCHAR(100) WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

==================
[6,42]
```

В примере число 5 передаётся с помощью переменной TR.

Выходное значение

Синтаксис предназначен для указания типа данных, возвращаемого функцией JSON:

```
<выходное значение> ::=
   RETURNING <тип данных>
```

Если возвращаемый тип данных не задан, то по умолчанию используется тип входного значения:

- если входное значение типа BLOB, то выходное значение будет BLOB TEXT в кодировке UTF8;
- если входное значение текстового типа, то выходным значением будет VARCHAR размером длина входной строки + 2 в кодировке UTF8;
- если входное текстовое значение больше максимального размера VARCHAR, то для выходного значения будет использован тип BLOB;
- если входной значение не является текстовым типом, то выходным значением будет VARCHAR достаточного размера. Если не получится вычислить достаточный размер строки, то будет использован тип BLOB;
- для функций генерации данных по умолчанию используеься VARCHAR(6000) в кодировке UTF8.

9.1.8 Функции работы с данными

Функции работы с данными:

- JSON_VALUE – извлекает скалярное значение;
- JSON_QUERY – извлекает объекты JSON и массивы JSON;
- JSON_TABLE – извлекает реляционные данные из данных JSON;

- JSON_MODIFY – изменяет существующие значения или добавляет новые.

JSON_VALUE

JSON_VALUE – это функция для извлечения скалярного значения из JSON.

Синтаксис оператора:

```
<функция JSON_VALUE> ::=
   JSON_VALUE <левая скобка>
      <Общий синтаксис>
      [ <выходное значение> ]
      [ <поведение при пустом значении> ON EMPTY ]
      [ <поведение при ошибке> ON ERROR ]
   <правая скобка>

<выходное значение> ::=
   RETURNING <тип данных>

<поведение при пустом значении> ::=
   ERROR
 | NULL
 | DEFAULT <пользовательское значение>

<поведение при ошибке> ::=
   ERROR
 | NULL
 | DEFAULT <пользовательское значение>
```

Функция JSON_VALUE может извлекать только одно скалярное значение из JSON. Если есть разворачивание и в массиве один элемент, то функция вернёт его без ошибки:

```
SELECT JSON_VALUE('"numbers": [555.25]', '$.numbers.abs()')
FROM RDB$DATABASE;

====================
555.25
```

Поведение при пустом значении определяет поведение, если результат выражения пути пустой:

- NULL ON EMPTY означает, что JSON_VALUE вернёт значение null;
- ERROR ON EMPTY означает, что будет показана ошибка;
- DEFAULT <пользовательское значение> ON EMPTY означает, что JSON_VALUE вернёт указанное значение.

Для ON EMPTY и ON ERROR по умолчанию используется NULL.

JSON_QUERY

JSON_QUERY – это функция для извлечения массива JSON или объекта JSON.

Синтаксис JSON_QUERY:

```
<Функция JSON_QUERY> ::=
   JSON_QUERY <левая скобка>
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
      <общий синтаксис JSON API>
      [ <выходное значение> ]
      [ <разворачивание> WRAPPER ]
      [ <отображение кавычек> QUOTES [ ON SCALAR STRING ] ]
      [ <поведение при пустом значении> ON EMPTY ]
      [ <поведение при ошибке> ON ERROR ]
   <правая скобка>

<оборачивание> ::=
   WITHOUT [ ARRAY ]
 | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]

<отображение кавычек> ::=
   KEEP
 | OMIT

<поведение при пустом значении> ::=
   ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT

<поведение при ошибке> ::=
   ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT
```

Предложения `ON EMPTY` и `ON ERROR` те же, что в `JSON_VALUE`. Опция `DEFAULT` отсутствует, но можно указать `EMPTY ARRAY` или `EMPTY OBJECT` в качестве результата.

Поведение при пустом значении по умолчанию установлено `NULL ON EMPTY`. Поведение при ошибке по умолчанию `NULL ON ERROR`. Для разворачивания значением по умолчанию является `WITHOUT ARRAY`. Если входное значение является значением `null`, то результатом `JSON_QUERY` будет `null`.

При использовании `WITH` с опцией `UNCONDITIONAL` результат всегда будет заключаться в квадратные скобки. При использовании `WITH` с опцией `CONDITIONAL` результат будет заключаться в квадратные скобки, только если возвращаемое функцией значение не является массивом. Для `WITH` значением по умолчанию является `UNCONDITIONAL`.

Если в запросе содержится выражение фильтра (знак вопроса) или числовые методы элементов (`double()`, `ceiling()`, `floor()`, `abs()`), то массив автоматически разворачивается. С методами `size()` и `type()` массив автоматически разворачиваться не будет.

Если на выходе у `JSON_QUERY` получается последовательность элментов (например, после развращивания массива), то необходимо добавить предикат `WITH ARRAY WRAPPER`, так как элементы уже являются отдельными значениями. Также `WRAPPER` требуется в любом запросе с квадратными скобками, иначе будет ошибка.

```
SELECT JSON_QUERY('{"numbers": ["555", "345.567", "0.12355"]}', '$.numbers[*].double()
' WITH ARRAY WRAPPER)
FROM RDB$DATABASE;

========================
[555,345.567,0.12355]
```

JSON_MODIFY

JSON_MODIFY – функция позволяет изменить существующий JSON: вставить новые поля в объект, добавить элементы в массив, изменить поля/элементы массива или удалить их.

```
<функция JSON_MODIFY> ::=
   JSON_MODIFY(<путь JSON>, <JSON-текст>, {DELETE | [<режим>] <MODIFY_VALUE>}
[RETURNING <TYPE>] [<поведение при ошибке>] [<поведение при пустом значении>])

<режим> =
   UPDATE
 | APPEND
 | INSERT

<поведение при ошибке> =
   NULL ON ERROR
 | ERROR ON ERROR
 | EMPTY ARRAY ON ERROR
 | EMPTY OBJECT ON ERROR
 | DEFAULT value on ERROR

<поведение при пустом значении> =
   WORK ON EMPTY
 | ERROR ON EMPTY
 | NULL ON EMPTY
 | EMPTY ARRAY ON EMPTY
 | EMPTY OBJECT ON EMPTY
 | DEFAULT value on EMPTY
```

Есть несколько режимов работы функции:

- UPDATE - используется для замены значения;
- APPEND - используется для добавления нового элемента в конец массива;
- DELETE - используется для удаления элемента;
- INSERT - для вставки элемента на определённую позицию в массиве.

Режимом работы по умолчанию является UPDATE. При использовании его в режиме lax и указанном WORK ON EMPTY в случае, если по указанному пути значение не найдено, то оно будет добавлено.

```
SELECT JSON_MODIFY('{"data": "test"}', '$.id', UPDATE 5 RETURNING VARCHAR(30))
from RDB$DATABASE;

JSON_MODIFY
======================
{"data":"test","id":5}
```

При использовании режима DELETE нужно опустить MODIFY_VALUE и запятую, то есть передавать только два аргумента.

При использовании INSERT значение будет вставлено именно на указанный индекс, а остальные значения сдвинутся. Например:

```
SELECT JSON_MODIFY('[1,2,3,4]', '$[0]', INSERT 'HI' returning varchar(30)) from RDB
$DATABASE;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
==================
["HI",1,2,3,4]
```

При использовании DELETE значение будет удалено:

```
SELECT JSON_MODIFY('{"data": "test", "id":14}', '$.id', delete) from RDB$DATABASE;

JSON_MODIFY
============================
{"data":"test"}
```

Поведение при ошибке определяет поведение, если при обработке выражения пути возникла ошибка:

- NULL ON ERROR означает, что при возникновении ошибки JSON_MODIFY вернёт значение null;
- ERROR ON ERROR означает, что при возникновении ошибки будет показана ошибка;
- EMPTY ARRAY ON ERROR означает, что при возникновении ошибки JSON_MODIFY вернёт пустой массив;
- EMPTY OBJECT ON ERROR означает, что при возникновении ошибки JSON_MODIFY вернёт пустой объект;
- DEFAULT <пользовательское значение> ON ERROR означает, что епри возникновении ошибки JSON_MODIFY вернёт указанное значение.

По умолчанию используется NULL ON ERROR.

Поведение при пустом значении определяет поведение, если результат выражения пути пустой:

- WORK ON EMPTY означает, что JSON_MODIFY вернёт отформатированный JSON;
- ERROR ON EMPTY означает, что будет показана ошибка;
- NULL ON EMPTY означает, что JSON_MODIFY вернёт значение null;
- ERROR ON EMPTY означает, что будет показана ошибка;
- EMPTY ARRAY ON EMPTY означает, что JSON_MODIFY вернёт пустой массив;
- DEFAULT <пользовательское значение> ON EMPTY означает, что JSON_MODIFY вернёт указанное значение.

По умолчанию используется WORK ON EMPTY.

JSON_TABLE

JSON_TABLE - это функция, которая принимает JSON в качестве входных данных и извлекает из них реляционные данные. Функция имеет три параметра:

- Значение JSON;
- Выражение пути SQL/JSON;
- Предложение COLUMNS для определения формы выходной таблицы.

Синтаксис предложения JSON_TABLE:

```
<JSON_TABLE> ::=
   JSON_TABLE <левая скобка>
      <общий синтаксис JSON API>
      <предложение COLUMNS>
      [<предложение PLAN>]
      [<поведение при ошибке JSON_TABLE> ON ERROR]
   <правая скобка>
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<предложение COLUMNS> ::=
   COLUMNS <левая скобка>
      <определение столбца>
      [ { <запятая> <определение столбца> }... ]
   <правая скобка>

<определение столбца> ::=
   <порядковый номер строки>
 | <определение стандартного столбца>
 | <определение форматированного столбца>
 | <вложенные столбцы>

<порядковый номер строки> ::=
   <название столбца> FOR ORDINALITY

<определение стандартного столбца> ::=
   <название столбца> <тип данных>
      [ PATH <путь к столбцу> ]
      [ <поведение при пустом значении столбца> ON EMPTY ]
      [ <поведение при ошибке> ON ERROR ]

<поведение при пустом значении столбца> ::=
   ERROR
 | NULL
 | DEFAULT <пользовательское значение>

<поведение при ошибке> ::=
   ERROR
 | NULL
 | DEFAULT <пользовательское значение>

<определение форматированного столбца> ::=
   <название столбца> <тип данных>
   FORMAT <представление JSON>
      [ PATH <путь к столбцу> ]
   [ <разворачивание> WRAPPER ]
   [ <отображение кавычек> QUOTES
      [ ON SCALAR STRING ] ]
   [ <поведение при пустом значении форматированного столбца> ON EMPTY ]
   [ <поведение при ошибке форматированного столбца> ON ERROR ]

<разворачивание> ::=
   WITHOUT [ ARRAY ]
 | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]

<отображение кавычек> ::=
   KEEP
 | OMIT

<поведение при пустом значении форматированного столбца> ::=
   ERROR
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<поведение при ошибке форматированного столбца> ::=
   ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<поведение при ошибке JSON_TABLE> ::=
   ERROR
  | EMPTY
```

С помощью предложения `COLUMNS` можно устанавливать четыре типа столбцов: `столбцы с порядковым номером`, `стандартные`, `форматированные` и `вложенные`.

В столбцах с порядковым номером нумерация строк начинается с 1.

Стандартный столбец создаёт столбцы скалярного типа. Он создается с использованием семантики `JSON_VALUE` и использует выражение `JSON` пути. Столбец также содержит необязательные предложения `ON EMPTY` и `ON ERROR` с теми же вариантами выбора и семантикой, что и `JSON_VALUE`.

Форматированный столбец создаёт столбцы скалярного типа. Он создается с использованием семантики `JSON_QUERY`. Столбец также содержит необязательные предложения `WRAPPER`, `QUOTES`, `ON EMPTY` и `ON ERROR` с теми же вариантами выбора и семантикой, что и `JSON_QUERY`.

Пример работы `JSON_TABLE`:

Таблица 9.7 — BOOKCLUB

ID	JCOL

111	{
	"Name": "John Smith",
	"address": {
	"streetAddress": "21 2nd Street",
	"city": "New York",
	"state": "NY",
	"postalCode": 10021
	},
	"phoneNumber": [
	{
	"type": "home",
	"number": "212 555-1234"
	},
	{
	"type": "fax",
	"number": "646 555-4567"
	}
	],
	"books": [
	{
	"title": "The Talisman",
	"authorList": [
	"Stephen King",
	"Peter Straub"
	],
	"category": [
	"SciFi",
	"Novel"
	]
	},
	{
	"title": "Far from the Madding Crowd",
	"authorList": [
	"Thomas Hardy"
	],
	"category": [
	"Novel"
	]
	}
	]
	}

(разрыв таблицы)

(разрыв таблицы)

ID	JCOL
222	{ "Name": "Peter Walker", "address": { "streetAddress": "111 Main Street", "city": "San Jose", "state": "CA", "postalCode": 95111 }, "phoneNumber": [{ "type": "home", "number": "408 555-9876" }, { "type": "office", "number": "650 555-2468" }], "books": [{ "title": "Good Omens", "authorList": ["Neil Gaiman", "Terry Pratchett"], "category": ["Fantasy", "Novel"] }, { "title": "Smoke and Mirrors", "authorList": ["Neil Gaiman"], "category": ["Novel"] }] }
333	{ "Name" : "James Lee" }

```
SELECT jt.postal FROM bookclub, JSON_TABLE ( bookclub.jcol, '$' COLUMNS (postal INT
PATH '$.address.postalCode' DEFAULT '0' ON EMPTY DEFAULT '1' ON ERROR) ) AS jt;

POSTAL
============
10021
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
95111
0
```

Вложенные столбцы

Последний вариант определения столбца - это вложенные столбцы. Синтаксис представлен ниже:

```
<вложенные столбцы> ::=
   NESTED [ PATH ] <путь к вложенной таблице>
      [ AS <имя вложенного пути> ]
      <предложение COLUMNS>

<путь к вложенной таблице>::=
   <путь>

<имя вложенного пути> ::=
   <идентификатор>
```

Предложение NESTED позволяет объединить вложение объектов или массивов JSON в одном запросе, а не связывать несколько выражений JSON_TABLE с помощью SQL. За ключевым словом NESTED следует путь, для которого опционально можно указать псевдоним. Путь обеспечивает уточненный контекст для вложенных столбцов. Имя пути в основном используется, чтобы создать явный план.

Пример работы с вложенными столбцами:

BOOKCLUB приведена *выше*.

```
SELECT jt.Name, jt.phone FROM bookclub,
JSON_TABLE ( bookclub.jcol, 'lax $'
COLUMNS (Name varchar(50) path '$.Name',
        NESTED PATH '$.phoneNumber[*].number' COLUMNS (
            phone CHAR(30) PATH '$' NULL ON EMPTY))
) AS jt;

NAME           PHONE
============== ==============
John Smith     212 555-1234
John Smith     646 555-4567
Peter Walker   408 555-9876
Peter Walker   650 555-2468
James Lee      <null>
```

Предложение PLAN

Для каждого JSON-пути можно указать его псевдоним с попомщью предложения AS. Имена путей являются идентификаторами и должны быть уникальными. Они используются в предложении PLAN для определения плана вывода.

Синтаксис предложения PLAN:

```
<предложение PLAN> ::=
   <определение плана>
 | <план по умолчанию>
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<определение плана> ::=
   PLAN (<план>)

<план> ::=
   <имя пути>
 | <родительский/дочерний>
 | <пара>

<имя пути> ::=
   <идентификатор>

<родительский/дочерний> ::=
   <внешний план>
 | <внутренний план>

<внешний план> ::=
   <путь> OUTER <первичный план>

<внутренний план> ::=
   <путь> INNER <первичный план>

<пара> ::=
   <план объединения>
 | <перекрестный план>

<план объединения> ::=
   <первичный план> UNION <первичный план>
      [ { UNION <первичный план> }... ]

<перекрестный план> ::=
   <первичный план> CROSS <первичный план>
      [ { CROSS <первичный план> }... ]

<первичный план> ::=
   <имя пути>
 | ( <план> )

<план по умолчанию> ::=
   PLAN DEFAULT ( <определение плана по умолчанию> )

<определение плана по умолчанию> ::=
   <внутренний/внешний план>
      [ <план объединения/перекрестный> ]
 | <план объединения/перекрестный>
      [ , <внутренний/внешний план> ]

<внутренний/внешний план> ::=
   INNER
 | OUTER

<план объединения/перекрестный> ::=
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    UNION
  | CROSS
```

Ключевые слова INNER, OUTER, UNION и CROSS в контексте предложения PLAN имеют следующие характеристики:

- INNER соответствует семантике INNER JOIN.
- OUTER соответствует семантике LEFT OUTER JOIN и используется по умолчанию для отношений родительский/дочерний.
- Первым операндом INNER и OUTER является путь, который должен быть предком всех имен путей во втором операнде.
- Если план явный, то все пути должны быть явными и встречаться в предложении PLAN ровно один раз.
- CROSS соответствует семантике CROSS JOIN.
- UNION соответствует семантике FULL OUTER JOIN неудовлетворительным предикатом, таким как 1=0, и по умолчанию используется для отношений между сиблингами.
- UNION является ассоциативным.
- CROSS является ассоциативным.
- Круглые скобки необходимы для разграничения сложных выражений. Не существует приоритета между UNION и CROSS.

Примеры работы с предложением PLAN:

BOOKCLUB приведена *выше*.

Пример работы с планом DEFAULT:

```
SELECT bookclub.id, jt.name, jt.title, jt.author, jt.category
FROM bookclub,
JSON_TABLE ( bookclub.jcol, 'lax $'
      COLUMNS (
               name VARCHAR(30) PATH 'lax $.Name',
               NESTED PATH 'lax $.books[*]'
                        COLUMNS (
                                 title VARCHAR(60) PATH 'lax $.title',
                                 NESTED PATH 'lax $.authorList[*]' AS ATH
                                          COLUMNS ( author VARCHAR(30) PATH 'lax $'),
                                 NESTED PATH 'lax $.category[*]' AS CAT
                                          COLUMNS ( category VARCHAR(30) PATH 'lax $')
                                )
              )
PLAN DEFAULT(INNER,CROSS)
)AS jt;

ID  NAME         TITLE              AUTHOR          CATEGORY
=== ============ ================  =============== =========
111 John Smith   The Talisman      Stephen King    SciFi
111 John Smith   The Talisman      Stephen King    Novel
111 John Smith   The Talisman      Peter Straub    SciFi
111 John Smith   The Talisman      Peter Straub    Novel
111 John Smith   Far From the      Thomas Hardy    Novel
                 Madding Crowd
222 Peter Walker Good Omens        Neil Gaiman     Fantasy
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
222 Peter Walker Good Omens        Neil Gaiman     Novel
222 Peter Walker Good Omens        Terry Pratchett Fantasy
222 Peter Walker Good Omens        Terry Pratchett Novel
222 Peter Walker Smoke and Mirrors Neil Gaiman     Novel
```

Пример работы внутреннего плана с объединением:

```
SELECT  JT.t, JT.a
FROM JSON_TABLE( '[{"t":[1,2], "a":[10,20]}]', '$[*]' as PERSON
COLUMNS (
    NESTED PATH 'lax $.t[*]' as SUB1 columns(t VARCHAR(30) PATH 'lax $'),
    NESTED PATH 'lax $.a[*]' as SUB2 columns(a VARCHAR(30) PATH 'lax $')
)
PLAN (PERSON INNER (SUB1 UNION SUB2) )
) AS JT;

T        A
======== =========
1        <null>
2        <null>
<null>   10
<null>   20
```

Пример работы внутреннего перекрёстного плана:

```
SELECT  JT.t, JT.a
FROM JSON_TABLE( '[{"t":[1,2], "a":[10,20]}]', '$[*]' as PERSON
COLUMNS (
    NESTED PATH 'lax $.t[*]' as SUB1 columns(t VARCHAR(30) PATH 'lax $'),
    NESTED PATH 'lax $.a[*]' as SUB2 columns(a VARCHAR(30) PATH 'lax $')
)
PLAN (PERSON INNER (SUB1 CROSS SUB2) )
) AS JT;

T  A
== ====
1  10
1  20
2  10
2  20
```

Пример работы внешнего плана:

```
SELECT  JT.t, JT.a
FROM JSON_TABLE( '{"t":null, "a":[10,20]}', '$' as PERSON
COLUMNS (
    T VARCHAR(30) PATH 'lax $.t',
    NESTED PATH 'lax $.a[*]' as SUB2 columns(a VARCHAR(30) PATH 'lax $')
)
PLAN (PERSON OUTER SUB2)
) AS JT;

T       A
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
======= ===
<null>  10
<null>  20
```

9.1.9 Функции валидации

Функции валидации JSON:

- JSON_EXISTS – определяет, существует ли значение по заданному пути;
- IS JSON – определяет, является ли указанное строковое значение текстом JSON.

JSON_EXISTS

Функция JSON_EXISTS необходима для проверки существования какого-либо значения по заданному пути. Возвращает логический результат. True, если выражение пути находит один или несколько элементов SQL/JSON.

Синтаксис оператора:

```
<функция JSON_EXISTS> ::=
   JSON_EXISTS <левая скобка>
      <Общий синтаксис JSON API>
      [ <поведение при ошибке> ON ERROR]
   <правая скобка>

<поведение при ошибке> ::=
   TRUE | FALSE | UNKNOWN | ERROR
```

Пример работы функуии:

```
SELECT JSON_EXISTS('{"tags":{"test":[1,2,3,4,5]}}', '$.tags.test[2]') FROM RDB
$DATABASE;

JSON_EXISTS
===========
<true>
```

Предложение ON ERROR по умолчанию равно FALSE. Если входное значение является значением null, то результатом JSON_EXISTS будет Unknown.

IS JSON

Предикат IS JSON проверяет значение на соответствие схеме JSON.

Синтаксис оператора следующий:

```
<предикат IS JSON> ::=
   <значение> [FORMAT <формат>] IS JSON
      [ { | [SYSTEM] FORMAT}]
      [ <ограничение уникальности ключа> ]

<тип значения> ::=
   VALUE
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  | ARRAY
  | OBJECT
  | SCALAR

<ограничение уникальности ключа> ::=
    WITH UNIQUE [ KEYS ]
  | WITHOUT UNIQUE [ KEYS ]
```

Например, следующий запрос проверяет, является ли указанное значение JSON-текстом:

```
SELECT '[{"value":5}, 10, true]' IS JSON
FROM RDB$DATABASE;

====================
<true>
```

Скалярные значения также являются JSON-текстом:

```
SELECT '"String scalar value"' IS JSON
FROM RDB$DATABASE;

====================
<true>
```

Тип значения позволят проверить, соответствует ли входная строка указанному типу:

- VALUE – для проверки, является ли входная строка любым значением JSON, в том числе null:

  ```
  SELECT 'null' IS JSON VALUE
  FROM RDB$DATABASE;

  ====================
  <true>
  ```

  При указании SQL значения null результатом будет null:

  ```
  SELECT null IS JSON VALUE
  FROM RDB$DATABASE;

  ==================
  <null>
  ```

- ARRAY – для проверки, является ли входная строка массивом:

  ```
  SELECT '[1,2,3]' IS JSON ARRAY
  FROM RDB$DATABASE;

  ==================
  <true>
  ```

- OBJECT – для проверки, является ли входная строка объектом;

  ```
  SELECT '{"value":5}' IS JSON OBJECT
  ```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
FROM RDB$DATABASE;

====================
<true>
```

- SCALAR – для проверки, является ли входная строка скаляром.

```
SELECT '1' IS JSON SCALAR
FROM RDB$DATABASE;

====================
<true>
```

SYSTEM FORMAT проверяет, является ли входная строка результатом работы JSON-функции:

```
SELECT JSON_QUERY('[]', '$') IS JSON SYSTEM FORMAT
FROM RDB$DATABASE;

====================
<true>
```

Типом входных данных по умолчанию является FORMAT JSON.

Ограничение уникальности ключа проверяет, есть ли во входной строке повторяющиеся имена полей у объектов. Если есть, то при использовании WITHOUT UNIQUE KEYS функция IS JSON вернёт true, при использовании WITH UNIQUE KEYS – false. По умолчанию используется WITHOUT UNIQUE KEYS.

Работа фуннкции IS JSON с WITHOUT UNIQUE KEYS:

```
SELECT '{"A":1, "B":2, "A":3}' IS JSON
FROM RDB$DATABASE;

====================
<true>
```

Работа фуннкции IS JSON с WITH UNIQUE KEYS:

```
SELECT '{"A":1, "B":2, "A":3}' IS JSON WITH UNIQUE
FROM RDB$DATABASE;

====================
<false>
```

9.1.10 Функции генерации данных

Функции генерации данных:

- JSON_OBJECT – создаёт объект JSON из явных пар ключ-значение;
- JSON_OBJECTAGG – создаёт объект JSON путем агрегирования полей из таблицы или другого селективного источника;
- JSON_ARRAY – создаёт массив JSON из явного списка данных или подзапроса;
- JSON_ARRAYAGG – создаёт массив JSON путём путём агрегирования данных SQL.

Общий формат входных значений

Функции генерации данных используют общий формат для входных значений:

```
<значение и формат> ::=
   <значение> FORMAT <формат>
```

`Значение` может быть любым типом данных.

Формат может принимать три значения: `AUTO`, `SQL` и `JSON`. По умолчанию используется `FORMAT AUTO`.

`FORMAT AUTO` – это формат, при котором все значения, созданные функциями генерации данных (`JSON_OBJECT`, `JSON_OBJECTAGG`, `JSON_ARRAY`, `JSON_ARRAYAGG`) или являющиеся результатом работы функции `JSON_QUERY`, будут обрабатываться как `JSON`. Все остальные значения будут преобразованы в `JSON скаляр`. Двойные кавычки и слеши будут экранированы.

Например:

```
SELECT JSON_ARRAY('1', '2' FORMAT SQL, '3' FORMAT JSON)
from RDB$DATABASE;

============
["1","2",3]
```

`FORMAT SQL` – формат, при котором указанное значение будет преобразовано в `JSON скаляр`, даже если фактически значение является `JSON`.

Например:

```
SELECT JSON_ARRAY(JSON_QUERY('{}', '$'), JSON_QUERY('{}', '$') FORMAT JSON,
JSON_QUERY('{}', '$') FORMAT SQL)
from RDB$DATABASE;

============
[{},{},"{}"]
```

`FORMAT JSON` – формат, при котором указанное значение будет обрабатываться как JSON, даже если фактически значение является строкой.

Например:

```
SELECT JSON_ARRAY(1, '2', '3' FORMAT JSON)
from RDB$DATABASE;

============
[1,"2",3]
```

JSON_OBJECT

Функция `JSON_OBJECT` создаёт объекты `JSON` из явных пар ключ-значение.

Синтаксис функции:

```
<конструктор объекта JSON> ::=
   JSON_OBJECT <левая скобка>
      [ <ключ и значение> [ { <запятая> <ключ и значение> }... ]
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
      [ <поведение при значении null> ]
      [ <ограничение уникальности ключа> ] ]
      [ <выходное значение>]
   <правая скобка>

<ключ и значение> ::=
   [ KEY ] <ключ> VALUE <значение и формат>
 | <ключ> <двоеточие> <значение и формат>

<ключ JSON> ::=
   <строковое значение>

<поведение при значении null> ::=
   NULL ON NULL
 | ABSENT ON NULL

<ограничение уникальности ключа> ::=
   WITH UNIQUE [ KEYS ]
 | WITHOUT UNIQUE [ KEYS ]
```

`Ключ JSON` может быть только строковым типом и не может быть равным `NULL`. Если `значение JSON` равно `NULL`, то поведение определяется предложением `<поведение при значении null>`. `NULL ON NULL` возврвщает значение `NULL`, а `ABSENT ON NULL` опускает такую пару ключ-значение из результирующего объекта. По умолчанию используется `NULL ON NULL`.

Построение объекта `JSON` с `NULL ON NULL`:

```
SELECT JSON_OBJECT('size': 3, key 'name' value null, 'ref': false NULL ON NULL)
FROM RDB$DATABASE;

========================================
{"size":3,"name":null,"ref":false}
```

Построение объекта `JSON` с `ABSENT ON NULL`:

```
SELECT JSON_OBJECT('size': 3, key ''name' value null, 'ref': false ABSENT ON NULL)
FROM RDB$DATABASE;

==========================
{"size":3,"ref":false}
```

Ограничение уникальности ключа определяет, должны ли ключи повторяться. При использовании `WITHOUT UNIQUE KEYS` ключи могут дублироваться. При использовании `WITH UNIQUE KEYS` ключи должны быть уникальными, иначе результатом работы функции будет ошибка. По умолчанию используется `WITHOUT UNIQUE KEYS`.

Работа функции `JSON_OBJECT` с `WITHOUT UNIQUE KEYS`:

```
SELECT JSON_OBJECT('A':1, 'B':2, 'A':3)
FROM RDB$DATABASE;

======================
{"A":1,"B":2,"A":3}
```

Работа функции JSON_OBJECT с WITH UNIQUE KEYS:

```
SELECT JSON_OBJECT('A':1, 'B':2, 'A':3 WITH UNIQUE)
FROM RDB$DATABASE;

=======================================
There is a non-unique key with name 'A' on rows 0 and 2
```

JSON_OBJECTAGG

Можно построить объект JSON путем агрегирования информации из таблицы SQL. Подразумевается, что таблица фактически содержит столбец с ключами JSON и другой столбец с соответствующими ключам значениями.

Синтаксис функции:

```
<агрегатный конструктор объектов JSON> ::=
   JSON_OBJECTAGG <левая скобка>
      <ключ и значение>
      [ <поведение при значении null> ]
      [ <ограничение уникальности ключа> ]
      [ <выходное значение> ]
   <правая скобка>

<поведение при значении null> ::=
   NULL ON NULL
 | ABSENT ON NULL

<ограничение уникальности ключа> ::=
   WITH UNIQUE [ KEYS ]
 | WITHOUT UNIQUE [ KEYS ]
```

Таблица 9.8 — Таблица OPTIONS

ID	OPTION
1	"SIZE"
2	"NAME"
3	"VALUE"

Пример построения объекта:

```
SELECT JSON_OBJECTAGG(option : id)
FROM OPTIONS;

=======================================
{"size":1,"name":2,"value":3}
```

Поведение при значении null по умолчанию равно NULL ON NULL.

JSON_ARRAY

Функция JSON_ARRAY позволяет построить массив JSON из явного списка данных.

Синтаксис функции:

```
<конструктор массивов JSON> ::=
   <создание массива JSON из списка значений>
 | <создание массива JSON из результата запроса>

<создание массива JSON из списка значений> ::=
   JSON_ARRAY <левая скобка>
      [ <значение и формат> [ { <запятая> <значение и формат> }... ]
      [ <поведение при значении null> ] ]
      [ <выходное значение> ]
   <правая скобка>

<создание массива JSON из результата запроса> ::=
   JSON_ARRAY <левая скобка>
   <SQL-запрос>
      [ <формат> ]
      [ <поведение при значении null> ]
      [ <выходное значение> ]
   <правая скобка>
```

JSON_ARRAY предоставляет два варианта создания массива. Первый вариант создает результат из явного списка значений SQL:

```
SELECT JSON_ARRAY(1,2,3,4,5)
FROM RDB$DATABASE;

====================
[1,2,3,4,5]
```

Второй вариант создаёт массив из результата запроса SQL, вызванного внутри функции. Запрос должен возвращать ровно один столбец, а элементы массива будут сформированы из значений столбца, сгенерированного запросом:

Таблица 9.9 — Таблица OPTIONS

ID	OPTION
1	"SIZE"
2	"NAME"
3	"VALUE"

```
SELECT JSON_ARRAY(select option from options)
FROM RDB$DATABASE;

=======================
["size","name","value"]
```

Поведение при значении null по умолчанию равно ABSENT ON NULL (что отличается от значения по умолчанию для JSON_OBJECT).

JSON_ARRAYAGG

Функция JSON_ARRAYAGG позволяет создать массив JSON путём агрегирования данных SQL.

Синтаксис функции:

```
<агрегатный конструктор объектов JSON> ::=
   JSON_ARRAYAGG <левая скобка>
      <запрос>
    [ <поведение при значении null> ]
    [ <выходное значение> ]
   <правая скобка>
```

Таблица 9.10 — Таблица OPTIONS

ID	OPTION
1	"SIZE"
2	"NAME"
3	"VALUE"

Пример построения массива JSON:

```
SELECT JSON_ARRAYAGG(option)
FROM OPTIONS;

=========================
["size","name","value"]
```

Поведение при значении null по умолчанию равно ABSENT ON NULL.

9.2 Предложение SKIP LOCKED

Предложение SKIP LOCKED позволяет пропускать записи, заблокированные другими транзакциями, вместо того чтобы ждать их разблокирования или выдавать ошибку конфликта обновления. Предложение может быть использовано в операторах SELECT WITH LOCK, UPDATE и DELETE.

Синтаксис: redstatement

```
SELECT
  [FIRST ...]
  [SKIP ...]
  FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [{ ROWS ... } | {OFFSET ...} | {FETCH ...}]
  [FOR UPDATE [OF ...]]
  [WITH LOCK [SKIP LOCKED]]

UPDATE <sometable>
  SET ...
  [WHERE ...]
  [PLAN ...]
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]

DELETE FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

Если в операторе есть подоператоры `SKIP LOCKED` и `OFFSET/SKIP/ROWS`, заблокированные строки могут быть пропущены до того, как подоператор `OFFSET/SKIP/ROWS` сможет их учесть, и таким образом будет пропущено больше строк, чем указано в `OFFSET/SKIP/ROWS`.

Примеры:

- Подготовка метаданных:

```
create table emails_queue (
  subject varchar(60) not null,
  text blob sub_type text not null
);

set term !;

create trigger emails_queue_ins after insert on emails_queue
as
begin
    post_event('EMAILS_QUEUE');
end!

set term ;!
```

- Передача данных приложению:

```
insert into emails_queue (subject, text)
  values ('E-mail subject', 'E-mail text...');
commit;
```

- Клиентское приложение:

```
-- Client application can listen to event `EMAILS_QUEUE` to actually send e-
mails using this query:

delete from emails_queue
  rows 10
  skip locked
  returning subject, text;
```

9.3 Поддержка WHEN NOT MATCHED BY SOURCE в операторе MERGE

Синтаксис предложения:

```
<предложение WHEN> ::=
              <предложение WHEN MATCHED> |
              <предложение WHEN NOT MATCHED BY TARGET> |
              <предложение WHEN NOT MATCHED BY SOURCE>

<предложение WHEN NOT MATCHED BY TARGET> ::=
    WHEN NOT MATCHED [ BY TARGET ] [ AND <условие> ] THEN
      INSERT [ (<список таблиц>) ]
        VALUES (<список значений>)

<предложение WHEN NOT MATCHED BY SOURCE> ::=
    WHEN NOT MATCHED BY SOURCE [ AND <условие> ] THEN
      { UPDATE SET <assignment list> | DELETE }
```

`Условие WHEN NOT MATCHED BY TARGET` выполняется, когда исходная запись не совпадает с ни с одной записью в целевой таблице. `INSERT` изменит целевую таблицу.

`Условие WHEN NOT MATCHED BY SOURCE` выполняется, когда целевая запись не совпадает ни с одной записью в источнике. `UPDATE` или `DELETE` изменяют целевую таблицу.

Пример:

```
MERGE
  INTO customers c
  USING new_customers nc
  ON (c.id = nc.id)
  WHEN MATCHED THEN
    UPDATE SET name = nc.name
  WHEN NOT MATCHED BY SOURCE THEN
    DELETE
```

9.4 Поддержка многострочного вывода RETURNING

Начиная с РЕД База Данных 5 операторы `INSERT`, `SELECT`, `UPDATE`, `DELETE`, `UPDATE OR INSERT` и `MERGE`, содержащие предложение `RETURNING` возвращают курсор, то есть они способны вернуть множество записей.

Теперь эти запросы во время подготовки описываются как `isc_info_sql_stmt_select`.

9.5 Вложенные выражения

Синтаксис `DML` теперь позволяет использовать выражения запроса в круглых скобках (`select`, включая предложения `order by`, `offset` и `fetch`, но без `with`) там, где ранее разрешалась только спецификация запроса (`select` без пунктов `with`, `order by`, `offset` и `fetch`).

Это позволяет использовать более гибкие запросы, особенно в операторах `UNION`, и обеспечивает большую совместимость с запросами, генерируемыми некоторыми `ORM`.

Использование выражений запроса в скобках обходится дорого, поскольку они требуют дополнительного контекста запроса. Максимальное количество контекстов запроса в операторе — 255.

Пример:

```
(
  select emp_no, salary, 'lowest' as type
  from employee
  order by salary asc
  fetch first row only
)
union all
(
  select emp_no, salary, 'highest' as type
  from employee
  order by salary desc
  fetch first row only
);
```

9.6 Поддержка PLAN и ORDER BY в операторе MERGE

Оператор MERGE теперь поддерживает предложения PLAN и ORDER BY.

Синтаксис:

```
MERGE INTO target [[AS] <алиас>]
  USING <source> [[AS] <алиас>]
  ON <условие соединения>
  <предложение WHEN> [<предложение WHEN> ...]
  [PLAN <выражение для построения плана>]
  [ORDER BY <выражение для упорядочивания выборки>]
  [RETURNING <список возвращаемых значений> [INTO <список переменных>]]
```

9.7 Поддержка PLAN, ORDER BY и ROWS в операторе UPDATE OR INSERT

Оператор UPDATE OR INSERT теперь поддерживает предложения PLAN, ORDER BY и ROWS.

Синтаксис:

```
UPDATE OR INSERT INTO
  target [(<список столбцов>)]
  [<override_opt>]
  VALUES (<список значений>)
  [MATCHING (<список столбцов>)]
  [PLAN <выражение для построения плана>]
  [ORDER BY <выражение для упорядочивания выборки>]
  [ROWS <m> [TO <n>]]
  [RETURNING <список возвращаемых значений> [INTO <список переменных>]]
```

9.8 Предложение OPTIMIZE FOR

Оператор SELECT теперь поддерживает предложение OPTIMIZE FOR.

Синтаксис:

```
SELECT
  ...
  [WITH LOCK [SKIP LOCKED]]
  [OPTIMIZE FOR {FIRST | LAST} ROWS]
```

Предложение OPTIMIZE FOR позволяет изменить стратегию оптимизатора на уровне текущего SQL оператора. Оно может встречаться только в SELECT операторе верхнего уровня.

Существует две стратегии оптимизации запросов:

- FIRST ROWS - оптимизатор строит план запроса так, чтобы наиболее быстро извлечь только первые строки запроса;
- ALL ROWS - оптимизатор строит план запроса так, чтобы наиболее быстро извлечь все строки запроса.

В большинстве случаев требуется стратегия оптимизации ALL ROWS. Однако если у вас есть приложения с сетками данных, в которых отображаются только первые строки результата, а остальные извлекаются по мере необходимости, то стратегия FIRST ROWS может быть более предпочтительной, поскольку сокращается время отклюика.

По умолчанию используется стратегия оптимизации указанная в параметре OptimizeForFirstRows конфигурационного файла firebird.conf или database.conf. OptimizeForFirstRows = false соответствует стратегии ALL ROWS, OptimizeForFirstRows = true соответствует стратегии FIRST ROWS.

Стратегия оптимизации может быть также изменена на уровне сессии с помощью оператора SET OPTIMIZE. Предложение OPTIMIZE FOR указанное в SQL операторе позволяет переопределить стратегию указанную на уровне сессии.

Предложение OPTIMIZE FOR всегда указывает самым последним в SELECT запросе, но перед предложением INTO.

> Если в SELECT запросе встречаются предложения FIRST ... SKIP, ROWS, OFFSET ... FETCH, то оптимизатор неявно переключаются в режим FIRST ROWS.

9.9 Изменения в литералах

9.9.1 Изменения синтаксиса строковых литералов

Синтаксис строковых литералов был изменён, чтобы полностью поддерживать стандарт SQL синтаксиса. Это означает, что литералы могут "прерываться" пробелами или комментариями. Это можно использовать, например, чтобы разбить длинный литерал на несколько строк, или для использования комментариев в строке.

```
<строковая константа> ::=
  [ <символ начала строки> <строка> ]
    ' [ <символ>... ] '
    [ { <разделитель> ' [ <символ>... ] ' }... ]

<разделитель> ::=
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  { <комментарий> | <пробел> }...
```

Пример:

```
-- пробелы между литералами
select 'ab'
       'cd'
from RDB$DATABASE;
-- вывод: 'abcd'

-- комментарий и пробелы между литералами
select 'ab' /* comment */ 'cd'
from RDB$DATABASE;
-- вывод: 'abcd'
```

9.9.2 Изменения синтаксиса двоичных строк

Синтаксис бинарных строковых литералов был изменен, чтобы полностью поддерживать стандарт SQL синтаксиса. Это означает, что литерал может содержать пробелы для разделения шестнадцатеричных символов, а также может "прерываться" пробелами или комментариями. Это можно использовать, например, для того, чтобы сделать шестнадцатеричную строку более читаемой, сгруппировав символы, или разбить длинный литерал на несколько строк, или написать встроенные комментарии.

```
<двоичный строковый литерал> ::=
  X ' [ <пробел>... ] [ { <шестнадцатеричная строка> [ <пробел>... ]
<шестнадцатеричная строка> [ <пробел>... ] }... ] '
    [ { <разделитель> ' [ <пробелов>... ] [ { <шестнадцатеричная строка> [ <пробел>...
]
    <шестнадцатеричная строка> [ <пробел>... ] }... ] ' }... ]
```

Пример:

```
-- группировка по байтам (пробелы внутри литерала)
select _win1252 x'42 49 4e 41 52 59'
from RDB$DATABASE;
-- output: BINARY

-- пробелы между литералами
select _win1252 x'42494e'
                 '415259'
from RDB$DATABASE;
-- output: BINARY
```

9.9.3 Изменения синтаксиса числовых литералов

Синтаксис числовых литералов был расширен. В РЕД Базе Данных 5 появилась возможность записывать их в двоичном, восьмеричном и шестнадцатеричном формате. Теперь в числовых литералах можно использовать подчёркивание ("_").

Синтаксис числовых констант:

```
<числовой литерал со знаком> ::=
  [ <знак> ] <беззнаковый числовой литерал>

<беззнаковый числовой литерал> ::=
  <точная запись числа>
| <экспоненциальная запись>

<точная запись числа> ::=
  <беззнаковое целое число>
| <беззнаковое десятичное целое число>.[ <беззнаковое десятичное целое число> ]
| .<беззнаковое десятичное целое число>

<знак> ::=
  <знак плюс>
| <знак минус>

<экспоненциальная запись> ::=
  <мантисса> E <экспонента>

<мантисса> ::=
  <точная запись числа>

<экспонента> ::=
  <десятичное целое число со знаком>

<десятичное целое число со знаком> ::=
  [ <знак> ] <беззнаковое десятичное целое число>

<целое число со знаком> ::=
  [ <знак> ] <беззнаковое целое число>

<беззнаковое целое число> ::=
  <беззнаковое десятичное целое число>
| <беззнаковое шестнадцатеричное целое число>
| <беззнаковое восьмеричное целое число>
| <беззнаковое двоичное целое число>

<беззнаковое десятичное целое число> ::=
  <цифра> [ { [ <подчёркивание> ] <цифра> }... ]

<беззнаковое шестнадцатеричное целое число> ::=
  0X { [ <подчёркивание> ] <шестнадцатеричная цифра> }...

<беззнаковое восьмеричное целое число> ::=
  0O { [ <подчёркивание> ] <восьмеричная цифра> }...

<беззнаковое двоичное целое число> ::=
  0B { [ <подчёркивание> ] <двоичная цифра> }

<знак плюс> ::=
  +
| U+002B
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<знак минус> ::=
  -
| U+002D

<точка> ::=
  .
| U+002E

<подчёркивание> ::=
  _
| U+005F

<шестнадцатеричная цифра> ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e |
f

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

<восьмеричная цифра> ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<двоичная цифра> ::=
  0 | 1
```

Примеры использования подчёркиваний в числовых литералах:

```
SELECT 1234567_890 FROM rdb$database;
===========
1234567890

SELECT -1_234_567_890 FROM rdb$database;
============
-1234567890

SELECT 123_45.67809 FROM rdb$database;
============
12345.67809

SELECT 123000E-1_0 FROM rdb$database;
======================
1.230000000000000e-05

SELECT 0o12_34_56_70 FROM rdb$database;
==========
2739128

SELECT -0o_12345670 FROM rdb$database;
=========
-2739128
```

9.10 Улучшения в IN

Обработка предикатов IN <список> теперь линейная, а не рекурсивная, поэтому ограничений на стек во время выполнения нет. Ограничение списка в 1500 элементов было увеличено до 65535 элементов.

Списки, которые являются постоянными, предварительно оцениваются как инварианты и кэшируются в виде бинарного дерева, что ускоряет сравнение, если условие нужно проверить для многих строк или если список значений длинный.

Если список очень длинный или предикат IN не является селективным, сканирование индекса поддерживает поиск групп по указателю на сиблингов (т. е. по горизонтали), а не поиск каждой группы от корня (т. е. по вертикали), что позволяет использовать одно сканирование индекса для всего списка IN.

9.11 Новые выражения и встроенные функции

9.11.1 UNICODE_CHAR

Функция UNICODE_CHAR возвращает UNICODE символ для заданной кодовой точки.

```
UNICODE_CHAR( <номер> )
```

Допустимая кодовая точка UTF-32 вне диапазона суррогатов верхней/нижней границы (от 0xD800 до 0xDFFF). В противном случае будет выдана ошибка.

Пример:

```
select unicode_char(x) from y;
```

9.11.2 UNICODE_VAL

Функция UNICODE_VAL возвращает UTF-32 кодовую точку для первого символа в строке. Возвращает 0 для пустой строки.

```
UNICODE_VAL( <строка> )
```

Пример:

```
select unicode_val(x) from y;
```

9.11.3 QUARTER добавлен в функции EXTRACT, FIRST_DAY и LAST_DAY

Скалярные функции EXTRACT, FIRST_DAY и LAST_DAY теперь поддерживают QUARTER.

Пример:

```
select
  extract(quarter from date '2023-09-21') as Q,
  first_day(of quarter from date '2023-09-21') as Q_START,
  last_day(of quarter from date '2023-09-21') as Q_END
from rdb$database;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
Q  Q_START      Q_END
== ===========  ============
3  2023-07-01   2023-09-30
```

9.12 Соединение с производными таблицами

Производная таблица, определенная с помощью ключевого слова `LATERAL`, называется боковой производной таблицей. Если производная таблица определена как боковая, то ей разрешается ссылаться на другие таблицы в том же предложении `FROM`, но только на те, которые объявлены перед ней в предложении `FROM`.

Примеры соединений с боковыми производными таблицами:

```
select c.name, ox.order_date as last_order, ox.number
from customer c
left join LATERAL (
  select first 1 o.order_date, o.number
  from orders o
  where o.id_customer = c.id
  order by o.ORDER_DATE desc
) as ox on true
-
select dt.population, dt.city_name, c.country_name
from (select distinct country_name from cities) AS c
  cross join LATERAL (
    select first 1 city_name, population
    from cities
    where cities.country_name = c.country_name
    order by population desc
  ) AS dt;
-
select salespeople.name, max_sale.amount, customer_of_max_sale.customer_name
from salespeople,
  LATERAL ( select max(amount) as amount
            from all_sales
            where all_sales.salesperson_id = salespeople.id
  ) as max_sale,
  LATERAL ( select customer_name
            from all_sales
            where all_sales.salesperson_id = salespeople.id
            and all_sales.amount = max_sale.amount
  ) as customer_of_max_sale;
```

9.13 Значение DEFAULT для добавления и обновления

Для операторов `INSERT`, `UPDATE`, `MERGE` и `UPDATE OR INSERT` указывать в списке `VALUES` вместо значения столбца можно использовать ключевое слово `DEFAULT`. В этом случае столбец получит значение по умолчанию, указанное при определении целевой таблицы. Если значение по умолчанию для столбца отсутствует, то столбец получит значение `NULL`.

```
insert into sometable (id, column1)
values (DEFAULT, 'name')
--
update sometable
  set column1 = 'a', column2 = default
```

Если ключевое слово DEFAULT указано для столбца, определенного как GENERATED BY DEFAULT AS IDENTITY, то столбец получит следующее значение идентификации, так как будто этот столбец не был указан в запросе вовсе.

Если DEFAULT указан для вычисляемого столбца, парсер обработает его, но он не будет иметь никакого эффекта.

9.13.1 DEFAULT и DEFAULT VALUES

Предложение DEFAULT применяется к отдельному столбцу в списке VALUES, в то время как DEFAULT VALUES применяется ко всей вставляемой строке. Оператор INSERT INTO sometable DEFAULT VALUES эквивалентен INSERT INTO sometable VALUES (DEFAULT, ...) с DEFAULT в списке VALUES для каждого столбца в sometable.

9.14 Предложение OVERRIDING для столбцов IDENTITY

Значения столбцов идентификации (GENERATED BY DEFAULT AS IDENTITY) могут быть переопределены в операторах INSERT, UPDATE OR INSERT, MERGE. Для этого просто достаточно указать значение столбца в списке значений. Однако для столбцов определённых как GENERATED ALWAYS это недопустимо. Директива OVERRIDING SYSTEM VALUE позволяет заменить сгенерированное системой значение на значение указанное пользователем. Директива OVERRIDING SYSTEM VALUE вызовет ошибку, если в таблице нет столбцов идентификации или если они определены как GENERATED BY DEFAULT AS IDENTITY.

```
CREATE TABLE objects (
  id INT GENERATED ALWAYS AS IDENTITY,
  name CHAR(50));

INSERT INTO objects (id, name)
OVERRIDING SYSTEM VALUE
VALUES (11, 'Laptop' );
-- будет вставлено значение с кодом 11
```

Директива OVERRIDE USER VALUE выполняет обратную задачу, т.е. заменяет значение указанное пользователем на значение сгенерированное системой, если столбец идентификации определён как GENERATED BY DEFAULT AS IDENTITY. Директива OVERRIDING USER VALUE вызовет ошибку, если в таблице нет столбцов идентификации или если они определены как GENERATED ALWAYS AS IDENTITY.

```
CREATE TABLE objects (
  id INT GENERATED BY DEFAULT AS IDENTITY,
  name CHAR(50));

INSERT INTO objects (id, name)
OVERRIDING SYSTEM VALUE
VALUES (12, 'Laptop' );
-- значение 12 будет проигнорировано
```

9.15 Улучшения оконных функций

Предложение OVER для оконных функций теперь поддерживает не только PARTITION и ORDER, но и фреймы и именованные окна, которые могут быть повторно использованы в одном и том же запросе.

```
<оконная функция> ::=
  <имя оконной функции>([<выражение> [, <выражение> ...]])
    OVER {<спецификация окна> | имя окна}

<спецификация окна> ::=
  ([<имя окна>][<выражение секционирования>][<выражение сортировки>][<рамка окна>])

<выражение секционирования> ::=
  PARTITION BY <выражение> [, <выражение> ...]

<выражение сортировки> ::=
  ORDER BY <выражение>  [{ASC|DESC}] [NULLS {FIRST|LAST}]
          [, <выражение>  [{ASC|DESC}] [NULLS {FIRST|LAST}] ... ]

<рамка окна> ::=
  {RANGE | ROWS} <диапазон окна>

<диапазон окна> ::=
  {<начало окна> | <границы окна>}

<начало окна> ::=
  {UNBOUNDED PRECEDING | <выражение> PRECEDING | CURRENT ROW}

<границы окна> ::=
  BETWEEN <граница 1> AND <граница 2>

<граница 1> ::=
  {UNBOUNDED PRECEDING | <выражение> PRECEDING | <выражение> FOLLOWING | CURRENT ROW}

<граница 2> ::=
  {UNBOUNDED FOLLOWING | <выражение> PRECEDING | <выражение> FOLLOWING | CURRENT ROW}

<определение запроса> ::=
  SELECT
    [FIRST <значение>] [SKIP <значение>]
    [DISTINCT | ALL]
    <выходное поле> [, <выходное поле>]
  FROM <источники> [<соединения (joins)>]
  [WHERE <условие выборки>]
  [GROUP BY <условие группирование выбранных данных>
  [HAVING <условие выборки>]]
  [WINDOW <спецификация окна> [, <спецификация окна>] ...]
  [UNION [DISTINCT | ALL] <другой набор данных>]
  [PLAN <выражение для плана поиска>]

<именованное окно> ::=
  WINDOW <определение окна> [, <определение окна>] ...
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
<определение окна> ::=
  <имя окна> AS <спецификация окна>

<ранжирующая функция> ::=
    DENSE_RANK
  | RANK
  | PERCENT_RANK
  | CUME_DIST
  | NTILE
  | ROW_NUMBER

<навигационная функция> ::= LEAD | LAG | FIRST_VALUE | LAST_VALUE | NTH_VALUE
```

9.15.1 Рамки для оконных функций

Синтаксис рамок:

```
<рамка окна> ::=
{RANGE | ROWS} <диапазон окна>

<диапазон окна> ::=
  {<начало окна> | <границы окна>}

<начало окна> ::=
  {UNBOUNDED PRECEDING | <выражение> PRECEDING | CURRENT ROW}

<границы окна> ::=
  BETWEEN <граница 1> AND <граница 2>

<граница 1> ::=
  {UNBOUNDED PRECEDING | <выражение> PRECEDING | <выражение> FOLLOWING | CURRENT ROW}

<граница 2> ::=
  {UNBOUNDED FOLLOWING | <выражение> PRECEDING | <выражение> FOLLOWING | CURRENT ROW}
```

Набор строк внутри секции которым оперирует оконная функция называется рамкой окна (кадры окна). Рамка окна состоит из трёх частей: единица (`unit`), начальная граница и конечная граница. В качестве единицы может быть использовано ключевые слова RANGE или ROWS, которые указывают каким образом определены границы окна. Границы окна определяются следующими выражениями:

- `<выражение> PRECEDING`
- `<выражение> FOLLOWING`
- `CURRENT ROW`

Если рамка окна задаётся с помощью предложения `RANGE`, то предложение `ORDER BY` может содержать только одно выражение и выражение должно быть числового типа, `DATE`, `TIME` или `TIMESTAMP`. Для границ `PRECEDING` и `FOLLOWING` выражения добавляются и вычитаются к выражению указанному в `ORDER BY`, таким образом получаются границы значений для рамки. Для `CURRENT ROW` выражение в `ORDER BY` используется как есть.

Затем все строки (внутри секции) между границам считаются частью результирующей рамки окна.

Если рамка окна задаётся с помощью предложения `ROWS`, то на предложение `ORDER BY` не накла-

дывается ограничений на количество и типы выражений. В этом случае фраза `<выражение> PRECEDING` указывает количество строк предшествующее текущей строке, соответственно фраза `<выражение> FOLLOWING` указывает количество строк после текущей строки.

`UNBOUNDED PRECEDING` и `UNBOUNDED FOLLOWING` работают одинаково для предложений `ROWS` и `RANGE`. Фраза `UNBOUNDED PRECEDING` указывает, что окно начинается с первой строки секции. `UNBOUNDED PRECEDING` может быть указано только как начальная точка окна. Фраза `UNBOUNDED FOLLOWING` указывает, что окно заканчивается последней строкой секции. `UNBOUNDED FOLLOWING` может быть указано только как конечная точка окна.

Если указана только начальная точка окна, то конечной точкой окна считается `CURRENT ROW`. Например, если указано `ROWS 1 PRECEDING`, то это аналогично указанию `ROWS BETWEEN 1 PRECEDING AND CURRENT ROW`.

Некоторые оконные функции игнорируют выражение рамки:

- `ROW_NUMBER`, `LAG` и `LEAD` всегда работают как `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.
- `DENSE_RANK`, `RANK`, `PERCENT_RANK` и `CUME_DIST` работают как `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.
- `FIRST_VALUE`, `LAST_VALUE` и `NTH_VALUE` работают на рамке, но `RANGE` работает идентично `ROWS`.

Навигационные функции с фреймами

Навигационные функции могут работать с фреймами:

```
<навигационная оконная функция> ::=
  FIRST_VALUE(<выражение>) |
  LAST_VALUE(<выражение>) |
  NTH_VALUE(<выражение>, <offset>) [FROM FIRST | FROM LAST] |
  LAG(<expr> [ [, <offset> [, <default> ] ] ) |
  LEAD(<expr> [ [, <offset> [, <default> ] ] )
```

По умолчанию используется `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, что может привести к странным результатам, если с такой рамкой работают `FIRST_VALUE`, `NTH_VALUE` или `LAST_VALUE`.

Примеры использования рамок

Когда используется `ORDER BY`, но опущено предложение рамки, рамка по умолчанию, приводит к странному поведению запроса, приведенного ниже, для столбца `sum_salary`. Он суммирует от начала рамки до текущего ключа, вместо того чтобы суммировать всю рамку.

```
select
    id,
    salary,
    sum(salary) over (order by salary) sum_salary
  from employee
  order by salary;

| id | salary | sum_salary |
|---:|-------:|-----------:|
|  3 |   8.00 |       8.00 |
|  4 |   9.00 |      17.00 |
|  1 |  10.00 |      37.00 |
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
|  5 |  10.00 |      37.00 |
|  2 |  12.00 |      49.00 |
```

Можно явно задать рамку для суммирования всего раздела, как показано ниже:

```
select
    id,
    salary,
    sum(salary) over (
      order by salary
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
                    ) sum_salary
  from employee
  order by salary;

| id | salary | sum_salary |
|---:|-------:|-----------:|
|  3 |   8.00 |      49.00 |
|  4 |   9.00 |      49.00 |
|  1 |  10.00 |      49.00 |
|  5 |  10.00 |      49.00 |
|  2 |  12.00 |      49.00 |
```

Этот запрос "исправляет" поведение рамки по умолчанию, выдавая результат, похожий на простое предложение `OVER ()` без `ORDER BY`.

Мы можем использовать диапазона рамки, чтобы вычислить количество сотрудников с зарплатой между `зарплата сотрудника - 1` и `его зарплата + 1` с помощью этого запроса:

```
select
    id,
    salary,
    count(*) over (
      order by salary
      RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) range_count
  from employee
  order by salary;

| id | salary | range_count |
|---:|-------:|------------:|
|  3 |   8.00 |           2 |
|  4 |   9.00 |           4 |
|  1 |  10.00 |           3 |
|  5 |  10.00 |           3 |
|  2 |  12.00 |           1 |
```

9.15.2 Именованные окна

Для того чтобы не писать каждый раз сложные выражения для задания окна, имя окна можно задать в предложении `WINDOW`. Имя окна может быть использовано в предложении `OVER` для ссылки на определение окна, кроме того оно может быть использовано в качестве базового окна для другого именованного или встроенного (в предложении `OVER`) окна. Окна с рамкой (с предложениями `RANGE` и `ROWS`)

не могут быть использованы в качестве базового окна, но могут быть использованы в предложении OVER window_name. Окно, которое использует ссылку на базовое окно, не может иметь предложение PARTITION BY и не может переопределять сортировку с помощью предложения ORDER BY.

```
SELECT
  id,
  department,
  salary,
  count(*) OVER w1,
  first_value(salary) OVER w2,
  last_value(salary) OVER w2,
  sum(salary) over (w2 ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) AS s
FROM employee
WINDOW w1 AS (PARTITION BY department),
  w2 AS (w1 ORDER BY salary)
ORDER BY department, salary;
```

9.15.3 Ранжирующие функции

Ранжирующие функции вычисляют порядковый номер ранга внутри секции окна.

```
<ранжирующая функция> ::=
  DENSE_RANK
| RANK
| PERCENT_RANK
| CUME_DIST
| NTILE
| ROW_NUMBER
```

- PERCENT_RANK - вычисления относительного положения значения в секции или результирующем наборе запроса.
- CUME_DIST - рассчитывается как (число строк, предшествующих или равных текущей) / (общее число строк).
- NTILE - распределяет строки упорядоченной секции в заданное количество групп так, чтобы размеры групп были максимально близки.

Эти функции могут применяться с использованием секционирования и сортировки и без них. Однако их использование без сортировки почти никогда не имеет смысла. Функции ранжирования могут быть использованы для создания различных типов инкрементных счётчиков. Рассмотрим SUM(1) OVER (ORDER BY SALARY) в качестве примера того, что они могут делать, каждая из них различным образом. Ниже приведён пример запроса, который позволяет сравнить их поведение по сравнению с SUM.

```
SELECT
  id,
  salary,
  DENSE_RANK() OVER (ORDER BY salary),
  RANK() OVER (ORDER BY salary),
  PERCENT_RANK() OVER(ORDER BY salary),
  CUME_DIST() OVER(ORDER BY salary),
  NTILE(3) OVER(ORDER BY salary),
  ROW_NUMBER() OVER (ORDER BY salary),
  SUM(1) OVER (ORDER BY salary)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
FROM employee
ORDER BY salary;

id  salary   dense_rank   rank   percent_rank   cume_dist   ntile   row_number   sum
3     8.00            1      1      0.0000000  0.20000000       1            1     1
4     9.00            2      2      0.2500000  0.40000000       1            2     2
1    10.00            3      3      0.5000000  0.80000000       2            3     4
5    10.00            3      3      0.5000000  0.80000000       2            4     4
2    12.00            4      5      1.0000000  1.00000000       3            5     5
```

9.16 Предложение FILTER для агрегатных функций

Предложение FILTER дополняет агрегатные функции предложением WHERE. Возвращаемый набор - это совокупность строк, удовлетворяющих условиям как основного предложения WHERE, так и условий, содержащихся в FILTER.

Синтаксис:

```
<агрегатная функция> [FILTER (WHERE <условие>)] [OVER (<окно>)]
```

Его можно рассматривать как сокращение для ситуаций, когда используется агрегатная функция с условием (decode, case, iif) для игнорирования некоторых значений, которые будут учитываться при агрегации.

Это предложение можно использовать с любыми агрегатными функциями в агрегированных или оконных (OVER) операторах, но не с функциями работы с окнами, такими как DENSE_RANK.

Пример:

Предположим, у вас есть запрос, в котором нужно подсчитать количество статусов = 'A' и количество статусов = 'E' как разные столбцы. Старый способ сделать это был бы следующим:

```
select count(decode(status, 'A', 1)) status_a,
       count(decode(status, 'E', 1)) status_e
  from data;
```

Предложение FILTER позволяет выразить эти условия в более явном виде:

```
select count(*) filter (where status = 'A') status_a,
       count(*) filter (where status = 'E') status_e
  from data;
```

9.17 AUTOCOMMIT для SET TRANSACTION

Режим автоматической фиксации теперь поддерживается оператором SET TRANSACTION.

```
SET TRANSACTION SNAPSHOT NO WAIT AUTO COMMIT;
```

9.18 Совместное использование снимков транзакций

С помощью этой функции можно создавать параллельные процессы (использующие разные подключения), читающие согласованные данные из базы данных. Например, процесс резервного копирования может создавать несколько потоков, параллельно считывающих данные из базы. Или веб-сервис может диспетчеризировать распределенные подсервисы, выполняющие обработку параллельно.

Для этого оператор `SET TRANSACTION` дополнен опцией `SNAPSHOT [ AT NUMBER snapshot_number ]`. Также эту возможность можно использовать через `API`, для этого добавлен новый элемент `isc_tpb_at_snapshot_number <длина снапшота> <номера снапшота>`.

Узнать `snapshot_number` активной транзакции можно с помощью `RDB$GET_CONTEXT('SYSTEM', 'SNAPSHOT_NUMBER')` или через `API` тегом `fb_info_tra_snapshot_number`. Обратите внимание, что `snapshot_number`, передаваемый новой транзакции, должен быть снимком текущей активной транзакции.

```
SET TRANSACTION SNAPSHOT AT NUMBER 12345;
```

9.19 Выражения и встроенные функции

9.19.1 Новые функции и выражения

Функции и выражения для работы с часовыми поясами

Выражение AT

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

```
<выражение> AT {TIME ZONE '<часовой пояс>' | LOCAL}

<часовой пояс> ::=
  <регион часового пояса> | [+/-] <разница часов с GMT> [:<разница минут с GMT>]
```

Пример:

```
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;
```

Выражение LOCALTIME

`LOCALTIME` типа `TIME WITHOUT TIME ZONE` возвращает текущее время сервера в часовом поясе сессии. В обращении к контекстной переменной `LOCALTIME` можно указать количество знаков в долях секунды:

```
LOCALTIME [(<количество знаков в долях секунды>)]
```

Пример:

```
select localtime from rdb$database;
```

Выражение LOCALTIMESTAMP

Контекстная переменная `LOCALTIMESTAMP` типа `TIMESTAMP WITHOUT TIME ZONE` возвращает текущую дату и текущее время сервера в часовом поясе сессии.

```
LOCALTIMESTAMP [(<количество знаков в долях секунды>)]
```

Пример:

```
select localtimestamp from rdb$database;
```

Новые функции для работы с датой и временем

FIRST_DAY

Функция возвращает первый день года, месяца или недели для заданной даты.

```
FIRST_DAY( OF {YEAR | MONTH | WEEK | QUARTER} FROM <дата> )
```

Первым днём недели считается воскресенье, как это возвращает функция EXTRACT с WEEKDAY.

Когда в качестве аргумента функции передаётся выражение типа TIMESTAMP, то возвращаемое значение сохраняет временную часть.

Пример:

```
select first_day(of month from current_date) from rdb$database;
select first_day(of year from current_timestamp) from rdb$database;
select first_day(of week from date '2017-11-01') from rdb$database;
```

LAST_DAY

Функция возвращает последний день года, месяца или недели для заданной даты.

```
LAST_DAY( OF {YEAR | MONTH | WEEK | QUARTER} FROM <дата> )
```

Последним днём недели считается суббота, как это возвращает функция EXTRACT с WEEKDAY.

Когда в качестве аргумента функции передаётся выражение типа TIMESTAMP, то возвращаемое значение сохраняет временную часть.

Пример:

```
select last_day(of month from current_date) from rdb$database;
select last_day(of year from current_timestamp) from rdb$database;
select last_day(of week from date '2017-11-01') from rdb$database;
```

Функции безопасности

Функция RDB$SYSTEM_PRIVILEGE

Функция RDB$SYSTEM_PRIVILEGE возвращает используется ли системная привилегия текущим соединением:

```
RDB$SYSTEM_PRIVILEGE (<системная привилегия>)
```

Функция RDB$ROLE_IN_USE

Функция RDB$ROLE_IN_USE показываает используется ли роль текущим пользователем.

```
RDB$ROLE_IN_USE (<имя роли>)
```

Данная функция позволяет проверить использование любой роли: указанной явно (при входе в систему или изменённой с помощью оператора SET ROLE) и назначенной неявно (роли назначенные пользователю с использованием предложения DEFAULT).

Узнать список активных в данный момент ролей можно следующим образом:

```
SELECT * FROM RDB$ROLES WHERE RDB$ROLE_IN_USE(RDB$ROLE_NAME)
```

Функции для DECFLOAT

COMPARE_DECFLOAT

Функция COMPARE_DECFLOAT сравнивает два значения типа DECFLOAT.

```
COMPARE_DECFLOAT (<значение1>, <значение2>)
```

Результирующие значения функции COMPARE_DECFLOAT:

- 0 - Значения равны;
- 1 - Первое значение меньше, чем второе;
- 2 - Первое значение больше, чем второе;
- 3 - Значения не упорядочены (одно или оба NAN / SNAN).

В отличие от операторов сравнения (<, >, = и др.) сравнение с помощью COMPARE_DECFLOAT является точным, т.е.

```
COMPARE_DECFLOAT(2.17, 2.170)
```

вернёт 2, а не 0

NORMALIZE_DECFLOAT

Функция NORMALIZE_DECFLOAT возвращает число в нормализованном виде. Это означает, что для любого ненулевого значения удаляются завершающие нули с соответствующей коррекцией экспоненты.

```
NORMALIZE_DECFLOAT (<значение>)
```

Пример:

```
NORMALIZE_DECFLOAT(12.00)
12
NORMALIZE_DECFLOAT(120)
1.2E+2
```

QUANTIZE

Функция QUANTIZE возвращает значение первого аргумента масштабированным с использованием второго аргумента в качестве шаблона.

```
QUANTIZE (<значение>, <шаблон>)
```

Функция QUANTIZE возвращает значение типа DECFLOAT, равное по значению (за исключением любого округления) и знаку первому аргументу, а также экспоненте, равной по значению экспоненте, указанной в параметре <шаблон>. Функцию QUANTIZE можно использовать для реализации округления с точностью до нужного знака, например, округление до ближайшего десятка с использованием установленного режима округления DECFLOAT.

Для значения шаблона не никаких ограничений, тем не менее при использовании SNaN функция

выдаст исключение, при использовании NULL результатом будет NULL и т.д.

```
select v, pic, quantize(v, pic) from examples;

V     PIC    QUANTIZE
===== =====  ===========
3.16  0.001  3.160
3.16  0.01   3.16
3.16  0.1    3.2
3.16  1      3
3.16  1E+1   0E+1
-0.1  1      -0
0     1E+5   0E+5
316   0.1    316.0
316   1      316
316   1E+1   3.2E+2
316   1E+2   3E+2
```

TOTALORDER

Функция TOTALORDER сравнивает два значения типа DECFLOAT, включая специальные значения. Сравнение является точным.

```
TOTALORDER (<значение1>, <значение2>)
```

Результирующие значения функции TOTALORDER:

- -1 - Первое значение меньше второго;
- 0 - Значения равны;
- 1 - Первое значение больше второго.

Значения DECFLOAT упорядочиваются следующим образом:

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

Функция RDB$GET_TRANSACTION_CN

Возвращает номер фиксации (Commit Number) заданной транзакции.

```
RDB$GET_TRANSACTION_CN (<номер транзакции>)
```

Внутренние механизмы РЕД базы Данных используют беззнаковое 8-байтное целое для Commit Number и беззнаковое 6-байтное целое для номера транзакции. Поэтому, не смотря на то, что язык SQL не имеет беззнаковых целых, а RDB$GET_TRANSACTION_CN возвращает знаковый BIGINT, невозможно увидеть отрицательный номер подтверждения, за исключением нескольких специальных значений, используемых для неподтверждённых транзакций.

Таким образом, числа возвращаемые RDB$GET_TRANSACTION_CN могут иметь следующие значения:

- -2 - Мёртвые транзакции (отмененные);
- -1 - Зависшие транзакции (в состоянии limbo 2PC транзакций);
- 0 - Активные транзакции;
- 1 - Для транзакций подтверждённых до старта базы данных или с номером меньше чем OIT (Oldest Interesting Transaction);
- >1 - Транзакции подтверждённые после старта базы данных;

- NULL - Если номер транзакции равен NULL или больше чем Next Transaction.

```
select rdb$get_transaction_cn(current_transaction) from rdb$database;
select rdb$get_transaction_cn(123) from rdb$database;
```

MAKE_DBKEY

MAKE_DBKEY создает значение DBKEY, используя имя или идентификатор таблицы, номер записи и (не обязательно) номера страницы данных и страницы указателей.

```
MAKE_DBKEY (<таблица>, <номер записи> [, <номер страницы данных> [, <номер страницы
указателей>]])
```

Параметры функции имеют следующие значения:

- Таблица - имя или идентификатор таблицы;
- Номер записи - либо абсолютный номер записи (если аргументы <номер страницы данных> и <номер страницы указателей> отсутствуют), либо относительный номер записи (если присутствует аргумент <номер страницы данных>);
- Номер страницы данных - может быть либо абсолютным (если аргумент <номер страницы указателей> не задан), либо относительным (если указанный аргумент задан);
- Номер страницы указателя - логический номер страницы указателя в таблице.

Примеры работы функции MAKE_DBKEY:

Выбор записи с использованием имени таблицы (обратите внимание, что имя таблицы указано в верхнем регистре):

```
select *
from rdb$relations
where rdb$db_key = make_dbkey(RDB$RELATIONS, 0)
```

Выбор записи с использованием идентификатора таблицы:

```
select *
from rdb$relations
where rdb$db_key = make_dbkey(6, 0)
```

Выбор всех записей, физически находящихся на первой странице данных:

```
select *
from rdb$relations
where rdb$db_key >= make_dbkey(6, 0, 0)
and rdb$db_key < make_dbkey(6, 0, 1)
```

Выбор всех записей, физически находящихся на первой странице данных шестой страницы указателей:

```
select *
from SOMETABLE
where rdb$db_key >= make_dbkey('SOMETABLE', 0, 0, 5)
and rdb$db_key < make_dbkey('SOMETABLE', 0, 1, 5)
```

BASE64_ENCODE

Функция BASE64_ENCODE кодирует входные данные в представлении BASE64. Функция может работать как с символьной строкой, так и с BLOB.

```
BASE64_ENCODE (<двоичные данные>)
```

BASE64_DECODE

Функция BASE64_DECODE декодирует входные данные из представления BASE64. Функция может работать как с символьной строкой, так и с BLOB.

```
BASE64_ENCODE (<данные в base64>)
```

HEX_ENCODE

Функция HEX_ENCODE кодирует двоичные данные в шестнадцатеричное представлении. Функция может работать как с символьной строкой, так и с BLOB.

```
HEX_ENCODE (<двоичные данные>)
```

HEX_DECODE

Функция HEX_DECODE декодирует данные в шестнадцатеричном представлении в двоичные данные. Функция может работать как с символьной строкой, так и с BLOB.

```
HEX_DECODE (<шестнадцатеричные данные>)
```

CRYPT_HASH

Принимает аргумент, который может быть полем, переменной или выражением любого типа, распознаваемого DSQL/PSQL, и возвращает хэш-значение, вычисленный из входного аргумента с помощью указанного алгоритма.

```
CRYPT_HASH( <значение> USING <алгоритм> )

<алгоритм> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Пример:

```
select crypt_hash(job_title using sha256) from job;
```

BLOB_APPEND

Оператор || с BLOB-аргументами создает временный BLOB для каждой пары аргументов, содержащих BLOB. Это может привести к чрезмерному потреблению памяти и увеличению файла базы данных. Функция BLOB_APPEND предназначена для объединения BLOB без создания промежуточных объектов.

Чтобы достичь этого, результирующий BLOB остается открытым для записи, а не закрывается сразу после заполнения данными. Данные в такой BLOB можно добавлять столько раз, сколько потребуется. Сервер помечает такой BLOB внутренним флагом BLB_close_on_read и закрывает его при необходимости.

```
BLOB_APPEND( <значение> [, <значение>, ... <значение> ] )
```

В зависимости от значения первого аргумента возможны следующие варианты поведения:

- NULL – создается новый BLOB (незакрытый, с флагом BLB_close_on_read).
- постоянный BLOB (из таблицы) или временный BLOB, который уже был закрыт
- создается новый BLOB (незакрытый, с флагом BLB_close_on_read), его содержимое копируется из первого аргумента.
- временный незакрытый BLOB – он будет использоваться в дальнейшем.
- другие типы данных преобразуются в строку, создается новый BLOB (незакрытый, с флагом BLB_close_on_read), его содержимое копируется из этой строки.

Другие аргументы могут быть любого типа, для них определено следующее поведение:

- значения NULL игнорируются.
- не BLOB преобразуются в строки и добавляются к результату.
- BLOB при необходимости переводятся в кодировку первого аргумента, и их содержимое добавляется к результату.

Функция BLOB_APPEND возвращает временный незакрытый BLOB с флагом BLB_close_on_read. Это либо новый BLOB, либо тот, который указан в качестве первого аргумента. Таким образом, серия операций типа `blob = BLOB_APPEND (blob, ...)` приведет к созданию не более одного BLOB (если только вы не попытаетесь добавить BLOB к самому себе). Этот BLOB будет закрыт, когда клиент прочитает его, назначит его таблице или использует в других выражениях, требующих чтения содержимого.

Проверка BLOB на наличие значения NULL с помощью оператора IS [NOT] NULL не считывает его, и поэтому BLOB не будет закрыт после такой проверки.

Используйте функции LIST или BLOB_APPEND для объединения BLOB. Это уменьшает потребление памяти и дисковый ввод-вывод, а также предотвращает рост базы данных из-за создания большого количества временных BLOB.

```
execute block
returns (b blob sub_type text)
as
  begin
  -- создает новый временный незакрытый BLOB
  -- записывает в него строку из второго аргумента
  b = blob_append(null, 'Hello ');

  -- добавляет две строки во временный BLOB, не закрывая его
  b = blob_append(b, 'World', '!');

  -- сравнение BLOB со строкой приведет к его закрытию, потому что BLOB должен быть
прочитан
  if (b = 'Hello World!') then
  begin
  -- ...
  end
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
  -- создает временный закрытый BLOB, добавляя к нему строку
  b = b || 'Close';
  suspend;
end
```

Функция UNLIST

Функция UNLIST преобразует входную строку в набор записей, состоящий из одного столбца.

Синтаксис функции:

```
UNLIST(<входная строка> [, <разделитель>] [<возвращаемый тип данных>]) AS <псевдоним
набора записей> [(<псевдоним результирующего столбца>)]

<разделитель> ::= <строковый литерал>

<возвращаемый тип данных> ::= RETURNING <тип данных>
```

Описание параметров функции:

- `Входная строка` – Входное значение, представляющее собой любое выражение, возвращающее строку символов (строковый литерал, столбец таблицы, константа, переменная, выражение и т.д). Также может быть значением `BLOB TEXT`, ограниченным 32Кб. Обязательный параметр.
- `Разделитель` – Строковый литерал, заключенный в апострофы, который в результирующем списке будет отделять одно полученное значение от другого. Также может содержать выражение, возвращающее строку символов. Может быть значением `BLOB TEXT`, ограниченным 32Кб. Если в качестве разделителя указана пустая строка, то значения будут выведены одной записью. Если разделитель не задан, то будет использован символ запятой.
- `Возвращаемый тип данных` – Тип данных, к которому будут приведены значения в результирующем наборе записей. Если возвращаемый тип не задан, то по умолчанию используется тип `VARCHAR(32)`. В качестве возвращаемого типа может быть указан домен.
- `Псевдоним набора записей` – Псевдоним набора записей, возвращаемого функцией `UNLIST`. Обязательный параметр.
- `Псевдоним результирующего столбца` – Псевдоним столбца, возвращаемого функцией `UNLIST`. Если псевдоним не указан, то по умолчанию используется `UNLIST`.

Пример работы функции:

```
SELECT * FROM UNLIST('1,2,3,4') AS A;

UNLIST
=======
1
2
3
4
```

Пример использования UNLIST в качестве источника данных:

```
CREATE TABLE TEST_TABLE (ID INT);
INSERT INTO TEST_TABLE (ID) SELECT * FROM UNLIST('1,2,3,4' RETURNING INT) AS A;
SELECT * FROM TEST_TABLE;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
ID
============
1
2
3
4
```

9.19.2 Изменения в встроенных функциях и выражениях

Выражение EXTRACT

Добавлено два новых аргумента в выражение EXTRACT:

- TIMEZONE_HOUR - возвращает смещение часов часового пояса: целое число от -23 до 23;
- TIMEZONE_MINUTE - возвращает смещение минут часового пояса: целое число от -59 до 59.

```
select extract(timezone_hour from current_time) from rdb$database;
select extract(timezone_minute from current_timestamp) from rdb$database;
```

Изменения в CURRENT_TIME и CURRENT_TIMESTAMP

Теперь CURRENT_TIME и CURRENT_TIMESTAMP возвращают TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE соответственно. В предыдущих версиях CURRENT_TIME и CURRENT_TIMESTAMP возвращали системное время без временной зоны.

HASH

В HASH добавлено необязятельное предложение USING. Функция возвращает хэш-значение, соответствующее входной строке, используя для этого указанный алгоритм хэширования.

```
HASH( <значение> [ USING <алгоритм> ] )

<алгоритм> ::= { CRC32 }
```

Пример:

```
select hash(x using crc32) from y;

select hash(x) from y; -- not recommended
```

SUBSTRING

Теперь допускается начальная позиция SUBSTRING меньше 1. Она обладает некоторыми свойствами, которые необходимо учитывать для предсказания конца возвращаемого строкового значения.

```
select substring('abcdef' from 0) from rdb$database
-- Expected result: 'abcdef'

select substring('abcdef' from 0 for 2) from rdb$database
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
-- Expected result: 'a' (and NOT 'ab', because there is
-- "nothing" at position 0)

select substring('abcdef' from -5 for 2) from rdb$database
-- Expected result: ''
```

Длина строки считается от указанной начальной позиции, а не от начала строки, поэтому возвращаемая строка может быть короче указанной длины или даже пустой.

9.19.3 Изменения UDF

Многие UDF предыдущих версий стали встроенными функциями. Сама функция UDF сильно устарела. Большинство оставшихся UDF в библиотеках ib_udf и fbudf теперь имеют аналоги, либо в виде UDR в новой библиотеке udf_compat, либо в виде прекомпилированных PSQL-функций.

В /misc/upgrade/v4.0/ находится скрипт обновления существующих UDF.

Новая UDR GetExactTimestampUTC

Новая функция UDR GetExactTimestampUTC в библиотеке udf_compat не принимает никаких входных аргументов и возвращает значение TIMESTAMP WITH TIME ZONE в момент вызова функции.

Старая функция GetExactTimestamp была переделана в хранимую функцию, возвращающую, как и раньше, значение TIMESTAMP WITHOUT TIME ZONE в момент вызова функции.

9.19.4 Новые контекстные переменные пространства имён SYSTEM

Следующие контекстные переменные были добавлены в пространство имен SYSTEM в RDB$GET_CONTEXT:

- DB_FILE_ID - Идентификатор текущей базы данных на уровне файловой системы.
- DB_GUID - GUID текущей базы данных.
- EFFECTIVE_USER - Эффективный пользователь в текущий момент. Указывает пользователя с привилегиями которого в текущий момент времени выполняется процедура, функция или триггер.
- EXT_CONN_POOL_SIZE - Размер пула.
- EXT_CONN_POOL_LIFETIME - Время жизни неактивных соединений.
- EXT_CONN_POOL_IDLE_COUNT - Текущее количество неактивных соединений в пуле.
- EXT_CONN_POOL_ACTIVE_COUNT - Текущее количество активных соединений в пуле.
- GLOBAL_CN - Последнее значение текущего глобального счётчика Commit Number.
- REPLICATION_SEQUENCE - Текущее значение последовательности репликации (номер последнего сегмента, записанного в журнал репликации).
- SESSION_IDLE_TIMEOUT - Содержит текущее значение тайм-аут простоя соединения в секундах, который был установлен на уровне соединения, или ноль, если таймаут не был установлен
- SESSION_TIMEZONE - Часовой пояс текущего соединения.
- SNAPSHOT_NUMBER - Номер моментального снимка базы данных: уровня транзакции (для транзакции SNAPSHOT или CONSISTENCY) или уровня запроса (для транзакции READ COMMITTED READ CONSISTENCY). NULL, если моментальный снимок не существует.
- STATEMENT_TIMEOUT - Содержит текущее значение тайм-аута выполнения оператора в миллисекундах, который был установлен на уровне подключения, или ноль, если таймаут не был

установлен.

- WIRE_CRYPT_PLUGIN - Если соединение зашифровано - возвращает имя текущего плагина, иначе NULL.

9.20 Другие улучшения DML

9.20.1 Уточнение сообщения об ошибке при недопустимой операции записи

Если в операции UPDATE ... SET xxx указан столбец, доступный только для чтения, в сообщении об ошибке теперь указывается имя столбца, на который производится запись.

9.20.2 Уточнение сообщения об ошибке для выражений индексов

Если выражение индекса завершилось ошибкой, сообщение теперь будет включать имя индекса.

9.20.3 Поддержка RETURNING *

Сервер теперь поддерживает синтаксис RETURNING * и его варианты для возвращения полного набора значений полей после фиксации строки, которая была вставлена, обновлена или удалена. Синтаксис и семантика RETURNING * аналогичны SELECT *.

Пример:

```
INSERT INTO T1 (F1, F2) VALUES (:F1, :F2) RETURNING *

DELETE FROM T1 WHERE F1 = 1 RETURNING *

UPDATE T1 SET F2 = F2 * 10 RETURNING OLD.*, NEW.*
```

Глава 10

Процедурный SQL (PSQL)

10.1 Подпрограммы могут обращаться к переменным, определенным на внешнем уровне

Подпрограммы теперь могут читать из и записывать в переменные и параметры внешней/родительской подпрограммы. Это не относится к курсорам: подпрограммы не могут обращаться к курсорам своего родителя.

Переменные и параметры, к которым обращаются подпрограммы, могут иметь небольшое снижение производительности (даже в основной подпрограмме) при чтении.

10.2 Рекурсия для подпрограмм

Подпрограммы могут быть рекурсивными или вызывать другие подпрограммы.

Пара рекурсивных подфункций в `EXECUTE BLOCK`:

```
execute block returns (i integer, o integer)
as
    -- Recursive function without forward declaration.
    declare function fibonacci(n integer) returns integer
    as
    begin
      if (n = 0 or n = 1) then
      return n;
    else
      return fibonacci(n - 1) + fibonacci(n - 2);
    end
    begin
      i = 0;

      while (i < 10)
      do
      begin
        o = fibonacci(i);
        suspend;
        i = i + 1;
      end
  end
```

```
-- With forward declaration and parameter with default values

execute block returns (o integer)
as
    -- Forward declaration of P1.
    declare procedure p1(i integer = 1) returns (o integer);

    -- Forward declaration of P2.
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    declare procedure p2(i integer) returns (o integer);

    -- Implementation of P1 should not re-declare parameter default value.
    declare procedure p1(i integer) returns (o integer)
    as
    begin
        execute procedure p2(i) returning_values o;
    end

    declare procedure p2(i integer) returns (o integer)
    as
    begin
        o = i;
    end
begin
    execute procedure p1 returning_values o;
    suspend;
end
```

10.3 Функция RDB$ERROR

Возвращает значение контекста активного исключения.

Функция RDB$ERROR всегда возвращает NULL вне блока обработки ошибок WHEN ... DO.

```
RDB$ERROR (<контекст>)
<контекст> ::= { GDSCODE | SQLCODE | SQLSTATE | EXCEPTION | MESSAGE }
```

Доступные контексты в качестве аргумента функции RDB$ERROR:

- EXCEPTION - функция возвращает имя исключения, если активно исключение определённое пользователем, или NULL если активно одно из системных исключений. Для контекста EXCEPTION тип возвращаемого значения: VARCHAR(63) CHARACTER SET UTF8.
- MESSAGE - функция возвращает интерпретированный текст активного исключения. Для контекста MESSAGE тип возвращаемого значения: VARCHAR(1024) CHARACTER SET UTF8.
- GDSCODE - функция возвращает значение контекстной переменной GDSCODE.
- SQLCODE - функция возвращает значение контекстной переменной SQLCODE.
- SQLSTATE - функция возвращает значение контекстной переменной SQLSTATE.

Пример:

```
BEGIN
  ...
WHEN ANY DO
  EXECUTE PROCEDURE P_LOG_EXCEPTION(RDB$ERROR(MESSAGE));
END
```

10.4 Операторы управления в блоках PSQL

В предыдущих версиях операторы управления не допускались внутри блоков PSQL. Они были разрешены только как операторы SQL верхнего уровня или как оператор верхнего уровня в EXECUTE STATEMENT, встроенном в PSQL-блок.

Теперь их можно использовать непосредственно в блоках PSQL (триггерах, процедурах, EXECUTE BLOCK), что особенно полезно для приложений, в которых некоторые операторы управления должны выполняться в начале сеанса, в частности в триггерах ON CONNECT.

Можно использовать следующие операторы:

```
ALTER SESSION RESET
SET BIND
SET DECFLOAT ROUND
SET DECFLOAT TRAPS TO
SET ROLE
SET SESSION IDLE TIMEOUT
SET STATEMENT TIMEOUT
SET TIME ZONE
SET TRUSTED ROLE
```

Пример:

```
create or alter trigger on_connect on connect
as
begin
    set bind of decfloat to double precision;
    set time zone 'America/Sao_Paulo';
end
```

Глава 11

Мониторинг и утилиты командной строки

11.1 Мониторинг

11.1.1 Новые таблицы мониторинга

RDB$KEYWORDS

Содержит информацию о ключевых словах сервера.

Таблица 11.1 — RDB$KEYWORDS

Идентификатор столбца	Тип данных	Описание
RDB$KEYWORD_NAME	VARCHAR(63)	Ключевое слово.
RDB$KEYWORD_RESERVED	BOOLEAN	Зарезервировано ли ключевое слово.

MON$COMPILED_STATEMENTS

Хранит информацию о скомпилированных запросах.

Таблица 11.2 — MON$COMPILED_STATEMENTS

Идентификатор столбца	Тип данных	Описание
MON$COMPILED_STATEMENT_ID	BIGINT	Идентификатор запроса.
MON$SQL_TEXT	BLOB	Текст запроса, если он доступен.
MON$EXPLAINED_PLAN	BLOB	Расширенный план запроса.
MON$OBJECT_NAME	CHAR(63)	Имя объекта PSQL.
MON$OBJECT_TYPE	SMALLINT	Тип объекта PSQL.
MON$PACKAGE_NAME	CHAR(63)	Имя пакета объекта PSQL.
MON$STAT_ID	INTEGER	Идентификатор статистики

11.1.2 Новые столбцы в таблицах мониторинга

В таблице MON$DATABASE:

- MON$CRYPT_STATE - Состояние шифрование БД:
  - 0 - база данных не зашифрована,
  - 1 - зашифрована,
  - 2 - в процессе шифрования
- MON$GUID - GUID базы данных.
- MON$FILE_ID - Идентификатор файла базы данных на уровне файловой системы.
- MON$NEXT_ATTACHMENT - Текущее значение Next attachment ID.
- MON$NEXT_STATEMENT - Текущее значение Next statement ID.

- MON$REPLICA_MODE - Режим реплики базы данных (не реплика = 0, реплика в режиме read-only = 1, реплика в режиме read-write = 2).

В таблице MON$ATTACHMENTS:

- MON$SESSION_TIMEZONE - Часовой пояс сессии.
- MON$IDLE_TIMEOUT - Таймаут простоя соединения уровня соединения. Содержит значение тайм-аута простоя уровня соединения, в секундах. Если таймаут не установлен — 0.
- MON$IDLE_TIMER - Время истечения таймера ожидания. Содержит NULL, если тайм-аут простоя соединения не установлен, или если таймер не запущен.
- MON$STATEMENT_TIMEOUT - Таймаут SQL запроса уровня соединения. Содержит значение таймаута, установленное на уровне соединения, в миллисекундах. Если таймаут не установлен - 0.
- MON$WIRE_COMPRESSED - Сжатие сетевого трафика (включено = 1, отключено = 0).
- MON$WIRE_ENCRYPTED - Шифрование сетевого трафика (включено = 1, отключено = 0).
- MON$WIRE_CRYPT_PLUGIN - Имя плагина шифрования сетевого трафика, используемого клиентом.

В таблице MON$STATEMENTS:

- MON$COMPILED_STATEMENT_ID - Идентификатор скомпилированного запроса.
- MON$STATEMENT_TIMEOUT - Таймаут SQL запроса уровня соединения. Содержит значение таймаута, установленное на уровне соединения, в миллисекундах. Если таймаут не установлен - 0.
- MON$STATEMENT_TIMER - Значение таймера для оператора.

В таблице MON$RECORD_STATS:

- MON$RECORD_IMGC - Количество записей, затронутых промежуточной сборкой мусора.

В таблице MON$CALL_STACK:

- MON$COMPILED_STATEMENT_ID - Идентификатор скомпилированного запроса

В таблице SEC$GLOBAL_AUTH_MAPPING:

- SEC$DESCRIPTION - Текстовое описание

11.2 NBACKUP

11.2.1 Инкрементное резервное копирование

Утилита nBackup может выполнять физическое резервное копирование, используя GUID (UUID) самой последней резервной копии резервной базы данных, доступной только для чтения, для создания целевого файла резервной копии. Изменения из исходной базы данных могут непрерывно применяться к резервной базе данных, что избавляет от необходимости сохранять и применять все изменения с момента последнего полного резервного копирования.

Новый вариант резервного копирования можно запустить, не затрагивая существующую схему многоуровневого резервного копирования в живой базе данных.

Создание резервной копии

Синтаксис создания инкрементной резервной копии:

```
nbackup -B[ACKUP] <уровень> | <GUID> <исходная база данных> [<файл резервной копии>]
```

Применение изменений

Синтаксис "восстановления на месте" для слияния инкрементного файла резервной копии с резервной базой данных:

```
nbackup -I[NPLACE] -R[ESTORE] <база данных> <файл резервной копии>
```

Пример резервного копирования и восстановления

1. Используйте gstat, чтобы получить UUID базы данных:

```
gstat -h <база данных>
...
  Variable header data:
      Database backup GUID: {8C519E3A-FC64-4414-72A8-1B456C91D82C}
```

1. Используйте UUID для создания инкрементной резервной копии:

```
nbackup -B {8C519E3A-FC64-4414-72A8-1B456C91D82C} <база данных> <файл
резервной копии>
```

1. Примените изменения к базе данных:

```
nbackup -I -R <база данных> <файл резервной копии>
```

11.2.2 Восстановление и исправление реплики базы данных

Для команд -restore и -fixup добавлена новая опция командной строки -sequence. Сохраняет существующий GUID и последовательность репликации исходной базы данных (в противном случае они сбрасываются). Эта опция должна использоваться при создании реплики, чтобы асинхронная репликация могла быть автоматически продолжена с того момента, когда было выполнено физическое резервное копирование на основной базе.

Синтаксис выглядит следующим образом:

```
nbackup -R[ESTORE] <база данных> <файл резервной копии> -SEQ[UENCE]
nbackup -F[IXUP] <база данных> -SEQ[UENCE]
```

11.3 ISQL

11.3.1 Поддержка таймаута запросов

В isql появилась новая команда, позволяет установить таймаут выполнения запроса (в миллисекундах) для следующего оператора. После выполнения SQL запроса он автоматически сбрасывается в ноль.

```
SET LOCAL_TIMEOUT <значение>
```

11.3.2 Улучшенный контроль транзакций

Команда SET KEEP_TRAN_PARAMS [{ ON | OFF}] включает/выключает сохранение параметров транзакции. Если установлено значение ON, то будет сохранён полный текст SQL следующего успешного оператора SET TRANSACTION, и новые транзакции запускаются с использованием того же текста SQL (вместо режима CONCURRENCY WAIT по умолчанию). Если установлено значение OFF, то новые транзакции будут запускаться с параметрами по умолчанию. Имя KEEP_TRAN можно использовать в качестве сокращения KEEP_TRAN_PARAMS.

Примеры:

```
-- check current value
SQL> SET;
...
Keep transaction params: OFF

-- toggle value
SQL> SET KEEP_TRAN;
SQL> SET;
...
Keep transaction params: ON
SET TRANSACTION

SQL>commit;

-- start new transaction, check KEEP_TRAN value and actual transaction's parameters
SQL>SET TRANSACTION READ COMMITTED WAIT;
SQL>SET;
...
Keep transaction params: ON
  SET TRANSACTION READ COMMITTED WAIT
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT
============================================================
READ COMMITTED

SQL> commit;

-- start new transaction, ensure is have parameters as KEEP_TRAN value
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT
============================================================
READ COMMITTED

-- disable KEEP_TRAN, current transaction is not changed
SQL> SET KEEP_TRAN OFF;
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT
============================================================
READ COMMITTED
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
SQL> commit;

-- start new transaction, ensure is has default parameters (SNAPSHOT)
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT
============================================================
SNAPSHOT

SQL> SET;
...
Keep transaction params: OFF
```

11.3.3 SHOW SYSTEM

Команда SHOW SYSTEM в isql теперь перечисляет системные пакеты и их процедуры и функции.

11.3.4 Отображение BLR оператора

Команда SET EXEC_PATH_DISPLAY - это отладочная команда, показывающая BLR (скомпилированную форму) оператора. Когда оператор выполняется, он получает скомпилированный план выполнения оператора, отформатированный как текст BLR.

```
SET EXEC_PATH_DISPLAY {BLR | OFF};
```

Эта функция тесно связана с внутренним устройством сервера. Не рекомендуется использовать ее, если вы не очень хорошо понимаете, как эти внутренние компоненты могут меняться между версиями.

11.3.5 Информация о репликации добавлена в вывод SHOW DATABASE

Команда SHOW DATABASE теперь выводит информацию о режиме репликации и состоянии репликации.

11.4 GBAK

11.4.1 Резервное копирование и восстановление с шифрованием

При наличии зашифрованной базы данных рано или поздно возникнет необходимость в ее резервном копировании и восстановлении. Нет ничего плохого в том, чтобы резервная копия базы данных также была зашифрована. Если ключ шифрования передается подключаемому модулю каким-либо способом, не требующим ввода от клиентского приложения, это не является большой проблемой. Однако если сервер ожидает, что ключ будет передан со стороны клиента, это может стать проблемой.

Новые ключи в gbak обеспечивают решение проблемы.

Необходимые условия

Необходим подключаемый модуль для хранения ключей. Этот плагин может загружать ключи из какого-либо внешнего источника, например из файла конфигурации, и передавать их с помощью вызова.

```
ICryptKeyCallback* IKeyHolderPlugin::chainHandle(IStatus* status)
```

Новые ключи для зашифрованного резервного копирования и восстановления

При наличии необходимых условий можно использовать следующие новые ключи. Они не зависят от регистра.

- `-keyholder <имя плагина>` - Это основной ключ, необходимый `gbak` для доступа к зашифрованной базе данных.
- `-keyname <имя ключа>` - Явное присвоение имени ключу вместо ключа по умолчанию, указанного в исходной базе данных (при резервном копировании) или в файле резервной копии (при восстановлении).
- `-crypt <имя плагина>` - Имя плагина для шифрования файла резервной копии или восстановленной базы данных вместо плагина по умолчанию. Его также можно использовать в сочетании с ключом `-KEYNAME` для шифрования резервной копии незашифрованной базы данных или для шифрования базы данных, восстановленной из незашифрованной резервной копии.
- `-zip` - Сжать файл резервной копии перед его шифрованием. После шифрования файл почти не сжимается. Такой сжатый файл может быть распакован только восстановлением `gbak`. Ключ `-ZIP` не нужен для восстановления, потому что формат определяется автоматически.

Примеры

Пример 1

Чтобы сделать бэкап зашифрованной базы данных, выполните подобную команду:

```
gbak -b -keyholder MyKeyHolderPlugin host:dbname backup_file_name
```

Файл резервной копии будет зашифрован с использованием того же плагина и ключа шифрования, которые используются для шифрования базы данных. Это гарантирует, что украсть данные из файла резервной копии будет не легче, чем из базы данных.

Пример 2

Чтобы восстановить базу данных, которая была ранее зашифрована, выполните следующую команду:

```
gbak -c -keyholder MyKeyHolderPlugin backup_file_name host:dbname
```

Восстановленная база данных будет зашифрована с использованием того же плагина и ключа, что и файл резервной копии. В данном примере это тот же плагин и ключ, что и в исходной базе данных.

Пример 3

С помощью следующей команды можно:

- либо восстановить базу данных из файла резервной копии, созданного с использованием нестандартных плагинов шифрования `Crypt` и `Keyholder`, используя тот же `Keyname`, который использовался для резервного копирования;

- либо восстановить незашифрованную резервную копию как зашифрованную базу данных.

```
gbak -c -keyholder MyKeyHolderPlugin -crypt MyDbCryptPlugin -keyname SomeKey non_
encrypted_backup_file host:dbname
```

Пример 4

Чтобы сделать зашифрованную резервную копию незашифрованной базы данных, выполните:

```
gbak -b -keyholder MyKeyHolderPlugin -crypt MyDbCryptPlugin -keyname SomeKey
host:dbname encrypted_backup_file
```

Попытки создать незашифрованную резервную копию зашифрованной базы данных или восстановить зашифрованную резервную копию в незашифрованную базу данных потерпят неудачу. Такие операции намеренно запрещены.

Пример 5

Чтобы создать сжатую зашифрованную резервную копию, выполните:

```
gbak -b -keyholder MyKeyHolderPlugin -zip host:dbname backup_file_name
```

Файл резервной копии будет сжат до того, как он будет зашифрован с использованием того же плагина шифрования и того же ключа, который используется для шифрования базы данных.

11.4.2 Повышение производительности рестора

Новый `Batch API` используется для повышения производительности восстановления из резервной копии.

Изменены `"-fix_fss_*"` сообщения

В сообщениях, выводимых при восстановлении с помощью ключей `"-fix_fss_*"`, теперь используется слово `"adjusting"` вместо `"fixing"`.

11.4.3 Возможность выполнения бэкапа/рестора только выбранных таблиц

В `gbak` добавлен новый ключ: `-INCLUDE(_DATA)`. Аналогично существующему ключу `-SKIP_D(ATA)`, он принимает один параметр, который представляет собой шаблон регулярного выражения, используемый для сопоставления имен таблиц. Если он указан, то определяет таблицы для резервного копирования или восстановления. Синтаксис регулярных выражений, используемых для сопоставления имен таблиц, такой же, как и в логических выражениях `SIMILAR TO`. Взаимодействие между обоими переключателями описано в следующей таблице.

SKIP_DATA	INCLUDE_DATA		
	не установлено	соответствует таблице	не соответствует таблице
не установлено	включается	включается	исключается
соответствует таблице	исключается	исключается	исключается
не соответствует таблице	включается	включается	исключается

11.5 GSTAT

РЕД База Данных 5 поддерживает многопоточный сбор статистики. Для включения распараллеливания в `gstat` используйте новую опцию `-par <n>`:

```
gstat -par <n> <база_данных>
```

11.6 GFIX

11.6.1 Обновление минорной версии ODS

Новая опция `gfix -upgrade` обновляет базу данных до последней минорной версии `ODS` в пределах поддерживаемой мажорной версии.

> Обновление должно быть выполнено вручную, используя команду `gfix -upgrade`. Команда должна выполняться в эксклюзивном режиме. Для выполнения требуется системная привилегия `USE_GFIX_UTILITY`. Если обновление завершится с ошибками, то все изменения будут возвращены. После обновления РЕД База Данных 3 больше не сможет открыть базу данных.

```
gfix -upgrade <база данных> -user <имя пользователя> -pass <пароль>
```

11.6.2 Настройка и управление репликацией

В `gfix` добавлен новый ключ `-replica` для настройки и управления репликацией. Он принимает один из трех аргументов:

- `read_only` - Устанавливает копию базы данных как реплику, доступную только для чтения.
- `read_write` - Устанавливает копию базы данных как реплику, доступную для чтения и записи.
- `none` - Отключает репликацию и переводит реплику в режим `read_write`. Как правило, этот режим должен использоваться после сбоя ведущей базе данных, когда эта реплика переходит в статус ведущего. Или если нужно сделать физические резервные копии из реплики.

11.7 Агрегатный аудит

Агрегатный аудит собирает статистику по событиям. Метриками являются следующие значения запросов: `read`, `write`, `fetch`, `mark` и время выполнения. Аудит хранит значения метрик во время работы сервера. При выключении/перезагрузке сервера сохранённая статистика будет очищена.

Чтобы использовать агрегатный аудит, в файле конфигурации `firebird.conf` для параметра `TracePlugin` укажите значение `aggtrace`.

```
TracePlugin = aggtrace
```

Запросить значения, собранные агрегатным аудитом, можно следующей командой:

```
./rdbsvcmgr -service_mgr -user <имя пользователя> -password <пароль> -action_get_
aggtrace [опции]
```

Набор всех возможных опций представлен ниже.

Таблица 11.3 — Опции агрегатного трейса

Опция	Описание
-atrc_get	Запросить метрики новых и обновлённых запросов
-atrc_get_new	Запросить метрики только новых запросов
-atrc_get_old	Запросить метрики только старых запросов. Которые уже запрашивались ранее, но значения метрик не изменились
-atrc_get_all	Запросить метрики всех запросов
-atrc_with_query	Добавлять текст запроса в вывод
-atrc_clear	Очистить сохранённую статистику
-atrc_get_updated	Запросить метрики только обновлённых запросов т.е тех, у которых изменились значения метрик
-atrc_with_plan	Добавлять план запроса в вывод

Собранные значения метрик будут выведены в формате JSON:

```
[
  {
    "event": "<событие>",
    "hash": "<хэш>",
    "status": "<статус>",
    "perf": [
      [
        <среднее количество страниц, считанных из страничного кэша (fetch)>,
        <минимальное количество страниц, считанных из страничного кэша (fetch)>,
        <максимальное количество страниц, считанных из страничного кэша (fetch)>
      ],
      [
        <среднее количество прочитанных (read) страниц базы данных>,
        <минимальное количество прочитанных (read) страниц базы данных>,
        <максимальное количество прочитанных (read) страниц базы данных>
      ],
      [
        <среднее количество страниц, изменённых в страничном кэше (mark)>,
        <минимальное количество страниц, изменённых в страничном кэше (mark)>,
        <максимальное количество страниц, изменённых в страничном кэше (mark)>
      ],
      [
        <среднее количество страниц, записанных на диск (write)>,
        <минимальное количество страниц, записанных на диск (write)>,
        <максимальное количество страниц, записанных на диск (write)>
      ],
      [
        <общее время выполнения (ns)>,
        <среднее время выполнения (ns)>,
        <минимальное время выполнения (ns)>,
        <максимальное время выполнения (ns)>
      ]
    ],
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    "count": 1,
    "succeed": 1,
    "failed": 0,
    "dbname": "<путь к базе данных>",
    "query": "<текст запроса>",
    "plan": "<план запроса>"
  },
  ...
]
```

Описание значений в выводе:

- `event` – Название события. Отслеживаются следующие события:
  - Завершение выполнения хранимой процедуры (`FINISH PROCEDURE`);
  - Завершение выполнения триггера (`FINISH TRIGGER`);
  - Подготовка запроса (`PREPARE STATEMENT`);
  - Завершение выполнения запроса (`FINISH EXECUTE STATEMENT`);
  - Завершение выполнения хранимой функции (`FINISH FUNCTION`).
- `hash` – Хэш запроса;
- `status` – Статус запроса:
  - `new` – Запрос, метрики которого ранее не запрашивались из агрегатного трейса;
  - `same` – Старый запрос, у которого сохранённые значения мметрик не изменились;
  - `updated` – Обновлённый запрос, у которого обновились сохранённые значения метрик.
- `perf` – Собранные значения метрик;
- `count` – Количество выполнений запроса;
- `succeed` – Количество успешных выполнений;
- `failed` – Количество выполнений, завершённых с ошибкой;
- `dbname` – Путь к базе данных или алиас;
- `query` – Текст запроса;
- `plan` – План запроса.

Схема вывода результата работы `aggtrace.schema.json` расположена в каталоге `doc` установки сервера.

Глава 12

Устаревшие функции

12.1 Диалект 1

Диалект 1 объявлен устаревшим. Его поддержка будет прекращена в следующих версиях, и единственным поддерживаемым диалектом станет диалект 3. Рассмотрите возможность перехода на диалект 3 как можно скорее.

12.2 Внешние функции (UDF)

Поддержка внешних UDF функций (реализованных в динамических библиотеках и определенных в базе данных с помощью функций оператора DECLARE FUNCTION) в версии 5 устарела. Теперь по умолчанию для параметра UdfAccess в firebird.conf задано значение NONE. UDF-библиотеки ib_udf и fbudf изъяты из дистрибутива. Конечно, вы можете реализовать свою собственную динамическую библиотеку (или использовать библиотеки из предыдущей версии РЕД Базы Данных), настроить конфигурацию и работать с устаревшими функциями. Но этого делать не рекомендуется — вам лучше переключиться на использование функций UDR или PSQL. Чтобы переход стал проще, в дистрибутив включен скрипт udf_replace.sql, выполняющий соответствующую замену устаревших функций. К этим функциям относятся:

ADDDAY	ADDDAY2	ADDHOUR
ADDMILLISECOND	ADDMINUTE	ADDMONTH
ADDSECOND	ADDWEEK	ADDYEAR
DIV	DNULLIF	DNVL
DOW	DPOWER	GETEXACTTIMESTAMP
GETEXACTTIMESTAMPUTC	I64NULLIF	I64NVL
I64ROUND	I64TRUNCATE	INULLIF
INVL	ISLEAPYEAR	LTRIM
ROUND	RTRIM	SDOW
SNULLIF	SNVL	SRIGHT
STRING2BLOB	STRLEN	SUBSTR
SUBSTRLEN	TRUNCATE	UDF_FRAC или FRAC

При восстановлении базы данных с версии 3 на версию 5 может возникнуть ошибка, если в базе данных использовались устаревшие UDF функции:

```
gbak: WARNING:function XXX is not defined
gbak: WARNING: module name or entrypoint could not be found
```

Просто запустите скрипт:

```
isql -user sysdba -pas masterkey -i udf_replace.sql <база данных>
```

и UDF, распространяемые в ib_udf и fbudf, будут заменены соответствующими новыми функциями.

Если необходимо использовать UDF, то для параметра UdfAccess нужно указать значение Restrict <path-list>. По умолчанию <path-list> указывает на подкаталог UDF в корне РЕД Базы Данных. Строка в файле firebird.conf должна быть такой:

```
UdfAccess = Restrict UDF
```

12.3 AUTO ADMIN MAPPING

В настоящее время AUTO ADMIN MAPPING является устаревшим и поддерживается для обратной совместимости, вместо него рекомендуется использовать операторы {CREATE | ALTER | DROP} MAPPING.

12.4 Уровень изоляции READ COMMITTED [NO] RECORD VERSION

Существующие опции для READ COMMITTED, такие как RECORD VERSION и NO RECORD VERSION, всё ещё поддерживаются, но они устарени и будут удалены в будущих версиях.

Использование устаревших режимов READ COMMITTED (RECORD VERSION и NO RECORD VERSION) не рекомендуется, вместо них лучше использовать режим READ CONSISTENCY.

12.5 Утилита GSEC

Утилита GSEC устарела. Вместо неё лучше использовать SQL-команды для управления пользователями или Services API.

12.6 Опции GBAK

Таблица 12.2 — Устаревшие backup опции GBAK

Опция	Описание
-fa[ctor] n	Использовать блокирующий фактор n для ленточного накопителя. Опция устарела.
-ol[d_descriptions]	Производит резервное копирование метаданных в формате старого стиля, т.е. в режиме совместимости со старыми базами данных. Опция устарела.

Таблица 12.3 — Устаревшие restore опции GBAK

Опция	Описание
-k[ill]	Восстановить без создания теневых копий

Глава 13

Проблемы совместимости

13.1 Для транзакций READ COMMITTED по умолчанию используется Read Consistency

Read Consistency теперь является режимом по умолчанию для всех транзакций READ COMMITTED, независимо от параметров RECORD VERSION или NO RECORD VERSION. Это сделано для того, чтобы предоставить лучшее поведение - как соответствующее спецификации SQL, так и менее подверженное конфликтам. Если режим READ CONSISTENCY по каким-то причинам нежелателен, можно изменить конфигурационный параметр ReadConsistency, чтобы использовать старое поведение.

13.2 Изменения в DDL и DML для поддержки часовых поясов

13.2.1 Расширение типов данных TIMESTAMP и TIME

Синтаксис для объявления типов данных TIMESTAMP и TIME был расширен, чтобы включить аргументы, определяющие, должны ли столбец, домен, параметр или переменная быть определены с установкой часового пояса или без него:

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

По умолчанию используется WITHOUT TIME ZONE.

13.2.2 Изменения CURRENT_TIME и CURRENT_TIMESTAMP

В версии 5 переменные CURRENT_TIME и CURRENT_TIMESTAMP изменены: теперь они возвращают значения типа TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE с часовым поясом, установленным часовым поясом сеанса. В предыдущих версиях CURRENT_TIME и CURRENT_TIMESTAMP возвращали соответствующие типы в соответствии с системными часами, то есть без какого-либо часового пояса.

Переменные LOCALTIMESTAMP и LOCALTIME теперь заменяют прежние функциональные возможности CURRENT_TIMESTAMP и CURRENT_TIME соответственно.

13.3 Сокращенное преобразование неявных литералов даты/времени не поддерживается

Синтаксис сокращенного преобразования типов, используемый вместе с неявными литералами даты/времени, такой как, например:

```
TIMESTAMP 'NOW'
DATE 'TODAY'
DATE 'YESTERDAY'
```

мог привести к неожиданным результатам:

- в хранимых процедурах и функциях вычисление этих выражений будет происходить во время компиляции, а не во время вызова процедуры или функции, сохраняя результат в BLR и извлекая это устаревшее значение во время выполнения;
- в DSQL вычисление этих выражений происходит во время подготовки запросов, а не на каждой итерации оператора, как можно было бы ожидать при правильном использовании неявных литералов даты/времени. Разница во времени между подготовкой и выполнением оператора может быть слишком мала, чтобы обнаружить проблему. Пользователи могли быть введены в заблуждение, полагая, что выражение вычисляется на каждой итерации оператора во время выполнения, хотя на самом деле это происходит во время подготовки.

Если что-то вроде TIMESTAMP 'NOW' использовалось в SQL запросах в коде приложения или в PSQL, возникнет проблема совместимости с РЕД Базой Данных 5 . Теперь сервер будет выдавать ошибку.

В тоже время сокращенное преобразование явных литералов, таких как DATE '2019.02.20' допустимо. Также преобразование CAST('NOW' AS TIMESTAMP) продолжает работать как раньше.

13.4 Стартовое значение последовательности

До версии 5 последовательности (generator/sequence) создавались с текущим значением равным стартовому значению (или 0 по умолчанию).

Следующим значением последовательности со стартовым значением 0 и приращением 1 было 1. В версии 5 последовательности создаются с текущим значением равным стартовому минус инкремент. И начальным значением по умолчанию является 1 (а не 0).

То есть результат оператора NEXT VALUES FOR для последовательности со стартовым значением 100 и приращением 10 будет 100 (а не 110, как было раньше). Аналогично функция GEN_ID(SEQ, 1) возвратит 91 (а не 101, как было раньше).

13.5 Для INSERT ... RETURNING теперь требуется привилегия SELECT

Если оператор INSERT содержит предложение RETURNING, которое ссылается на столбцы основной таблицы, пользователю должна быть предоставлена соответствующая привилегия SELECT.

13.6 Ограничение количества символов UNICODE_FSS

Попытки хранить в столбце с набором символов UNICODE_FSS значения, превышающие заявленную длину, теперь будут завершаться ошибкой "string right truncation".

13.7 Многострочный вывод RETURNING

Клиентские запросы INSERT ... SELECT, UPDATE, DELETE, MERGE и UPDATE OR INSERT, содержащие предложение RETURNING, теперь могут возвращать несколько записей, а не вызывать ошибку "multiple rows in singleton select", как это было раньше.

Эти запросы теперь описываются как isc_info_sql_stmt_select при препарировании, в то время как в предыдущих версиях они описывались как isc_info_sql_stmt_exec_procedure.

Одиночные операторы INSERT ... VALUES, а также упорядоченные операторы UPDATE и DELETE (т. е. те, которые содержат предложение WHERE CURRENT OF) сохраняют существующее поведение, описываясь как isc_info_sql_stmt_exec_procedure. Они также сохраняют возможность выполнения в течение одного обращения протокола к серверу.

Однако все эти запросы, если они используются в PSQL и применяется предложение RETURNING, по-прежнему считаются одиночными.

13.8 Удалён протокол WNET

Сетевой протокол WNET, ранее используемый на Windows, удалён в РЕД Базе Данных 5. Пользователи Windows, работающие с любой строкой подключения WNET (\\server\dbname or wnet://server/dbname), должны перейти на протокол INET (TCP).

13.9 Оператор MERGE не допускает несколько совпадающих строк

В предыдущих версиях, когда строка в целевой таблице оператора MERGE совпадала с несколькими строками из исходного выражения запроса, эта строка обновлялась несколько раз (в недетерминированном порядке).

В соответствии с требованиями стандарта SQL, теперь при этом возникает ошибка "Multiple source records cannot match the same target during MERGE" (код ошибки 335545269/isc_merge_dup_update, SQLSTATE 21000).

13.10 Удалена QLI

Утилита командной строки QLI удалена в РЕД Базе Данных 5.