Оглавление

Новые возможности DML 4
Оконные (аналитические функции) 4
Агрегатные функции, используемые как оконные функции 4
Секционирование 5
Сортировка 6
Эксклюзивные оконные функции 7
Ранжирующие функции 7
Навигационные функции 8
Статистические функции 11
Функции линейной регрессии 12
Обратные гиперболические функции 14
OFFSET … FETCH 14
Дополнительные возможности оператора MERGE 15
Поддержка предложения DELETE 16
Поддержка множества предложений WHEN MATCHED | WHEN NOT MATCHED с дополнительными условиями 17
Поддержка предложения RETURNING … INTO 17
Регулярные выражения в SUBSTRING 18
Алиасы в предложении RETURNING 19
Поддержка предложения RETURNING для позиционных операторов UPDATE и DELETE 20
Альтернативы для внедрённых кавычек в строковых литералах 20
Запрет смешивания явных и неявных JOIN 21
Стабильность курсора 22
Внутреннее соединение с хранимыми процедурами 23
Новое в DDL 25
Тип BOOLEAN 25
Оператор IS 25
Тип столбца IDENTITY 27
Детали реализации 28
Управление допустимостью NULL значений для доменов и столбцов 28
Генераторы (последовательности) 30
Шаг приращения генератора (последовательности) 30
Изменение набора символов по умолчанию 31
Предложение ROLE в операторе CREATE DATABASE 32
«Задержка» закрытия базы данных для архитектуры SuperServer 33

Расширение оператора DROP SHADOW 34
Расширение оператора COMMENT 34
Новое в PSQL 37
PSQL функции 37
CREATE FUNCTION 37
ALTER FUNCTION 43
CREATE OR ALTER FUNCTION 44
DROP FUNCTION 45
RECREATE FUNCTION 46
Внешние процедуры 47
Внешние триггеры 49
Пакеты 51
DDL триггеры 59
Безопасность 62
Поддержка в утилитах 62
Пространства имён контекстных переменных DDL_TRIGGER 62
Переменные в пространстве имён DDL_TRIGGER 62
Примеры использования DDL триггеров 63
PSQL подпрограммы 66
Расширение использования префикса двоеточия 68
Ссылки на PSQL курсоры как на переменные 69
Двунаправленные (прокручиваемые) курсоры 72
Объявление двунаправленного курсора 72
Оператор FETCH 73
Исключения с параметрами 75
Оператор CONTINUE 77
SQLSTATE в обработчиках исключений 77
Стабильность PSQL курсоров 78
Удалены некоторые ограничения на размеры при использовании нового API 79
Новое в безопасности 80
Размещение списка пользователей 80
Управление пользователями с помощью SQL 80
CREATE USER 81
ALTER USER 83
CREATE OR ALTER USER 86
DROP USER 88
COMMENT ON USER 89

Псевдо таблицы со списком пользователей 89
SEC$USERS 90
SEC$USER_ATTRIBUTES 90
SET ROLE 90
SET TRUSTED ROLE 91
Новые права на объекты базы данных 92
Привилегия EXECUTE для функций и пакетов 92
Привилегии для защиты других объектов метаданных 94
Привилегии на изменение метаданных (DDL) 97
Управление отображением объектов безопасности 99
Правила отображения 100
Синтаксис для отображения объектов 101
Унаследованные правила отображения 103
Отображение пользователей Windows на CURRENT_USER 104
Шифрование базы данных 104
Секретный ключ 104
Задачи 104

Новые возможности DML

Оконные (аналитические функции)

Доступно: DSQL.

Согласно SQL спецификации оконные функции (также известные как аналитические функции) являются своего рода агрегатными функциями, не уменьшающими степень детализации. При этом агрегированные данные выводятся вместе с неагрегированными.

Синтаксически вызов оконной функции есть указание её имени, за которым всегда следует ключевое слово OVER() с возможными аргументами внутри скобок. В этом и заключается её синтаксическое отличие от обычной функции или агрегатной функции. Оконные функции могут находиться только в списке SELECT и предложении ORDER BY.

Предложение OVER может содержать указания выполнить действия с разбивкой по группам («секционирование») и сортировку.

Синтаксис:

```
<window function> ::=
<window function name>([<expr> [, <expr> ...]]) OVER (
  [PARTITION BY <expr> [, <expr> ...]]
  [ORDER BY <expr>
    [<direction>]
    [<nulls placement>]
    [, <expr> [<direction>] [<nulls placement>] ...]
)

<direction> ::= {ASC | DESC}

<nulls placement> ::= NULLS {FIRST | LAST}
```

Агрегатные функции, используемые как оконные функции

Все агрегатные функции могут быть использованы в качестве оконных функций, при добавлении предложения OVER.

Допустим, у нас есть таблица EMPLOYEE со столбцами ID, NAME и SALARY. Нам необходимо показать для каждого сотрудника, соответствующую ему заработную плату и процент от фонда заработной платы.

Простым запросом это решается следующим образом:

```
select
    id,
    department,
    salary,
```

```
    salary / (select sum(salary) from employee) percentage
from employee
order by id;
```

Результат

```
id department salary percentage
-- ---------- ------ ----------
1  R & D       10.00     0.2040
2  SALES       12.00     0.2448
3  SALES        8.00     0.1632
4  R & D        9.00     0.1836
5  R & D       10.00     0.2040
```

Запрос повторяется и может работать довольно долго, особенно если EMPLOYEE является сложным представлением.

Этот запрос может быть переписан в более быстрой и элегантной форме с использованием оконных функций:

```
select
  id,
  department,
  salary,
  salary / sum(salary) OVER () percentage
from employee
order by id;
```

Здесь **sum(salary) OVER ()** вычисляет сумму всех зарплат из запроса (таблицы сотрудников).

Секционирование

Как и для агрегатных функций, которые могут работать отдельно или по отношению к группе, оконные функции тоже могут работать для групп, которые называются “секциями” (partition).

Синтаксис:

```
<window function>(...) OVER (PARTITION BY <expr> [, <expr> ...])
```

Для каждой строки, оконная функция обсчитывает только строки, которые попадают в то же самую секцию, что и текущая строка.

Агрегирование над группой может давать более одной строки, таким образом, к результирующему набору, созданному секционированием, присоединяются результаты из основного запроса, используя тот же список выражений, что и для секции.

Продолжая пример с сотрудниками, вместо того чтобы считать процент зарплаты каждого сотрудника от суммарной зарплаты сотрудников, посчитаем процент от суммарной зарплаты сотрудников того же отдела:

```
select
  id,
  department,
  salary,
  salary / sum(salary) OVER (PARTITION BY department) percentage
from employee
order by id;
```

Результат

```
id department salary percentage
-- ---------- ------ ----------
1  R & D       10.00     0.3448
2  SALES       12.00     0.6000
3  SALES        8.00     0.4000
4  R & D        9.00     0.3103
5  R & D       10.00     0.3448
```

Сортировка

Предложение ORDER BY может быть использовано с секционированием или без него. Предложение ORDER BY внутри OVER задаёт порядок, в котором оконная функция будет обрабатывать строки. Этот порядок не обязан совпадать с порядком вывода строк. Для стандартных агрегатных функций, предложение ORDER BY заставляет возвращать частичные результаты агрегации по мере обработки записей.

Пример:

```
select
  id,
  salary,
sum(salary) over (order by salary) cumul_salary
from employee
order by salary;
```

Результат

```
id salary cumul_salary
-- ------ ------------
3    8.00         8.00
4    9.00        17.00
1   10.00        37.00
5   10.00        37.00
```

```
2    12.00         49.00
```

В этом случае cumul_salary возвращает частичную/накопительную агрегацию (функции SUM). Может показаться странным, что значение 37,00 повторяется для идентификаторов 1 и 5, но так и должно быть. Сортировка (ORDER BY) ключей группирует их вместе, и агрегат вычисляется единожды (но суммируя сразу два значения 10,00). Чтобы избежать этого, вы можете добавить поле ID в конце предложения ORDER BY.

Вы можете использовать несколько окон с различными сортировками, и дополнять предложение ORDER BY опциями ASC/DESC и NULLS FIRST/LAST.

С секциями предложение ORDER BY работает таким же образом, но на границе каждой секции агрегаты сбрасываются.

Все агрегатные функции могут использовать предложение ORDER BY, за исключением LIST().

Эксклюзивные оконные функции

Теперь рассмотрим эксклюзивные оконные функции (которые возможно использовать только с предложением OVER). В настоящее время они разделяются на две категории: функции ранжирования и навигационные функции. Оба типа функций могут применяться с использованием секционирования и сортировки и без них. Однако их использование без сортировки почти никогда не имеет смысла.

Ранжирующие функции

Ранжирующие функции вычисляют порядковый номер ранга внутри секции окна. Эта категория включает функции RANK, DENSE_RANK и ROW_NUMBER.

Синтаксис:

```
<ranking window function> ::=
    DENSE_RANK() |
    RANK() |
    ROW_NUMBER()
```

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),
  row_number() over (order by salary),
```

```
  sum(1) over (order by salary)
from employee
order by salary;
```

Результат

```
id salary dense_rank rank row_number sum
-- ------ ---------- ---- ---------- ---
3    8.00          1    1          1   1
4    9.00          2    2          2   2
1   10.00          3    3          3   4
5   10.00          3    3          4   4
2   12.00          4    5          5   5
```

Разница между функциями DENSE_RANK и RANK состоит в том, что для функции RANK существует разрыв, связанный с дублированием значения поля SALARY, по которому происходит сортировка. Функция DENSE_RANK продолжает присваивать последовательные номера, после дубликата зарплаты. С другой стороны, функция ROW_NUMBER назначает последовательные номера, даже когда есть повторяющиеся значения.

Навигационные функции

Навигационные функции получают простые (не агрегированные) значения выражения из другой строки запроса в той же секции.

Синтаксис:

```
<navigational window function> ::=
  FIRST_VALUE(<expr>) |
  LAST_VALUE(<expr>) |
  NTH_VALUE(<expr>, <offset>) [FROM FIRST | FROM LAST] |
  LAG(<expr> [ [, <offset> [, <default> ] ] ) |
  LEAD(<expr> [ [, <offset> [, <default> ] ] )
```

Важное замечание.

Функции FIRST_VALUE, LAST_VALUE и NTH_VALUE оперируют на кадрах окна. В настоящее время в Firebird кадры всегда определены с первой до текущей строки, но не последней, т.е.

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Из-за этого результаты функций NTH_VALUE и в особенности LAST_VALUE могут показаться странными.

Речь идёт о предложении кадрирования (фрейма). Для каждой строки, внутри её разбиения, существует список строк, который называется фрейм окна. Многие (но не все) оконные функции работают только со строками фрейма окна, а не со всем разбиением. По умолчанию, если задано предложение ORDER BY, то фрейм состоит из всех строк, от начала разбиения до текущей строки, плюс любые следующие строки, которые равны текущей строке в соответствии с предложением ORDER BY. Когда ORDER BY опускается, по умолчанию фрейм состоит из всех строк в разбиении.

В настоящее время предложение фрейма не реализовано см. **CORE-3647**. Предложение фрейма позволяет задать границы окна различными способами.

```
<window frame clause> ::=
  <window frame units> <window frame extent>
        [ <window frame exclusion> ]

<window frame units> ::= ROWS | RANGE

<window frame extent> ::=
      <window frame start>
    | <window frame between>

<window frame start> ::=
      UNBOUNDED PRECEDING
    | <window frame preceding>
    | CURRENT ROW

<window frame preceding> ::=
     <unsigned value specification> PRECEDING

<window frame between> ::=
    BETWEEN <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::=
    <window frame bound>

<window frame bound 2> ::=
    <window frame bound>

<window frame bound> ::=
      <window frame start>
    | UNBOUNDED FOLLOWING
    | <window frame following>

<window frame following> ::=
   <unsigned value specification> FOLLOWING

<window frame exclusion> ::=
     EXCLUDE CURRENT ROW
   | EXCLUDE GROUP
```

```
    | EXCLUDE TIES
    | EXCLUDE NO OTHERS
```

Пример:

```
select
  id,
  salary,
  first_value(salary) over (order by salary),
  last_value(salary) over (order by salary),
  nth_value(salary, 2) over (order by salary),
  lag(salary) over (order by salary),
  lead(salary) over (order by salary)
from employee
order by salary;
```

Результат

```
id salary first_value last_value nth_value    lag   lead
-- ------ ----------- ---------- ---------- ------ ------
3    8.00        8.00       8.00     <null> <null>   9.00
4    9.00        8.00       9.00       9.00   8.00  10.00
1   10.00        8.00      10.00       9.00   9.00  10.00
5   10.00        8.00      10.00       9.00  10.00  12.00
2   12.00        8.00      12.00       9.00  10.00 <null>
```

Функции FIRST_VALUE и LAST_VALUE возвращают, соответственно, первое и последнее значение в упорядоченной секции.

Функция NTH_VALUE возвращает N-ое значение, начиная с первой (опция FROM FIRST) или последней (опция FROM LAST) записи. По умолчанию используется опция FROM FIRST. Смещение 1 от первой записи будет эквивалентно функции FIRST_VALUE, смещение 1 от последней записи будет эквивалентно функции LAST_VALUE.

Функция LAG возвращает значение из предыдущей строки, LEAD – из следующей строки. Функции LEAD и LAG возвращают значение на дистанции смещения (offset), которое по умолчанию равно 1, относительно текущей строки.

Если смещение (offset) указывает за пределы секции, то будет возвращено значение default, которое по умолчанию равно NULL.

Примеры:

1. Предположим у вас есть таблица rate, которая хранит курс валюты на каждый день. Необходимо проследить динамику изменения курса за последние пять дней.

```
select
  bydate,
  cost,
  cost - lag(cost) over(order by bydate) as change,
  100 * (cost - lag(cost) over(order by bydate)) /
     lag(cost) over(order by bydate) as percent_change
from rate
where bydate between dateadd(-4 day to current_date)
                 and current_date
order by bydate
```

Результат

```
bydate         cost     change    percent_change
----------   -------   -------   --------------
27.10.2014    31.00     <null>           <null>
28.10.2014    31.53       0.53           1.7096
29.10.2014    31.40      -0.13          -0.4123
30.10.2014    31.67       0.27           0.8598
31.10.2014    32.00       0.33           1.0419
```

Статистические функции

Добавлены статистические функции для вычисления дисперсии и стандартного отклонения. Статистические функции как и агрегатные выполняют вычисление на наборе значений и возвращают одиночное значение. Эти функции не учитывают значения NULL. В отличии от агрегатных функций к аргументу функции не применимы параметры ALL и DISTINCT.

Статистические функции часто используются совместно с предложением GROUP BY. Любую из статистических функций можно использовать в качестве оконной.

- CORR – возвращает коэффициент корреляции для пары выражений, возвращающих числовые значения
- COVAR_POP – возвращает ковариацию совокупности (population covariance) пар выражений с числовыми значениями.
- COVAR_SAMP – возвращает выборочную ковариацию (sample covariance) пары выражений с числовыми значениями.
- VAR_POP – возвращает выборочную дисперсию
- VAR_SAMP – возвращает несмещённую выборочную дисперсию
- STDDEV_POP – возвращает среднеквадратическое отклонение
- STDDEV_SAMP – возвращает стандартное отклонение

CORR(<expr1>, <expr2>) эквивалентна

```
COVAR_POP(<expr1>, <expr2>) / (STDDEV_POP(<expr2>) * STDDEV_POP(<expr1>))
```

COVAR_POP(<expr1>, <expr2>) эквивалентна

```
(SUM(<expr1> * <expr2>) - SUM(<expr1>) * SUM(<expr2>) / COUNT(*))
  / COUNT(*)
```

COVAR_SAMP(<expr1>, <expr2>) эквивалентна

```
(SUM(<expr1> * <expr2>) - SUM(<expr1>) * SUM(<expr2>) / COUNT(*))
  / (COUNT(*) - 1)
```

VAR_POP(<expr>) эквивалентна

```
(SUM(<expr> ^ 2) - SUM(<expr>) ^ 2 / COUNT(<expr>)) / COUNT(<expr>)
```

VAR_SAMP(<expr>) эквивалентна

```
(SUM(<expr> ^ 2) - SUM(<expr>) ^ 2 / COUNT(<expr>)) / (COUNT(<expr>) - 1)
```

STDDEV_POP(<expr>) эквивалентна

```
SQRT(VAR_POP(<expr>))
```

STDDEV_SAMP(<expr>) эквивалентна

```
SQRT(VAR_SAMP(<expr))
```

Доступно: DSQL, PSQL.

Синтаксис:

```
<statistical function> ::=
  {{ VAR_POP | VAR_SAMP | STDDEV_POP | STDDEV_SAMP }(<expr>)}
  {{ CORR | COVAR_SAMP | COVAR_POP }(<expr1>, <expr2>)}
```

Примеры:

```
SELECT STDDEV_SAMP(salary) FROM employees;

SELECT CORR(salary, record_of_service) FROM employees;
```

Функции линейной регрессии

Функции линейной регрессии полезны для продолжения линии тренда. Линия тренда – это, как правило, закономерность, которой придерживается набор значений. Линия тренда полезна для прогнозирования будущих значений. Этот означает, что тренд будет продолжаться и в будущем. Для продолжения линии тренда необходимо знать угол наклона и точку пересечения с осью Y. Набор линейных функций включает функции для вычисления этих значений.

В синтаксисе функций, *y* интерпретируется в качестве переменной, зависящей от *x*.

Любую функции линейной регрессии из статистических функций можно использовать в качестве оконной.

```
<regr function> ::= <function name>(<expr1>, <expr2>)

<function name> := {
  REGR_AVGX | REGR_AVGY | REGR_COUNT | REGR_INTERCEPT |
  REGR_R2 | REGR_SLOPE | REGR_SXX | REGR_SXY | REGR_SYY }

где

Y = CASE
      WHEN <expr1> IS NOT NULL AND <expr2> IS NOT NULL
      THEN <expr1>
    END

X = CASE
      WHEN <expr1> IS NOT NULL AND <expr2> IS NOT NULL
      THEN <expr2>
    END

N = SUM(CASE WHEN x IS NOT NULL AND y IS NOT NULL THEN 1 END)

REGR_AVGX(Y, X) = SUM(X) / N

REGR_AVGY(Y, X) = SUM(Y) / N

REGR_COUNT(Y, X) = N

REGR_INTERCEPT(Y, X) = REGR_AVGY(Y, X) - REGR_SLOPE(Y, X) *
REGR_AVGX(Y, X)

REGR_R2(Y, X) = POWER(CORR(Y, X),2)

REGR_SLOPE(Y, X) = COVAR_POP(Y, X) / VAR_POP(X)

REGR_SXX(Y, X) = N * VAR_POP(X)

REGR_SXY(Y, X) = N * COVAR_POP(Y, X)

REGR_SYY(Y, X) = N * VAR_POP(Y)
```

Функция REGR_AVGX вычисляет среднее независимой переменной линии регрессии.

Функция REGR_AVGY вычисляет среднее зависимой переменной линии регрессии.

Функция REGR_COUNT возвращает количество непустых пар, используемых для создания линии регрессии.

Функция REGR_INTERCEPT вычисляет точку пересечения линии регрессии с осью Y.

Функция REGR_R2 вычисляет коэффициент детерминации, или R-квадрат, линии регрессии.

Функция REGR_SLOPE вычисляет угол наклона линии регрессии.

Обратные гиперболические функции

Добавлены обратные гиперболические функции ACOSH, ASINH, ATANH.

OFFSET … FETCH

Добавлен SQL:2008 совместимый эквивалент предложениям FIRST/SKIP и альтернатива предложению ROWS. Предложение OFFSET указывает, какое количество строк необходимо пропустить. Предложение FETCH указывает, какое количество строк необходимо получить.

Предложения OFFSET и FETCH могут применяться независимо уровня вложенности выражений запросов.

Доступно: DSQL, PSQL.

Синтаксис:

```
SELECT ...
[ORDER BY <expr_list>]
[OFFSET <simple_value_expr> {ROW | ROWS}]
[FETCH {FIRST | NEXT} [<simple_value_expr>] {ROW | ROWS} ONLY]

<simple_value_expr> ::= числовой литерал, SQL параметр (?) или
PSQL параметр (:param)
```

Примечания:

1. Firebird не поддерживает указание FETCH в процентах, определённое в стандарте.
2. Firebird не поддерживает FETCH … WITH TIES, определённое в стандарте.
3. FIRST…SKIP и ROWS являются нестандартными альтернативами.
4. Предложения OFFSET и/или FETCH не могут быть объединены с предложениями ROWS или FIRST/SKIP в одном выражении запроса.
5. Выражения, ссылки на столбцы и т.д. недопустимы в любом из предложений.
6. В отличие от предложения ROWS, предложения OFFSET и FETCH допустимы только в операторе SELECT.

Примеры:

1. Вернуть получить все строки кроме первых 10, упорядоченных по столбцу COL1:

```
SELECT *
FROM T1
ORDER BY COL1
OFFSET 10 ROWS
```

2. Вернуть первые 10 строк, упорядоченных по столбцу COL1:

```
SELECT *
FROM T1
ORDER BY COL1
FETCH FIRST 10 ROWS ONLY
```

3. Использование предложений OFFSET и FETCH в производной таблице, результат которой ограничивается ещё раз во внешнем запросе.

```
SELECT *
FROM (
  SELECT *
  FROM T1
  ORDER BY COL1 DESC
  OFFSET 1 ROW
  FETCH NEXT 10 ROWS ONLY
) a
ORDER BY a.COL1
FETCH FIRST ROW ONLY
```

Дополнительные возможности оператора MERGE

Доступно: DSQL, PSQL.

Оператор MERGE дополнен новыми возможностями из стандарта SQL-2008. В Firebird 3 добавлены следующие возможности:

- Поддержка предложения DELETE
- Поддержка множества предложений WHEN MATCHED | WHEN NOT MATCHED с дополнительными условиями
- Поддержка предложения RETURNING … INTO

Синтаксис:

```
MERGE
INTO <target> [AS target_alias]
USING <source> [AS source_alias]
ON <join condition>
<merge when> [<merge when> ...]
```

```
[RETURNING <returning list>
[INTO <variable list>]]

<merge when> ::= <merge when matched> | <merge when not matched>

<merge when matched> ::=
WHEN MATCHED [ AND <condition> ] THEN
{ UPDATE SET <assignment list> | DELETE }

<merge when not matched> ::=
WHEN NOT MATCHED [ AND <condition> ] THEN
INSERT [ (<column list>) ]
VALUES (<value list>)

<assignment list> ::= colname = value [, colname = value ...]

<column list> ::= colname [, colname ...]

<value list> ::= value [, value ...]

<variable list> ::= [:]var [, [:]var ...]

<target> ::= Таблица или обновляемое представление

<source> ::= Таблица, представление, хранимая процедура или
производная таблица
```

Поддержка предложения DELETE

В Firebird 3 в предложениях WHEN MATCHED разрешено не только обновлять записи целевой таблицы (обновляемого представления), но и удалять их, если вместо предложения UPDATE указано предложение DELETE.

Примеры:

Следующий запрос

```
DELETE FROM SALARY_HISTORY
WHERE EMP_NO IN (SELECT EMP_NO
                 FROM EMPLOYEE
                 WHERE DEPT_NO = 120)
```

можно изменить на эквивалентный ему запрос с использованием оператора MERGE. В ряде случаев его выполнение может быть более эффективным.

```
MERGE INTO SALARY_HISTORY
USING (SELECT EMP_NO
```

```
        FROM EMPLOYEE
        WHERE DEPT_NO = 120) EMP
    ON SALARY_HISTORY.EMP_NO = EMP.EMP_NO
    WHEN MATCHED THEN DELETE
```

Поддержка множества предложений WHEN MATCHED | WHEN NOT MATCHED с дополнительными условиями

В Firebird 3 разрешено указывать несколько предложений WHEN MATCHED и WHEN NOT MATCHED.

Предложения WHEN проверяются в указанном порядке. Если условие в предложении WHEN не выполняется, то мы пропускаем его и переходим к следующему предложению. Так будет происходить до тех пор, пока условие для одного из предложений WHEN не будет выполнено. В этом случае выполняется действие, связанное с предложением WHEN, и осуществляется переход на следующую строку источника. Для каждой строки источника выполняется только одно действие.

Примеры:

В следующем примере происходит ежедневное обновление таблицы PRODUCT_IVENTORY на основе заказов, обработанных в таблице SALES_ORDER_LINE. Если количество заказов на продукт таково, что уровень запасов продукта опускается до нуля или становится еще ниже, то строка этого продукта удаляется из таблицы PRODUCT_IVENTORY.

```
MERGE INTO PRODUCT_IVENTORY AS TARGET
USING (
  SELECT
    SL.ID_PRODUCT,
    SUM(SL.QUANTITY)
  FROM SALES_ORDER_LINE SL
  JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
  WHERE S.BYDATE = CURRENT_DATE
  GROUP BY 1
) AS SOURCE(ID_PRODUCT, QUANTITY)
ON  TARGET.ID_PRODUCT = SOURCE.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SOURCE.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SOURCE.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
```

Поддержка предложения RETURNING … INTO

Оператор MERGE, затрагивающий не более одной строки, может содержать конструкцию RETURNING для возвращения значений добавленной,

модифицируемой или удаляемой строки. В RETURNING могут быть указаны любые столбцы из целевой таблицы (обновляемого представления), не обязательно все, а также другие столбцы и выражения.

Возвращаемые значения содержат изменения, произведенные в триггерах BEFORE.

Имена столбцов могут быть уточнены с помощью префиксов NEW и OLD для определения, какое именно значение вы хотите столбца вы хотите получить до модификации или после.

Для предложений WHEN MATCHED UPDATE и MERGE WHEN NOT MATCHED неуточнённые имена столбцов или уточнённые именами таблиц или их псевдонимами понимаются как столбцы с префиксом NEW, для предложений MERGE WHEN MATCHED DELETE – с префиксом OLD.

Примеры:

Немного модифицируем наш предыдущий пример, чтобы он затрагивал только одну строку, и добавим в него инструкцию RETURNING возвращающего старое и новое количество товара и разницу между этими значениями.

```
…
MERGE INTO PRODUCT_IVENTORY AS TARGET
USING (
  SELECT
    SL.ID_PRODUCT,
    SUM(SL.QUANTITY)
  FROM SALES_ORDER_LINE SL
  JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
  WHERE S.BYDATE = CURRENT_DATE
    AND SL.ID_PRODUCT = :ID_PRODUCT
  GROUP BY 1
) AS SOURCE(ID_PRODUCT, QUANTITY)
ON  TARGET.ID_PRODUCT = SOURCE.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SOURCE.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SOURCE.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
RETURNING OLD.QUANTITY, NEW.QUANTITY, SOURCE.QUANTITY
INTO :OLD_QUANTITY, :NEW_QUANTITY, :DIFF_QUANTITY
…
```

Регулярные выражения в SUBSTRING

Функция SUBSTRING с регулярным выражением возвращает часть строки соответствующей шаблону в предложении SIMILAR. Если соответствия не найдено, то возвращается NULL.

Синтаксис:

```
<regular expression substring> ::=
  SUBSTRING(<value>
            FROM <similar pattern>
            ESCAPE <escape character>)

<similar pattern> ::= <substring similar expression>

<substring similar expression> ::=
  <similar pattern: R1><escape>"<similar pattern:
R2><escape>"<similar pattern: R3>
```

Примечания:

1. Если любая из частей (R1, R2 или R3) регулярного выражения не является пустой строкой и не соответствует формату <similar pattern>, будет возбуждено исключение.
2. Возвращаемое значение соответствует части R2 регулярного выражения. Для этого значения истинно выражение

   <value> SIMILAR TO R1 || R2 || R3 ESCAPE <escape>

Примеры:

```
substring('abcabc' similar 'a#"bcab#"c' escape '#')  -- bcab
substring('abcabc' similar 'a#"%#"c' escape '#')     -- bcab
substring('abcabc' similar '_#"%#"_' escape '#')     -- bcab
substring('abcabc' similar '#"(abc)*#"' escape '#')  -- abcabc
substring('abcabc' similar '#"abc#"' escape '#')     -- <null>
```

Алиасы в предложении RETURNING

Добавлена возможность использования псевдонимов в предложении RETURNING. Это может быть полезно при получении этих значений клиентом.

Примеры с использованием псевдонимов и без:

```
UPDATE T1 SET F2 = F2 * 10
RETURNING OLD.F2, NEW.F2; -- without aliases

UPDATE T1 SET F2 = F2 * 10
RETURNING OLD.F2 OLD_F2, NEW.F2 AS NEW_F2; -- with aliases
```

Примечание.

Ключевое слово AS не является обязательным.

Поддержка предложения RETURNING для позиционных операторов UPDATE и DELETE

Добавлена поддержка предложения RETURNING для позиционных операторов UPDATE и DELETE.

Примеры:

```
UPDATE T1
SET F2 = F2 * 10
WHERE CURRENT OF C
RETURNING NEW.F2;
```

Альтернативы для внедрённых кавычек в строковых литералах

Теперь вместо двойного (экранированного) апострофа вы можете использовать другой символ или пару символов, для внедрённой строки в кавычках внутри другой строки. Ключевое слово q или Q предшествующее строке в кавычках сообщает парсеру, что некоторые левые и правые пары одинаковых символов являются разделителями для встроенного строкового литерала.

Синтаксис:

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

Правила.

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

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

Примеры:

```
-- result: abc{def}ghi
select q'{abc{def}ghi}' from rdb$database;

-- result: That's a string
select q'!That's a string!' from rdb$database;
```

Динамическая сборка запроса использующего строковые литералы.

```
EXECUTE BLOCK
RETURNS (
  RDB$TRIGGER_NAME CHAR(31)
)
AS
  DECLARE VARIABLE S VARCHAR(8191);
BEGIN
  S = 'SELECT RDB$TRIGGER_NAME FROM RDB$TRIGGERS WHERE
RDB$RELATION_NAME IN ';
  S = S || Q'! ('SALES_ORDER', 'SALES_ORDER_LINE')!';
  FOR
    EXECUTE STATEMENT :S
    INTO :RDB$TRIGGER_NAME
  DO
    SUSPEND;
END
```

Запрет смешивания явных и неявных JOIN

Смешивание явного и неявного синтаксиса JOIN не было рекомендовано никогда, но позволялось парсером. Некоторые «смешивания» приводили оптимизатор к непредсказуемым результатам, включая ошибку «no current record for fetch operation». Некоторые наиболее стили смешивания запрещены в других SQL движках, теперь они запрещены и в Firebird. Дело в том что:

(TA, TB JOIN TC) эквивалентно TA, (TB JOIN C) и не равно (TA, TB) JOIN C

где каждая часть разделённая запятыми есть неявная производная таблица.

Примечание.

Это изменение может нарушить работоспособность вашего приложения. Перед переводом вашей системы на Firebird 3 тщательно просмотрите используемые в ней запросы на наличии таких случаев.

Примеры:

Такой запрос вызовет ошибку «Column does not belong to referenced table».

```
SELECT *
FROM
    TA, TB
    JOIN TC ON TA.COL1 = TC.COL1
WHERE TA.COL2 = TB.COL2
```

Это происходит потому, что явный JOIN не может видеть таблицу TA. Однако следующий запрос будет выполнен без ошибок, поскольку изоляция не нарушена.

```
SELECT *
FROM
    TA, TB
    JOIN TC ON TB.COL1 = TC.COL1
WHERE TA.COL2 = TB.COL2
```

Стабильность курсора

Предыдущие релизы Firebird пострадали от печально известной ошибки, в результате которой операции модификации данных могли входить в бесконечный цикл и в зависимости от операции удалить все строки таблицы, обновлять строки до бесконечности или вставлять новые строки до тех пор, пока у хост машины не закончатся ресурсы. Все DML операторы (INSERT, UPDATE, DELETE, MERGE) пострадали от неё. Это произошло потому, что движок использует неявный курсор для этих операций.

Для обеспечения стабильности, вставленные, обновлённые или удалённые строки необходимо отметить каким-либо образом, чтобы избежать повторных визитов. Другое решение добавление в план запроса внешней сортировки (SORT), что бы заставить курсор материализоваться.

Начиная с Firebird 3, движок использует Undo Log для того, чтобы определить была ли строка уже вставлена или модифицирована в текущем курсоре.

Важно.

Стабилизация не работает, если в PSQL цикле используется оператор SUSPEND.

Примеры:

1. До Firebird 3, такой запрос мог привести к бесконечному циклу, и вставлять записи до тех пор, пока не будут исчерпаны ресурсы хост машины. Начиная с Firebird 3, в таблицу T буду вставлены только те записи, которые в ней существуют на момент старта запроса.

   ```
   INSERT INTO T(col)
   SELECT col FROM T
   ```

2. До Firebird 3, такой запрос удалит все записи в таблице, вместо первых пяти.

   ```
   delete from T
   where col in (select col first 5 from T);
   ```

3. Следующий пример демонстрирует различное поведение оператора merge в Firebird 3 и более ранних версиях.

```
-- таблица-источник:
recreate table tu_src(
  id int primary key using index tu_src_pk,
  x int);
insert into tu_src values(1, 10);
insert into tu_src values(2, 20);
insert into tu_src values(3, 10);
commit;
-- таблица-приёмник:
recreate table tu_tgt(
  x int primary key using index tu_tgt_pk,
  cnt int);
commit;
```

Теперь пробуем записать в таблицу-приёмник значения ‘x’ из источника и, например, количество раз, которое они там встречаются.

```
merge into tu_tgt t
using tu_src s
on t.x=s.x
when not matched then insert values(s.x, 1)
when matched then update set t.cnt = t.cnt + 1;
```

Результат в ФБ 3.0:

```
Invalid insert or update value(s): object columns are
constrained - no 2 table rows can have duplicate column
values.
violation of PRIMARY or UNIQUE KEY constraint "INTEG_22" on
table "TU_TGT".
```

Результат в ФБ 2.5: будут обновлены все три строки, исключения не возникнет.

Внутреннее соединение с хранимыми процедурами

До Firebird 3 оптимизатор не мог определить зависимость входных параметров процедуры от прочих потоков. Это приводило к ошибке isc_no_cur_rec (no current record for fetch operation) в том случае, если хранимая процедура, коррелированная с другими потоками через входные параметры, соединилась с этими потоками внутренним соединением.

Например, такой запрос

```
SELECT *
FROM MY_TAB
JOIN MY_PROC(MY_TAB.F) ON 1 = 1
```

приводил к ошибке *no current record for fetch operation* в предыдущих версиях Firebird. Теперь такие запросы допустимы.

Ранее эта проблема решалась заменой внутреннего соединения на внешнее, таким образом, неявно указывался порядок соединения потоков.

```
SELECT *
FROM MY_TAB
LEFT JOIN MY_PROC(MY_TAB.F) ON 1 = 1
```

Новое в DDL

Тип BOOLEAN

Доступно: DSQL, PSQL.

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

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

Оператор IS

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

> **Примечание**.
>
> Операторы эквивалентности («=», «!=», «<>» и др.) допустимы во всех сравнениях.

Примеры:

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

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

SELECT * FROM TBOOL

ID           BVAL
============ =======
1             <true>
2            <false>
```

```
3             <null>

-- Проверка TRUE значения
SELECT * FROM TBOOL WHERE BVAL

ID           BVAL
============ =======
1             <true>

-- Проверка FALSE значения
SELECT * FROM TBOOL WHERE BVAL IS FALSE

ID           BVAL
============ =======
2            <false>

-- Проверка UNKNOWN значения
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN

ID           BVAL
============ =======
3             <null>

-- Логические значения в SELECT списке
SELECT ID, BVAL, BVAL AND ID < 2
FROM TBOOL

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

-- PSQL объявления с начальным значением
DECLARE VARIABLE VAR1 BOOLEAN = TRUE;

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

Примечания:

- Представлен в API типом FB_BOOLEAN и константами FB_TRUE и FB_FALSE.
- Значение TRUE больше чем FALSE.
- Несмотря на то, что тип данных BOOLEAN не преобразуется неявно ни к одному типу, возможно явное преобразование к строке с помощью функции CAST.
- Из соображений совместимости незарезервированные ключевые слова INSERTING, UPDATING и DELETING продолжают вести себя как логические

выражения в контексте PSQL, однако ведут себя как значения, если они является именами столбцов или переменных в нелогических выражениях.

Рассмотрим пример использования слова INSERTING в трёх различных случаях:

```
SELECT
    INSERTING, -- value
    NOT INSERTING -- keyword
FROM TEST
WHERE
      INSERTING -- keyword
  AND INSERTING IS TRUE -- value
```

Тип столбца IDENTITY

Столбец идентификации представляет собой столбец, связанный с внутренним генератором последовательностей. Его значение устанавливается автоматически каждый раз, когда оно не указано в операторе INSERT.

Синтаксис:

```
<column definition> ::=
  <name> <type> GENERATED BY DEFAULT AS IDENTITY
  [ (START WITH <value>) ] <constraints>
```

При определении столбца, необязательное предложение START WITH позволяет указать начальное значение отличное от нуля.

```
<alter column definition> ::=
  <name> RESTART [ WITH <value> ]
```

В определении столбца может быть изменено начальное значение генератора. Если указано только предложение RESTART, то происходит сброс значение генератора в ноль. Необязательное предложение WITH позволяет указать для нового значения внутреннего генератора отличное от нуля значение.

Правила:

- Тип данных столбца идентификации должен быть целым числом с нулевым масштабом. Допустимыми типами являются SMALLINT, INTEGER, BIGINT, NUMERIC(x,0) и DECIMAL(x,0).
- Идентификационный столбец не может иметь DEFAULT и COMPUTED значений.

Примечания.

- Идентификационный столбец не может быть изменён, чтобы стать обычным столбцом. Обратное тоже верно.
- Идентификационные столбцы неявно являются NOT NULL столбцами.
- Уникальность не обеспечивается автоматически. Ограничения UNIQUE или PRIMARY KEY требуются для гарантии уникальности.

Примеры:

```
create table objects (
  id integer generated by default as identity primary key,
  name varchar(15)
);

insert into objects (name) values ('Table');
insert into objects (name) values ('Book');
insert into objects (id, name) values (10, 'Computer');

select * from objects;

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

Детали реализации

Два новых столбца добавлены в таблицу RDB$RELATION_FIELDS для поддержки столбцов идентификации: RDB$GENERATOR_NAME и RDB$IDENTITY_TYPE.

- Столбец RDB$GENERATOR_NAME хранит имя автоматически созданного генератора для столбца. В таблице RDB$GENERATORS значение поля RDB$SYSTEM_FLAG для этого генератора содержит значение 6.
- В настоящее время столбец RDB$IDENTITY_TYPE всегда хранит значение 0 (GENERATED BY DEFAULT) для столбца идентификации и NULL для других типов столбцов. В будущем этот столбец будет иметь возможность хранить значение 1 (GENERATED ALWAYS), когда этот тип идентификационных столбцов будет поддерживаться Firebird.

Управление допустимостью NULL значений для доменов и столбцов

Синтаксис ALTER теперь поддерживает изменение допустимости NULL значений для столбцов таблиц или доменов.

Синтаксис:

```
ALTER TABLE <table name> ALTER <field name> [NOT] NULL

ALTER DOMAIN <domain name> [NOT] NULL
```

Примечания.

Успешное изменение столбца таблицы с NULL в NOT NULL, только после полной проверки данных таблицы, для того чтобы убедится что столбец не содержит значений NULL.

При изменении домена все таблицы его использующие подвергаются такой проверке.

Явное ограничение NOT NULL на столбце, базирующегося на домене, преобладает над установками домена. В этом случае изменение домена для допустимости значения NULL, не распространяется на столбец таблицы.

При добавлении нового столбца, не допускающего значения NULL, в таблицу с данными необходимо также установить значение по умолчанию. Дело в том, что в этом случае также происходит проверка данных на допустимость. А поскольку при добавлении нового столбца, он для всех строк таблицы содержит значение NULL, будет сгенерировано исключение.

Примечания.

В прежних версиях отсутствие такой проверки приводило к появлению так называемых «невосстановимых» резервных копий, созданных утилитой gbak.

Примеры:

```
create table t1 (
  id int primary key,
  a int);

insert into t1(id, a) values(1, 1);
commit;

alter table t1 add b int not null; -- не будет выполнено

Statement failed, SQLSTATE = 22006
unsuccessful metadata update
-Cannot make field B of table T1 NOT NULL because there are
NULLs present
```

```
alter table t1 add b int default 0 not null; -- успешно
выполнено
```

Генераторы (последовательности)

Зарезервированные слова GENERATOR и SEQUENCE теперь являются полными синонимами. Там где раньше возможно было использовать ALTER SEQUENCE <sequence name> RESTART WITH <value>, теперь вместо слова SEQUENCE допускается слово GENERATOR.

Предложение RESTART может быть использовано самостоятельно для перезапуска значения последовательности с того значения с которого был начат старт генерации значений или предыдущий рестарт. Для хранения этого значения добавлен столбец RDB$INITIAL_VALUE в системную таблицу RDB$GENERATORS.

Необязательное предложение INCREMENT [BY] позволяет задать шаг приращения для оператора NEXT VALUES FOR <sequence name>. Для хранения шага приращения был добавлен столбец RDB$GENERATOR_INCREMENT в системную таблицу RDB$GENERATORS.

Синтаксис:

```
{ CREATE | RECREATE } { SEQUENCE | GENERATOR } <sequence name>
[ START WITH <value> ] [ INCREMENT [BY] <increment> ]

CREATE OR ALTER { SEQUENCE | GENERATOR } <sequence name> {
RESTART | START WITH <value> } [ INCREMENT [BY] <increment> ]

ALTER { SEQUENCE | GENERATOR } <sequence name>
RESTART [ WITH <value> ] [ INCREMENT [BY] <increment> ]
```

Шаг приращения генератора (последовательности)

Шаг приращения позволяет работать с генераторами согласно SQL стандарту с помощью конструкции

```
NEXT VALUE FOR <sequence name>

Аналогичной

GEN_ID(<sequence_name>, <increment>),

<increment> ::= значение инкремента указанное при создании
генератора
```

По умолчанию приращение равняется единице для пользовательских генераторов и нулю для системных. Это делает возможным применение NEXT

VALUE FOR <sys generator>, поскольку теперь вы не можете изменять значения системных генераторов пользовательскими запросами.

Примеры:

```
-- Стартуют с нуля и изменяется с шагом 10
create sequence seq increment 10;

select next value for seq from rdb$database; -- Результат 10

-- Результат 11, gen_id() не зависит от приращения.
select gen_id(seq, 1) from rdb$database;
```

Если база данных новая и не имеет хранимых процедур, то

```
select next value for rdb$procedures from rdb$database;
```

будет давать каждый раз ноль, потому что это системный генератор.

Приращение не может быть установлено в ноль для пользовательских генераторов.

Пример:

```
create generator g00 increment 0;

Statement failed, SQLSTATE = 42000
unsuccessful metadata update
-CREATE SEQUENCE G00 failed
-INCREMENT 0 is an illegal option for sequence G00
```

Примечания.

Изменение значения приращения – это возможность, которая вступает в силу для каждого запроса, который запускается после фиксаций изменения. Процедуры, которые вызваны впервые после изменения приращения, будут использовать новое значение, если они будут содержать операторы NEXT VALUE FOR. Процедуры, которые уже работают, не будут затронуты, потому что они кэшируются. Процедуры, использующие NEXT VALUE FOR, не должны быть перекомпилированы, чтобы видеть новое приращение, но если они уже работают или загружены, то никакого эффекта не будет. Конечно процедуры, использующие gen_id(gen, expression), не затронут при изменении приращения.

Изменение набора символов по умолчанию

Изменение набора символов по умолчанию для базы данных.

Синтаксис:

```
ALTER DATABASE
...
SET CHARACTER SET <new_charset>
```

Это изменение не затрагивает существующие данные. Новый набор символов по умолчанию будет использоваться только в последующих DDL командах, кроме того для них будет использоваться сортировка по умолчанию для нового набора символов.

Предложение ROLE в операторе CREATE DATABASE

В Firebird 3.0 установка роли при создании базы данных может повлиять на права пользователей для создания базы данных. Это может произойти, если пользователь получает (в соответствующей базе данных безопасности) системную роль RDB$ADMIN или какую либо обычную роль, которой предоставлены права CREATE DATABASE.

ISQL теперь устанавливает в качестве глобальной роли роль, указанную в операторе CREATE DATABASE.

Синтаксис:

```
CREATE {DATABASE | SCHEMA} '<filespec>'
[USER 'username' [PASSWORD 'password'] [ROLE 'rolename']]
[PAGE_SIZE [=] size]
[LENGTH [=] num [PAGE[S]]
[SET NAMES charset]
[DEFAULT CHARACTER SET default_charset
  [COLLATION collation]]
[<sec_file> [<sec_file> ...]]
[DIFFERENCE FILE 'diff_file'];

<filespec> ::= [<server_spec>]{filepath | db_alias}

<server_spec> ::= servername: | \\servername\

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

Пример:

create user john **password** 'superman' **grant admin role**;

create database 'localhost:D:\test.fdb'
user 'john' **password** 'superman' **role** 'rdb$admin';

В данном случае, если предложение ROLE не будет указано, то база данных не будет создана.

«Задержка» закрытия базы данных для архитектуры SuperServer

Иногда желательно иметь механизм для ядра SuperServer, сохраняющего базу данных в открытом состоянии в течение некоторого времени после того как последние соединение закрыто, т.е. иметь механизм задержки закрытия базы данных. Это может помочь улучшить производительность и уменьшить издержки в случаях, когда база данных часто открывается и закрывается, сохраняя при этом ресурсы «разогретыми» до следующего открытия.

Примечания.

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

Firebird 3.0 представляет улучшение команды ALTER DATABASE для управления дополнительной возможностью LINGER (задержки) для баз данных работающих под управлением SuperServer.

Синтаксис:

```
ALTER DATABASE SET LINGER TO {seconds};

ALTER DATABASE DROP LINGER;
```

Использование:

Чтобы установить задержку для базы данных выполните:

```
-- установка интервала задержки 30 секунд
ALTER DATABASE SET LINGER TO 30;
```

Любая из следующих форм очистит установки задержки и возвратит базу данных к нормальному состоянию (без задержки):

```
ALTER DATABASE DROP LINGER;

ALTER DATABASE SET LINGER TO 0;
```

Примечания.

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

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

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

Расширение оператора DROP SHADOW

Доступно: DSQL

Синтаксис:

```
DROP SHADOW number [{PRESERVE | DELETE} FILE];
```

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

Расширение оператора COMMENT

Оператор COMMENT ON был расширен для поддержания новых типов объектов базы данных PSQL функций и пакетов. Кроме того, появилась возможность добавить комментарии для учётных записей.

Доступно: DSQL

Синтаксис:

```
COMMENT ON <object> IS {'sometext' | NULL}
<object> ::=
  DATABASE |
  <basic-type> objectname |
  COLUMN relationname.fieldname |
  [PROCEDURE | FUNCTION] PARAMETER
    [package_name.] routinename.paramname |
  {PROCEDURE | [EXTERNAL] FUNCTION}
    [package_name.]routinename.param_name

<basic-type> ::=
  CHARACTER SET |
  COLLATION |
  DOMAIN |
  EXCEPTION |
  FILTER |
  GENERATOR |
  INDEX |
```

```
ROLE |
USER |
SEQUENCE |
TABLE |
TRIGGER |
VIEW
```

Аргумент	Описание
sometext	Текст комментария
basic-type	Тип объекта метаданных
objectname	Имя объекта метаданных
relationname	Имя таблицы или представления
filedname	Имя поля таблицы или представления
routinename	Имя хранимой процедуры или функции
paramname	Имя параметра хранимой процедуры или функции
package_name	Имя пакета

Описание:

Оператор COMMENT ON добавляет комментарии для объектов базы данных (метаданных). Комментарии при этом сохраняются в текстовые поля RDB$DESCRIPTION типа BLOB соответствующей системной таблицы (из этих полей клиентское приложение может просмотреть комментарии).

Примечание:

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

Добавлять комментарий могут:

- владелец объекта, для которого добавляется комментарий;
- пользователь SYSDBA;
- любой пользователь, подключенный с ролью RDB$ADMIN (роль должна быть назначена пользователю);
- пользователь операционной системы root (Linux);
- администраторы Windows, если используется доверительная авторизация (trusted authentication) и включено автоматическое предоставление роли RDB$ADMIN администраторам Windows.

Примеры:

1. Добавление комментария для текущей базы данных

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

2. Добавление комментария для таблицы METALS

```
COMMENT ON TABLE METALS IS 'Справочник металлов';
```

3. Добавление комментария для поля ISALLOY таблицы METALS

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

4. Добавление комментария для параметра

```
COMMENT ON PARAMETER ADD_EMP_PROJ. EMP_NO IS 'Код
  сотрудника';
```

5. Добавление комментария для пакета, его процедур и функций, и их параметров

```
COMMENT ON PACKAGE APP_VAR IS 'Переменные приложения';
COMMENT ON FUNCTION APP_VAR.GET_DATEBEGIN
IS 'Возвращает дату начала периода';
COMMENT ON
PROCEDURE APP_VAR.SET_DATERANGE
IS 'Установка диапазона дат';
COMMENT ON
PROCEDURE PARAMETER APP_VAR.SET_DATERANGE.ADATEBEGIN
IS 'Дата начала';
```

Новое в PSQL

PSQL функции

Хранимая функция является программой, хранящейся в области метаданных базы данных и выполняющейся на стороне сервера. К хранимой функции могут обращаться хранимые процедуры, хранимые функции (в том числе и сама к себе), триггеры и клиентские программы. При обращении хранимой функции самой к себе такая хранимая функция называется рекурсивной.

В отличие от хранимых процедур хранимые функции всегда возвращают одно скалярное значение. Для возврата значения из хранимой функции используется оператор RETURN, который немедленно прекращает выполнение функции.

CREATE FUNCTION

Создание новой хранимой функции.

Доступно: DSQL

Синтаксис:

```
CREATE FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
  { EXTERNAL NAME '<extname>' ENGINE <engine> }
| {
    AS
    [<declarations>]
    BEGIN
    [<PSQL_statements>]
    END
  }

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<value> ::= {literal | NULL | context_var}

<param_decl> ::=
  paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

<type> ::=
  <datatype>
| [TYPE OF] domain
| TYPE OF COLUMN rel.col
```

```
<datatype> ::=
  {SMALLINT | INTEGER | BIGINT}
| BOOLEAN
| {FLOAT | DOUBLE PRECISSION}
| {DATE | TIME | TIMESTAMP}
| {DECIMAL | NUMERIC} [(precision [, scale])]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
  [CHARACTER SET charset]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING]
  [(size)]
| BLOB [SUB_TYPE {subtype_num | subtype_name}]
  [SEGMENT SIZE seglen] [CHARACTER SET charset]
| BLOB [(seglen [, subtype_num])]

<declarations> ::= <declare_item> [<declare_item> …]

<declare_item> ::=
  <declare_var>; |
  <declare_cursor>; |
  <declare_subfunc> |
  <declare_subproc>
```

Аргумент	Описание
funcname	Имя хранимой функции. Может содержать до 31 символа.
inparam	Описание входного параметра.
declarations	Секция объявления локальных переменных, именованных курсоров и подпрограмм.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
declare_subfunc	Объявление подпрограммы – функции.
declare_subproc	Объявление подпрограммы – процедуры.
PSQL_statments	Операторы языка PSQL.
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного или выходного параметра процедуры. Может содержать до 31 символа. Имя параметра должно быть уникальным среди входных и выходных параметров процедуры, а также её локальных переменных.
module name	Имя внешнего модуля, в котором расположена функция.
routine name	Внутреннее имя функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain	Домен.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.
precision	Точность. От 1 до 18.
scale	Масштаб. От 0 до 18, должно быть меньше или равно precission.
size	Максимальный размер строки в символах.
charset	Набор символов.
subtype_num	Номер подтипа BLOB.

subtype_name	Мнемоника подтипа BLOB.
seqlen	Размер сегмента, не может превышать 65535.

Описание:

Оператор CREATE FUNCTION создаёт новую хранимую функцию. Имя хранимой функции должно быть уникальным среди имён всех хранимых функций и внешних (UDF) функций. Если только это не внутренняя функция («подпрограмма»). Для внутренних функций достаточно уникальности только в рамках модулей, которые их «охватывают».

Замечание:

Желательно также, чтобы имя хранимой функции было уникальным и среди имён функций расположенных в PSQL пакетах (package), хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноимённая функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

Входные параметры передаются в функцию по значению, то есть любые изменения входных аргументов внутри функции никак не повлияют на значения этих параметров в вызывающей программе. Входные параметры могут иметь значение по умолчанию. Параметры, для которых заданы значения, должны располагаться в конце списка параметров. Если входной параметр основан на домене, то значение по умолчанию, указанное для параметра, перекрывает значение по умолчанию указанное при описании домена.

У каждого параметра указывается тип данных. Кроме того, для параметра можно указать ограничение NOT NULL, тем самым запретив передавать в него значение NULL. Для параметра строкового типа существует возможность задать порядок сортировки с помощью предложения COLLATE.

В качестве типа параметра можно указать имя домена. В этом случае, параметр будет наследовать все характеристики домена. Если перед названием домена дополнительно используется предложение "TYPE OF", то используется только тип данных домена – не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию. Если домен текстового типа, то всегда используется его набор символов и порядок сортировки.

Входные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение TYPE OF COLUMN, после которого указывается имя таблиц или представления и через точку имя столбца. При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё набор символов и порядок сортировки. Ограничения и значения по умолчанию столбца никогда не используются.

Предложение RETURNS задаёт тип возвращаемого значения хранимой функции. Если функция возвращает значение строкового типа, то существует возможность задать порядок сортировки с помощью предложения COLLATE. В качестве типа выходного значения можно указать имя домена, ссылку на его тип (с помощью предложения TYPE OF) или ссылку на тип столбца таблицы (с помощью предложения TYPE OF COLUMN).

Необязательное предложение DETERMINISTIC указывает, что функция детерминированная. Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений. Если для функции указано, что она является детерминированной, то такая функция не вычисляется заново, если она уже была вычислена однажды с данным набором входных аргументов, а берёт свои значения из кэша метаданных (если они там есть).

Примечание:

На самом деле в текущей версии Firebird, не существует кэша хранимых функций с маппингом входных аргументов на выходные значения.

Указание инструкции ‘deterministic’ на самом деле нечто вроде «обещания», что код функции будет возвращать одно и то же. В данный момент детерминистическая функция считается инвариантом и работает по тем же принципам, что и другие инварианты. Т.е. вычисляется и кэшируется на уровне текущего выполнения данного запроса.

Это легко демонстрируется таким примером:

```
CREATE FUNCTION FN_T
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
begin
  return rand();
end

-- функция будет вычислена дважды и вернёт 2 разных значения
select fn_t() from rdb$database
union all
select fn_t() from rdb$database

-- функция будет вычислена единожды и вернёт 2 одинаковых значения
with t(n) as (
  select 1 from rdb$database
  union all
  select 2 from rdb$database
)
select n, fn_t()  from t
```

См.
http://www.sql.ru/forum/actualutils.aspx?action=gotomsg&tid=1081535&msg=15694955

В необязательной секции *declarations* описаны локальные переменные, именованные курсоры и подпрограммы.

После секции объявления в теле хранимой функции, следует блок PSQL операторов, заключённых в операторные скобки BEGIN и END.

Хранимая функция может быть расположена во внешнем модуле. В этом случае вместо тела функции указывается место расположения функции во внешнем модуле с помощью предложения EXTERNAL NAME. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя функции внутри модуля и определённая пользователем информация. В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок UDR.

Предупреждение:

Не следует путать внешние функции, объявленные как DECLARE EXTERNAL FUNCTION, так же известные как UDF, с функциями расположенными во внешних модулях объявленных как CREATE FUNCTION … EXTERNAL NAME, называемых UDR (User Defined Routine). Первые являются унаследованными (Legacy) из предыдущих версий Firebird. Их возможности существенно уступают возможностям нового типа внешних функций.

Создать новую хранимую функцию может:

- SYSDBA;
- Владелец базы данных;
- Любой пользователь, которому выдана привилегия на создание функций (GRANT CREATE FUNCTION TO [USER | ROLE] <user/role name>);
- Любой пользователь, вошедший с ролью RDB$ADMIN;
- Пользователь root операционной системы Linux;
- администраторы Windows, если используется доверительная авторизация (trusted authentication) и включено автоматическое предоставление роли RDB$ADMIN администраторам Windows.

Пользователь, создавший хранимую процедуру, становится её владельцем.

Примеры:

1. Функция сложения двух целочисленных значений. Второй параметр объявлен необязательным и имеет значение по умолчанию равное нулю.

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
```

```
AS
BEGIN
  RETURN A+B;
END
```

Вызов в запросе:

```
SELECT ADD_INT(2, 3) AS R FROM RDB$DATABASE
```

Вызов внутри PSQL кода, второй необязательный параметр не указан:

```
MY_VAR = ADD_INT(A);
```

2. Детерминированная функция, возвращающая значение константы *e*:

```
CREATE FUNCTION FN_E()
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
  RETURN EXP(1);
END
```

3. Функция, возвращающая имя мнемоники по имени столбца и значения мнемоники.

```
CREATE FUNCTION GET_MNEMONIC (
    AFIELD_NAME TYPE OF COLUMN RDB$TYPES.RDB$FIELD_NAME,
    ATYPE TYPE OF COLUMN RDB$TYPES.RDB$TYPE)
RETURNS TYPE OF COLUMN RDB$TYPES.RDB$TYPE_NAME
AS
BEGIN
  RETURN (SELECT RDB$TYPE_NAME
          FROM RDB$TYPES
          WHERE RDB$FIELD_NAME = :AFIELD_NAME
            AND RDB$TYPE = :ATYPE);
END
```

4. Создание функции находящейся во внешнем модуле (UDR). Реализация функции расположена во внешнем модуле udrcpp_example. Имя функции внутри модуля - wait_event.

```
create function wait_event (
   event_name varchar(31) character set ascii
) returns integer
external name 'udrcpp_example!wait_event'
engine udr;
```

См. также CREATE OR ALTER FUNCTION, RECREATE FUNCTION, ALTER FUNCTION, DROP FUNCTION

ALTER FUNCTION

Изменение существующей хранимой функции.

Доступно: DSQL

Синтаксис:

```
ALTER FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
  { EXTERNAL NAME '<extname>' ENGINE <engine> }
| {
    AS
    [<declarations>]
    BEGIN
    [<PSQL_statements>]
    END
  }
```

Подробнее см. CREATE FUNCTION

Описание:

Оператор ALTER FUNCTION позволяет изменять состав и характеристики входных параметров, типа выходного значения, локальных переменных, именованных курсоров, подпрограмм и тело хранимой функции. Для внешних (UDR) вы можете изменить точку входа и имя движка. Внешние функции, объявленные как DECLARE EXTERNAL FUNCTION, так же известные как UDF, невозможно преобразовать в PSQL функции и наоборот. После выполнения существующие привилегии и зависимости сохраняются.

Предупреждение:

Будьте осторожны при изменении количества и типов входных параметров хранимых функций. Существующий код приложения может стать неработоспособным из-за того, что формат вызова функции несовместим с новым описанием параметров. Кроме того, PSQL модули, использующие изменённую хранимую функцию, могут стать некорректными. Информация о том, как это обнаружить, находится в разделе RDB$VALID_BLR.

Изменить хранимую функцию могут:

- владелец хранимой функции;
- пользователь SYSDBA;
- Любой пользователь, которому выдана привилегия на изменение любой функций (GRANT ALTER ANY FUNCTION TO [USER | ROLE] <user/role name>);

- любой пользователь, подключенный с ролью RDB$ADMIN (роль должна быть назначена пользователю);
- пользователь операционной системы root (Linux);
- администраторы Windows, если используется доверительная авторизация (trusted authentication) и включено автоматическое предоставление роли RDB$ADMIN администраторам Windows.

Примеры:

Изменение функции сложения целочисленных значений. Вместо двух теперь складывается три значения.

```
ALTER FUNCTION ADD_INT(A INT, B INT, C INT)
RETURNS INT
AS
BEGIN
  RETURN A+B+C;
END
```

См. также CREATE FUNCTION, DROP FUNCTION

CREATE OR ALTER FUNCTION

Создание новой или изменение существующей хранимой функции.

Доступно: DSQL

Синтаксис:

```
CREATE OR ALTER FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
  { EXTERNAL NAME '<extname>' ENGINE <engine> }
| {
    AS
    [<declarations>]
    BEGIN
    [<PSQL_statements>]
    END
  }
```

Подробнее см. CREATE FUNCTION

Описание:

Оператор CREATE OR ALTER FUNCTION создаёт новую или изменяет существующую хранимую функцию. Если хранимая функция не существует, то она будет создана с использованием предложения CREATE FUNCTION . Если она

уже существует, то она будет изменена и перекомпилирована, при этом существующие привилегии и зависимости сохраняются.

Примеры:

Создание или изменение функции сложения двух целочисленных значений.

```
CREATE OR ALTER FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
  RETURN A+B;
END
```

См. также CREATE FUNCTION, ALTER FUNCTION

DROP FUNCTION

Удаление хранимой функции.

Доступно: DSQL

Синтаксис:

```
DROP FUNCTION funcname;
```

Аргумент	Описание
funcname	Имя хранимой функции.

Описание:

Оператор DROP FUNCTION удаляет существующую хранимую функцию. Если от хранимой функции существуют зависимости, то при попытке удаления такой функции будет выдана соответствующая ошибка.

Удалить хранимую функцию могут:

- владелец хранимой функции;
- пользователь SYSDBA;
- любой пользователь, подключенный с ролью RDB$ADMIN (роль должна быть назначена пользователю);
- любой пользователь, которому назначена привилегия на удаление любой функции (GRANT DROP ANY FUNCTION TO [USER | ROLE] <user/role name>);
- пользователь операционной системы root (Linux);
- администраторы Windows, если используется доверительная авторизация (trusted authentication) и назначено автоматическое предоставление роли RDB$ADMIN администраторам Windows.

Примеры:

Удаление функции ADD_INT.

```
DROP FUNCTION ADD_INT;
```

См. также CREATE FUNCTION

RECREATE FUNCTION

Создание новой или пересоздание существующей хранимой функции.

Доступно: DSQL

Синтаксис:

```
RECREATE FUNCTION funcname
[(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
  { EXTERNAL NAME '<extname>' ENGINE <engine> }
| {
    AS
    [<declarations>]
    BEGIN
    [<PSQL_statements>]
    END
  }
```

Подробнее см. CREATE FUNCTION

Описание:

Оператор RECREATE FUNCTION создаёт новую или пересоздаёт существующую хранимую функцию. Если функция с таким именем уже существует, то оператор попытается удалить её и создать новую функцию. Пересоздать функцию невозможно, если у существующей функции имеются зависимости. После пересоздания функции привилегии на выполнение хранимой функции и привилегии самой хранимой функции не сохраняются.

Примеры:

Создание или пересоздание функции сложения двух целочисленных значений.

```
RECREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
  RETURN A+B;
END
```

См. также CREATE FUNCTION, DROP FUNCTION

Внешние процедуры

Начиная с Firebird 3, хранимая процедура может быть расположена во внешнем модуле. В этом случае вместо тела процедуры указывается место её расположения во внешнем модуле с помощью предложения EXTERNAL NAME. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя процедуры внутри модуля и определённая пользователем информация. В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок UDR.

Доступно: DSQL

Синтаксис:

```
<procedure> ::=
  { CREATE [OR ALTER] | ALTER | RECREATE } PROCEDURE procname
  [(<inparam> [, <inparam> ...])]
  [RETURNS (<outparam> [, <outparam> ...])]
    { EXTERNAL NAME '<extname>' ENGINE <engine> }
  | {
      AS
      [<declarations>]
      BEGIN
      [<PSQL_statements>]
      END
    }

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<outparam> ::= <param_decl>

<value> ::= {literal | NULL | context_var}

<param_decl> ::=
  paramname <type> [NOT NULL] [COLLATE collation]

<type> ::=
  <datatype>
| [TYPE OF] domain
| TYPE OF COLUMN rel.col

<datatype> ::=
  {SMALLINT | INTEGER | BIGINT}
| {FLOAT | DOUBLE PRECISSION}
| {DATE | TIME | TIMESTAMP}
| {DECIMAL | NUMERIC} [(precision [, scale])]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
```

```
  [CHARACTER SET charset]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING]
  [(size)]
| BLOB [SUB_TYPE {subtype_num | subtype_name}]
  [SEGMENT SIZE seglen] [CHARACTER SET charset]
| BLOB [(seglen [, subtype_num])]

<declarations> ::=
  {<declare_var> | <declare_cursor>};
  [{<declare_var> | <declare_cursor>}; …]
```

Аргумент	Описание
procname	Имя хранимой процедуры. Может содержать до 31 символа.
inparam	Описание входного параметра.
outparam	Описание выходного параметра.
declarations	Секция объявления локальных переменных и именованных курсоров.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
PSQL_statments	Операторы языка PSQL.
literal	Литерал.
context_var	Любая контекстная переменная, тип которой совместим с типом параметра.
paramname	Имя входного или выходного параметра процедуры. Может содержать до 31 символа. Имя параметра должно быть уникальным среди входных и выходных параметров процедуры, а также её локальных переменных.
module name	Имя внешнего модуля, в котором расположена функция.
routine name	Внутреннее имя функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
datatype	Тип данных SQL.
collation	Порядок сортировки.
domain	Домен.
rel	Имя таблицы или представления.
col	Имя столбца таблицы или представления.
precision	Точность. От 1 до 18.
scale	Масштаб. От 0 до 18, должно быть меньше или равно precission.
size	Максимальный размер строки в символах.
charset	Набор символов.
subtype_num	Номер подтипа BLOB.
subtype_name	Мнемоника подтипа BLOB.
seqlen	Размер сегмента, не может превышать 65535.

Примеры:

Создание процедуры находящейся во внешнем модуле (UDR). Реализация процедуры расположена во внешнем модуле udrcpp_example. Имя процедура внутри модуля – gen_rows.

```
create procedure gen_rows (
  start_n integer not null,
  end_n integer not null
)
```

```
    returns (
      n integer not null
    )
    external name 'udrcpp_example!gen_rows'
    engine udr;
```

Внешние триггеры

Начиная с Firebird 3, триггеры могут быть расположены во внешнем модуле. В этом случае вместо тела триггера указывается место его расположения во внешнем модуле с помощью предложения EXTERNAL NAME. Аргументом этого предложения является строка, в которой через разделитель указано имя внешнего модуля, имя функции внутри модуля и определённая пользователем информация. В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями используется движок UDR.

Доступно: DSQL, ESQL

Синтаксис:

```
<trigger> ::=
  { CREATE [OR ALTER] | ALTER | RECREATE } TRIGGER trigname
  { <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger> }
    { EXTERNAL NAME '<extname>' ENGINE <engine> }
  | {
        AS
        [<declarations>]
        BEGIN
        [<PSQL_statements>]
        END
    }


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

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

<database_trigger> ::=
  [ACTIVE | INACTIVE]
      ON <db_event>
    | {BEFORE | AFTER} <ddl event>
```

```
  [POSITION number]

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }

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


<ddl event> ::=
  ANY DDL STATEMENT
| <ddl event item> [{OR <ddl event item>}...]

<ddl event item> ::=
  CREATE TABLE
| ALTER TABLE
| DROP TABLE
| CREATE PROCEDURE
| ALTER PROCEDURE
| DROP PROCEDURE
| CREATE FUNCTION
| ALTER FUNCTION
| DROP FUNCTION
| CREATE TRIGGER
| ALTER TRIGGER
| DROP TRIGGER
| CREATE EXCEPTION
| ALTER EXCEPTION
| DROP EXCEPTION
| CREATE VIEW
| ALTER VIEW
| DROP VIEW
| CREATE DOMAIN
| ALTER DOMAIN
| DROP DOMAIN
| CREATE ROLE
| ALTER ROLE
| DROP ROLE
| CREATE SEQUENCE
| ALTER SEQUENCE
| DROP SEQUENCE
| CREATE USER
| ALTER USER
| DROP USER
| CREATE INDEX
| ALTER INDEX
```

```
| DROP INDEX
| CREATE COLLATION
| DROP COLLATION
| ALTER CHARACTER SET
| CREATE PACKAGE
| ALTER PACKAGE
| DROP PACKAGE
| CREATE PACKAGE BODY
| DROP PACKAGE BODY

<declarations> ::=
  {<declare_var> | <declare_cursor>};
 [{<declare_var> | <declare_cursor>}; …]
```

Аргумент	Описание
trigname	Имя триггера. Может содержать до 31 символа.
relation_trigger_legacy	Объявление табличного триггера (унаследованное).
relation_trigger_sql2003	Объявление табличного триггера согласно стандарту SQL-2003.
database_trigger	Объявление триггера базы данных.
tablename	Имя таблицы.
viewname	Имя представления.
mutation_list	Список событий таблицы.
number	Порядок срабатывания триггера. От 0 до 32767.
db_event	Событие соединения или транзакции.
declarations	Секция объявления локальных переменных и именованных курсоров.
declare_var	Объявление локальной переменной.
declare_cursor	Объявление именованного курсора.
PSQL_statments	Операторы языка PSQL.

Примеры:

Создание триггера находящегося во внешнем модуле (UDR). Реализация триггера расположена во внешнем модуле udrcpp_example. Имя триггера внутри модуля – replicate. В функцию реализующую триггер внутри внешнего модуля передаётся дополнительная информация «ds1».

```
create trigger persons_replicate
after insert on persons
external name 'udrcpp_example!replicate!ds1'
engine udr;
```

Пакеты

Пакет - группа процедур и функций, которая представляет собой один объект базы данных. Введение пакетов преследует несколько целей:

- *Модульность*

  Идея состоит в том, чтобы выделить блоки взаимозависимого кода в логические модули, как это сделано в других языках программирования.

В программировании существует множество способов для группировки кода, например с помощью пространств имён (namespaces), модулей (units) и классов. Со стандартными процедурами и функциями базы данных это не возможно.

Хотя они могут быть сгруппированы в различных файлах сценария, остаются не решёнными две проблемы:

1. Группировка не представлена в метаданных базы данных.
2. В сценарии все подпрограммы находятся в плоском пространстве имён и могут быть вызваны всеми (здесь не имеется ввиду разрешения безопасности).

- *Упрощение отслеживания зависимостей*

  Мы хотим иметь упрощённый механизм отслеживания зависимостей между набором связанных процедур, а также между этим набором и другими процедурами, как упакованными, так и неупакованными.

  Пакеты Firebird состоят из двух частей: заголовка (ключевое слово PACKAGE) и тела (ключевые слова PACKAGE BODY). Такое разделение очень сильно напоминает модули Delphi, заголовок соответствует интерфейсной части, а тело – части реализации.

  Сначала создаётся заголовок (CREATE PACKAGE), а затем – тело (CREATE PACKAGE BODY).

  Каждый раз, когда упакованная подпрограмма определяет, что используется некоторый объект базы данных, информации о зависимости от этого объекта регистрируется в системных таблицах Firebird. После этого, для того чтобы удалить или изменить этот объект, вы сначала должны удалить, то что зависит от него. Поскольку зависимости от других объектов существуют только для тела пакета, это тело пакета может быть легко удалено, даже если какой-нибудь другой объект зависит от этого пакета. Когда тело удаляется, заголовок остаётся, что позволяет пересоздать это тело после того, как сделаны изменения связанные с удалённым объектом.

- *Упрощение управления разрешениями*

  Хорошей практикой при создании подпрограмм, является требование её привилегированного использования, а также использования пользователей и ролей, для того чтобы позволить привилегированного использования. Поскольку Firebird выполняет подпрограммы с полномочиями вызывающей стороны, необходимо также предоставить полномочия на использования ресурсов каждой вызывающей подпрограмме, если эти ресурсы не являются непосредственно доступными вызывающей стороне. Использование каждой подпрограммы требует предоставления привилегий на её выполнение для пользователей и/или ролей.

  У упакованных подпрограмм нет отдельных привилегий. Привилегии действуют на пакет в целом. Привилегии, предоставленные пакетам,

действительны для всех подпрограмм тела пакета, в том числе частных, и сохраняются для заголовка пакета.

> *Замечание:*
>
> На самом деле это является и преимуществом и недостатком одновременно. Если вы группируете в пакете некоторые бизнес процедуры, у которых должен быть разный набор привилегий, то вам придётся предусмотреть проверку прав в исходном коде упакованных процедур, или решать проблемы иным способом.

Примеры:

```
GRANT SELECT ON TABLE secret TO PACKAGE pk_secret;

GRANT EXECUTE ON PACKAGE pk_secret TO ROLE role_secret;
```

- *Разрешение «частных областей видимости»*

  Эта цель состоит в том, чтобы сделать некоторые процедуры частными (private), а именно разрешить их использование только внутри определения пакета.

  Все языки программирования имеют понятие области видимости подпрограмм, которое невозможно без какой-либо формы группировки. Пакеты Firebird в этом отношении подобны модулям Delphi. Если подпрограмма не объявлена в заголовке пакета (interface), но реализована в теле (implementation), то такая подпрограмма становится частной (private). Частную подпрограмму возможно вызвать только из её пакета.

Синтаксис:

```
<package_header> ::=
  { CREATE [OR ALTER] | ALTER | RECREATE } PACKAGE <name>
  AS
  BEGIN
    [ <package_item> ... ]
  END

<package_item> ::=
  <function_decl> ; |
  <procedure_decl> ;

<function_decl> ::=
  FUNCTION <name> [( <parameters> )] RETURNS <type>

<procedure_decl> ::=
  PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]
```

```
<package_body> ::=
  { CREATE | RECREATE } PACKAGE BODY <name>
  AS
  BEGIN
    [ <package_item> ... ]
    [ <package_body_item> ... ]
  END

<package_body_item> ::=
  <function_impl> |
  <procedure_impl>

<function_impl> ::=
  FUNCTION <name> [( <parameters> )] RETURNS <type>
  AS
  BEGIN
  ...
  END
  |
  FUNCTION <name> [( <parameters> )] RETURNS <type>
  EXTERNAL NAME '<name>' ENGINE <engine>

<procedure_impl> ::=
  PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]
  AS
  BEGIN
  ...
  END
  |
  PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]
  EXTERNAL NAME '<name>' ENGINE <engine>

<drop_package_header> ::=
  DROP PACKAGE <name>

<drop_package_body> ::=
  DROP PACKAGE BODY <name>
```

Правила:

- В теле пакеты должны быть реализованы все подпрограммы, стой же сигнатурой, что и объявленные в заголовке и в начале тела пакета.
- Значения по умолчанию для параметров процедур не могут быть переопределены (которые указываются в <package_item>). Это означает, что они могут быть в <package_body_item> только для частных процедур, которые не были объявлены.

Замечания:

Перед удалением заголовка пакета (DROP PACKAGE), необходимо выполнить удаление тела пакета (DROP PACKAGE BODY).

Исходный код тела пакета сохраняется после ALTER / RECREATE PACKAGE. Столбец RDB$PACKAGES.RDB$VALID_BODY_FLAG отображает состояние тела пакета.

UDF деклараций (DECLARE внешняя функция) в настоящее время не поддерживается внутри пакетов.

Желательно чтобы имена хранимых процедур и функций пакета не пересекались с именами хранимых процедур и функций из глобального пространства имён, хотя это и допустимо. Дело в том, что в настоящее время вы не сможете вызвать функцию/процедуру из глобального пространства имён внутри пакета, если в пакете объявлена одноимённая функция/процедура. В этом случае всегда будет вызвана процедура/функция пакета.

Примеры:

1. Пример простого пакета

```
SET TERM ^;
-- package header, declarations only
CREATE OR ALTER PACKAGE TEST
AS
BEGIN
  PROCEDURE P1(I INT) RETURNS (O INT); -- public procedure
END ^

-- package body, implementation
RECREATE PACKAGE BODY TEST
AS
BEGIN
  FUNCTION F1(I INT) RETURNS INT; -- private function

  PROCEDURE P1(I INT) RETURNS (O INT)
  AS
  BEGIN
  END

  FUNCTION F1(I INT) RETURNS INT
  AS
  BEGIN
    RETURN 0;
  END
END ^
```

2. Пакет функций и процедур для работы с сессионными переменными приложения.

```
SET TERM ^ ;
```

```
CREATE PACKAGE APP_VAR
AS
begin
  -- Возвращает дату начала периода
  -- Функция помечена как детерминированная, что позволяет
  -- рассматривать её как инвариант в запросах
  function GET_DATEBEGIN() returns date deterministic;

  -- Возвращает дату окончания периода
  -- Функция помечена как детерминированная, что позволяет
  -- рассматривать её как инвариант в запросах
  function GET_DATEEND() returns date deterministic;

  -- Устанавливает диапазон дат рабочего периода
  procedure SET_DATERANGE(ADATEBEGIN date, ADATEEND date);
end^

CREATE PACKAGE BODY APP_VAR
AS
begin
  -- Возвращает дату начала периода
  function GET_DATEBEGIN() returns date
  as
  begin
    return RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
  end

  -- Возвращает дату окончания периода
  function GET_DATEEND() returns date
  as
  begin
    return RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
  end

  -- Устанавливает диапазон дат рабочего периода
  procedure SET_DATERANGE(ADATEBEGIN date, ADATEEND date)
  as
  begin
    RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN',
ADATEBEGIN);
    RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
  end
end^

SET TERM ; ^
```

Использование:

```
-- Установка рабочего периода
execute procedure APP_VAR.SET_DATERANGE(
```

```
  date '01.01.2000', date '31.12.2000');

-- Использование в запросах
select *
from SALES_ORDER
where bydate between APP_VAR.GET_DATEBEGIN()
                 and APP_VAR.GET_DATEEND();
```

3. Пакет процедур и функций, работающих с Perl-совместимыми регулярными выражениями. Процедуры для работы с регулярными выражениями находятся во внешнем модуле.

```
SET TERM ^ ;

CREATE OR ALTER PACKAGE REGEXP
AS
begin
  procedure preg_match(
      APattern varchar(8192), ASubject varchar(8192))
    returns (Matches varchar(8192));

  function preg_is_match(
      APattern varchar(8192), ASubject varchar(8192))
    returns boolean;

  function preg_replace(
      APattern varchar(8192),
      AReplacement varchar(8192),
      ASubject varchar(8192))
    returns varchar(8192);

  procedure preg_split(
      APattern varchar(8192),
      ASubject varchar(8192))
    returns (Lines varchar(8192));

  function preg_quote(
      AStr varchar(8192),
      ADelimiter char(10) default null)
    returns varchar(8192);
end^

RECREATE PACKAGE BODY REGEXP
AS
begin
  procedure preg_match(
      APattern varchar(8192),
      ASubject varchar(8192))
    returns (Matches varchar(8192))
    external name 'PCRE!preg_match' engine UDR;
```

```
  function preg_is_match(
      APattern varchar(8192),
      ASubject varchar(8192))
    returns boolean
  as
  begin
    return exists(
      select * from preg_match(:APattern, :ASubject));
  end

  function preg_replace(
      APattern varchar(8192),
      AReplacement varchar(8192),
      ASubject varchar(8192))
    returns varchar(8192)
    external name 'PCRE!preg_replace' engine UDR;

  procedure preg_split(
      APattern varchar(8192),
      ASubject varchar(8192))
    returns (Lines varchar(8192))
    external name 'PCRE!preg_split' engine UDR;

  function preg_quote(
      AStr varchar(8192),
      ADelimiter char(10))
    returns varchar(8192)
    external name 'PCRE!preg_quote' engine UDR;
end^

SET TERM ; ^
```

4. Пример, показывающий как «выдать» привилегии на отдельные процедуры пакета. На самом деле, привилегии выдаются на весь пакет в целом, поэтому в коде процедур необходимо предпринять дополнительные действия для ограничения для отдельных пользователей или/и ролей.

```
SET TERM ^;

CREATE OR ALTER PACKAGE PKG_BILL
AS
begin
  procedure ADD_BILL(ABillNumber varchar(12),
                     ACodeCompany int,
                     AByDate date default current_date)
    returns (Code_Bill int);

  procedure CLOSE_BILL(ACode_Bill int);

end^
```

```
RECREATE PACKAGE BODY PKG_BILL
AS
begin
  -- Частная процедура. Предварительное объявление
  procedure P_CLOSE_BILL(ACode_Bill int);

  procedure ADD_BILL(ABillNumber varchar(12),
                     ACodeCompany int,
                     AByDate date)
    returns (Code_Bill int)
  as
  begin
    insert into bill(Code_Company, ANumber, Bydate)
    values (:ACodeCompany, :ABillNumber, :AByDate)
    returning Code_Bill
    into Code_Bill;
  end

  procedure CLOSE_BILL(ACode_Bill int)
  as
  begin
    -- Если текущая роль не менеджер генерируем исключение
    if (CURRENT_ROLE <> 'MANAGER') then
      exception E_ACCESS_DENIED;
    execute procedure P_CLOSE_BILL(ACode_Bill);
  end

  -- Частная процедура. Реализация
  procedure P_CLOSE_BILL(ACode_Bill int)
  as
  begin
    update bill
    set CloseFlag = true
    where Code_Bill = :ACode_Bill;
  end

end^

SET TERM ;^

GRANT SELECT, DELETE, INSERT, UPDATE ON BILL TO PACKAGE
PKG_BILL;

GRANT EXECUTE ON PACKAGE PKG_BILL TO ROLE SELLER;
GRANT EXECUTE ON PACKAGE PKG_BILL TO ROLE MANAGER;
```

DDL триггеры

Целью "DDL триггера" является обеспечение ограничений, которые будут распространены на пользователей, которые пытаются создать, изменить или удалить DDL объект. Другое их назначение – ведение журнала изменений метаданных.

Синтаксис:

```
<database-trigger> ::=
  {CREATE | RECREATE | CREATE OR ALTER}
  TRIGGER <name>
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <ddl event>
  [POSITION <n>]
  AS
  BEGIN
  ...
  END

<ddl event> ::=
  ANY DDL STATEMENT
| <ddl event item> [{OR <ddl event item>}...]

<ddl event item> ::=
  CREATE TABLE
| ALTER TABLE
| DROP TABLE
| CREATE PROCEDURE
| ALTER PROCEDURE
| DROP PROCEDURE
| CREATE FUNCTION
| ALTER FUNCTION
| DROP FUNCTION
| CREATE TRIGGER
| ALTER TRIGGER
| DROP TRIGGER
| CREATE EXCEPTION
| ALTER EXCEPTION
| DROP EXCEPTION
| CREATE VIEW
| ALTER VIEW
| DROP VIEW
| CREATE DOMAIN
| ALTER DOMAIN
| DROP DOMAIN
| CREATE ROLE
| ALTER ROLE
| DROP ROLE
| CREATE SEQUENCE
| ALTER SEQUENCE
| DROP SEQUENCE
| CREATE USER
| ALTER USER
```

```
| DROP USER
| CREATE INDEX
| ALTER INDEX
| DROP INDEX
| CREATE COLLATION
| DROP COLLATION
| ALTER CHARACTER SET
| CREATE PACKAGE
| ALTER PACKAGE
| DROP PACKAGE
| CREATE PACKAGE BODY
| DROP PACKAGE BODY
```

Важное правило:

Тип события [BEFORE | AFTER] DDL не может быть изменено.

Семантика (смысл):

1. BEFORE триггеры запускаются до изменений в системных таблицах. AFTER триггеры запускаются после изменений в системных таблицах.
2. Когда оператор DDL запускает триггер, в котором возбуждается исключение (BEFORE или AFTER, преднамеренно или неумышленно), оператор не будет фиксирован. Т.е. исключения могут использоваться, чтобы гарантировать, что оператор DDL будет отменён, если некоторые условия не будут соблюдены.
3. Действия DDL триггеров выполняются только при фиксации транзакции, в которой работает затронутая DDL команда. Никогда не забывайте о том, что в AFTER триггере, возможно сделать только то, что возможно сделать после DDL команды без автоматической фиксации транзакций. Вы не можете, например, создать таблицу в триггере и использовать её там.
4. Для операторов «CREATE OR ALTER» триггер срабатывает один раз для события CREATE или события ALTER, в зависимости от того существовал ли ранее объект. Для операторов RECREATE триггер вызывается для события DROP, если объект существовал, и после этого для события CREATE.
5. Триггеры на события ALTER и DROP обычно не запускаются. Если объект не существует. Исключения описаны в пункте 6.
6. Исключением из правила 5 являются BEFORE ALTER/DROP USER триггеры, которые будут вызваны, даже если имя пользователя не существует. Это вызвано тем, что эти команды выполняются для базы данных безопасности, для которой не делается проверка существования пользователей перед их выполнением. Данное поведение, вероятно, будет отличаться для встроенных пользователей, поэтому не пишите код, который зависит от этого.
7. Если некоторое исключение возбуждено после того как начала выполняться DDL команда и до того как запущен AFTER триггер, то AFTER триггер не запускается.
8. Для упакованных процедур и функций не запускаются индивидуальные триггеры {CREATE | ALTER | DROP} {PROCEDURE | FUNCTION}.

Безопасность

Следующие пользователи создавать, модифицировать или удалять DDL триггеры и имеют доступ к переключателям в утилитах Firebird связанными с триггерами:

- SYSBDA;
- Владелец базы данных;
- Пользователь, вошедший с ролью RDB$ADMIN;
- Пользователь, имеющий привилегию метаданных ALTER DATABASE.

Поддержка в утилитах

DDL триггеры относятся к триггерам базы данных, таким образом, параметры -nodbtriggers (GBAK и ISQL) и -T (NBACKUP) применяются к ним тоже. Помните, что только владелец базы данных и SYSDBA могут использовать эти переключатели.

Пространства имён контекстных переменных DDL_TRIGGER

Введение DDL триггеров повлекло и введение нового пространства имён DDL_TRIGGER для использования в функции RDB$GET_CONTEXT. Его использование допустимо, только во время работы DDL триггера. Его использование допустимо в хранимых процедурах и функциях, вызванных триггерами DDL.

Контекст DDL_TRIGGER работает как стек. Перед возбуждением DDL триггера, значения, относящиеся к выполняемой команде, помещаются в этот стек. После завершения работы триггера значения выталкиваются. Таким образом. В случае каскадных DDL операторов, когда каждая пользовательская DDL команда возбуждает DDL триггер, и этот триггер запускает другие DDL команды, с помощью EXECUTE STATEMENT, значения переменных в пространстве имён DDL_TRIGGER будут соответствовать команде, которая вызвала последний DDL триггер в стеке вызовов.

Переменные в пространстве имён DDL_TRIGGER

- EVENT_TYPE – тип события (CREATE, ALTER, DROP)
- OBJECT_TYPE – тип объекта (TABLE, VIEW и д.р.)
- DDL_EVENT – имя события (<ddl event item>), где <ddl event item> = EVENT_TYPE || ' ' || OBJECT_TYPE
- OBJECT_NAME – имя объекта метаданных
- SQL_TEXT – текст SQL запроса

Примеры использования DDL триггеров

Пример того, как вы могли бы использовать триггер DDL, чтобы осуществить непротиворечивую схему именования объектов, например, обеспечить правило, согласно которому имена хранимых процедур должны начаться с префиксна “SP_”:

```
create exception e_invalid_sp_name 'Неверное имя хранимой
процедуры (должно начинаться с SP_)';

set term !;

create trigger trig_ddl_sp before CREATE PROCEDURE
as
begin
  if (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME')
      not starting 'SP_') then
    exception e_invalid_sp_name;
end!

-- Test
create procedure sp_test
as
begin
end!

create procedure test
as
begin
end!

-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
-- -Неверное имя хранимой процедуры (должно начинаться с SP_)
-- -At trigger 'TRIG_DDL_SP' line: 4, col: 5

set term ;!
```

Реализация пользовательской безопасности для DDL, в данном случае разрешено запускать DDL команды только определённым пользователям:

```
create exception e_access_denied 'Access denied';

set term !;

create trigger trig_ddl before any ddl statement
as
begin
  if (current_user <> 'SUPER_USER') then
    exception e_access_denied;
```

```
end!

-- Test
create procedure sp_test
as
begin
end!

-- The last command raises this exception and procedure SP_TEST
is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5

set term ;!
```

Замечание:

На самом деле в Firebird 3 появились права на DDL операторы, поэтому прибегать к написанию DDL триггера нужно только в случае, если того же самого эффекта невозможно достичь стандартными методами.

Использование триггеров для регистрации DDL действий и их попыток:

```
create sequence ddl_seq;

create table ddl_log (
  id bigint not null primary key,
  moment timestamp not null,
  user_name varchar(31) not null,
  event_type varchar(25) not null,
  object_type varchar(25) not null,
  ddl_event varchar(25) not null,
  object_name varchar(31) not null,
  sql_text blob sub_type text not null,
  ok char(1) not null
);

set term !;

create trigger trig_ddl_log_before before any ddl statement
as
  declare id type of column ddl_log.id;
begin
  -- Мы должны производить изменения в AUTONOMOUS TRANSACTION,
  -- таким образом, если произойдёт исключение и команда
  -- не будет запущена, она всё равно будет зарегистрирована.
```

```
  in autonomous transaction do
  begin
    insert into ddl_log (
      id, moment, user_name, event_type, object_type,
      ddl_event, object_name, sql_text, ok)
    values (next value for ddl_seq,
            current_timestamp, current_user,
            rdb$get_context('DDL_TRIGGER', 'EVENT_TYPE'),
            rdb$get_context('DDL_TRIGGER', 'OBJECT_TYPE'),
            rdb$get_context('DDL_TRIGGER', 'DDL_EVENT'),
            rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME'),
            rdb$get_context('DDL_TRIGGER', 'SQL_TEXT'),
            'N')
    returning id into id;
    rdb$set_context('USER_SESSION', 'trig_ddl_log_id', id);
  end
end!

-- Примечание:
-- созданный выше триггер будет запущен для этой DDL.
-- Хорошей идеей является использование -nodbtriggers
-- при работе с ним
create trigger trig_ddl_log_after after any ddl statement
as
begin
  -- Здесь нам требуется автономная транзакция,
  -- потому что в оригинальной транзакции
  -- мы не увидим запись, вставленную в
  -- BEFORE триггере в автономной транзакции,
  -- если пользовательская транзакции не запущена
  -- с режимом изоляции  READ COMMITTED.
  in autonomous transaction do
    update ddl_log set ok = 'Y'
    where
      id = rdb$get_context('USER_SESSION', 'trig_ddl_log_id');
end!

commit!

set term ;!

-- Удаляем запись о создании trig_ddl_log_after.
delete from ddl_log;
commit;

-- Тест

-- Эта команда будет зарегистрирована единожды
-- (т.к. T1 не существует, RECREATE вызовет событие CREATE)
-- с OK = Y.
recreate table t1 (
```

```
  n1 integer,
  n2 integer
);

-- Оператор не выполнится, т.к. T1 уже существует,
-- таким образом OK будет иметь значение N.
create table t1 (
  n1 integer,
  n2 integer
);

-- T2 не существует. Это действие не будет зарегистрировано.
drop table t2;

-- Это действие будет зарегистрировано дважды
-- (т.к. T1 существует, действие RECREATE рассматривается
-- как DROP и CREATE) с полем OK = Y.
recreate table t1 (
  n integer
);

commit;

select id, ddl_event, object_name, sql_text, ok
from ddl_log order by id;
```

ID	DDL_EVENT	OBJECT_NAME	SQL_TEXT	OK

2	CREATE TABLE	T1	recreate table t1 (
			  n1 integer,
			  n2 integer
			)
				Y

3	CREATE TABLE	T1	create table t1 (
			  n1 integer,
			  n2 integer
			)
				N

4	DROP TABLE	T1	recreate table t1 (n integer)	Y
5	CREATE TABLE	T1	recreate table t1 (n integer)	Y

PSQL подпрограммы

В заголовках PSQL модулей (хранимых процедурах, хранимых функциях, триггерах и анонимных PSQL блоках) теперь разрешено объявлять подпроцедуры и подфункции для использования в теле модуля.

Доступно: PSQL

Синтаксис объявления подпроцедуры:

```
DECLARE PROCEDURE procname [(<inparam> [, <inparam> ...])]
[RETURNS (<outparam> [, <outparam> ...])]
AS
[<declarations>]
BEGIN
[<PSQL_statements>]
END
```

Синтаксис объявления подфункции:

```
DECLARE FUNCTION funcname [(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
AS
[<declarations>]
BEGIN
[<PSQL_statements>]
END
```

Ограничения:

1. Подпрограмма не может быть вложена в другую подпрограмму. Они поддерживаются только в основном модуле (хранимой процедуре, хранимой функции, триггере и анонимном PSQL блоке).
2. В настоящее время подпрограмма не имеет прямого доступа для использования переменных, курсоров и других подпрограмм из основного модуля. Кроме того, подпрограмма не может вызывать себя рекурсивно. Это может быть разрешено в будущем.

Примеры:

```
SET TERM ^;
--
-- Подпроцедуры в EXECUTE BLOCK
--
execute block returns (name varchar(31))
as
  declare procedure get_tables
  returns (table_name varchar(31))
  as
  begin
    for select rdb$relation_name
        from rdb$relations
        where rdb$view_blr is null
        into table_name
    do
```

```
      suspend;
  end

  declare procedure get_views
  returns (view_name varchar(31))
  as
  begin
    for select rdb$relation_name
        from rdb$relations
        where rdb$view_blr is not null
        into view_name
        do
          suspend;
  end

begin
  for select table_name
      from get_tables
      union all
      select view_name
      from get_views
      into name
  do
    suspend;
end^

--
-- Подфункция внутри хранимой функции
--
CREATE OR ALTER FUNCTION FUNC1 (n1 INTEGER, n2 INTEGER)
RETURNS INTEGER
AS
  DECLARE FUNCTION SUBFUNC (n1 INTEGER, n2 INTEGER)
  RETURNS INTEGER
  AS
  BEGIN
    RETURN n1 + n2;
  END
BEGIN
  RETURN SUBFUNC(n1, n2);
END ^

--
select func1(5, 6) from rdb$database ^
```

Расширение использования префикса двоеточия

До сих пор префикс двоеточие (:) использовался в PSQL, чтобы пометить ссылку на переменную в DML операторе. Его использование было расширено в Firebird 3 для двух независимых целей:

1. Позволить OLD/NEW полям в курсорах быть прочитанными и присваивать их переменным.
2. Сделать присвоение переменной в DML и PSQL операторах в модулях и блоках более гибким, и при необходимости разрешать неоднозначность между именами полей и именами переменных.

Теперь подобный синтаксис не вызывает ошибки.

```
create trigger t1 before insert on t1
as
declare v integer;
begin
  :v = :old.n;
  :new.n = :v;
end
```

Ссылки на PSQL курсоры как на переменные

Теперь в PSQL поддерживаются ссылки на курсоры, как на переменные типа запись. Текущая запись доступна через имя курсора для явных (DECLARE AS CURSOR) и неявных (FOR SELECT) PSQL курсоров, что делает необязательным предложение INTO.

Правила:

1. Для разрешения неоднозначности при доступе к переменной курсора перед именем курсора необходим префикс двоеточие.
2. К переменной курсора можно получить доступ без префикса двоеточия, но в этом случае, в зависимости от области видимости контекстов, существующих в запросе, имя может разрешиться как контекст запроса вместо курсора.
3. Переменные курсора доступны только для чтения.
4. В операторе FOR SELECT без предложения AS CURSOR необходимо использовать предложение INTO. Если указано предложение AS CURSOR, предложение INTO не требуется, но разрешено.
5. В операторе FETCH предложение INTO необязательное.
6. Чтение из переменной курсора возвращает текущие значения полей. Это означает, что оператор UPDATE (с предложением WHERE CURRENT OF) обновит также и значения полей в переменной курсора для последующих чтений. Выполнение оператора DELETE (с предложением WHERE CURRENT OF) установит NULL для значений полей переменной курсора для последующих чтений.

Замечание:

Обратите внимание на использование двоеточие (:) в качестве префикса к ссылке на поле курсора. Ранее, префикс двоеточия использовался в PSQL только, чтобы пометить использование переменной или параметра в регулярном SQL-

операторе. В Firebird 3 использование двоеточия расширено для реализации механизма обработки непеременных, как будто они являются переменными.

Примеры:

1. Использование явно объявленного курсора как курсорной переменной

```
execute block returns (o char(31))
as
  declare c cursor for (
    select rdb$relation_name name
    from rdb$relations
  );
begin
  open c;
  while (1 = 1) do
  begin
    fetch c;

    if (row_count = 0) then
      leave;

    o = c.name;
    suspend;
  end
  close c;
end
```

2. Использование неявно объявленного курсора как курсорной переменной

```
execute block returns (o char(31))
as
begin
  for select rdb$relation_name name
      from rdb$relations
      as cursor c
  do
  begin
    o = c.name;
    suspend;
  end
end
```

3. Разрешение неоднозначностей курсорной переменной внутри запросов

```
execute block returns (o1 char(31), o2 char(31))
as
begin
  for
```

```
    select rdb$relation_name
    from rdb$relations
    where rdb$relation_name = 'RDB$RELATIONS'
    as cursor c
  do
  begin
    for
      select
        -- с префиксом разрешается как курсор
        :c.rdb$relation_name x1,
        -- без префикса как псевдоним таблицы rdb$relations
        c.rdb$relation_name x2
      from rdb$relations c
      where rdb$relation_name = 'RDB$DATABASE'
      as cursor d
    do
    begin
      o1 = d.x1;
      o2 = d.x2;
      suspend;
    end
  end
end
```

4. Анонимный PSQL блок собирающий скрипты создания представлений

```
EXECUTE BLOCK
RETURNS (
    SCRIPT BLOB SUB_TYPE TEXT)
AS
    DECLARE VARIABLE FIELDS VARCHAR(8191);
BEGIN
  FOR
      SELECT
          RDB$RELATION_NAME,
          RDB$VIEW_SOURCE
      FROM
          RDB$RELATIONS
      WHERE RDB$VIEW_SOURCE IS NOT NULL
      AS CURSOR CUR_R
  DO
  BEGIN
    FOR
        SELECT
            RDB$FIELD_NAME
        FROM
            RDB$RELATION_FIELDS
        WHERE RDB$RELATION_NAME = :CUR_R.RDB$RELATION_NAME
        AS CURSOR CUR_F
    DO
    BEGIN
```

```
        IF (FIELDS IS NULL) THEN
          FIELDS = TRIM(CUR_F.RDB$FIELD_NAME);
        ELSE
          FIELDS = FIELDS || ', '
                   || TRIM(CUR_F.RDB$FIELD_NAME);
      END
      SCRIPT = 'CREATE VIEW ' || CUR_R.RDB$RELATION_NAME;
      IF (FIELDS IS NOT NULL) THEN
        SCRIPT = SCRIPT || ' (' || FIELDS || ')';
      SCRIPT = SCRIPT || ' AS ' || ASCII_CHAR(13);
      SCRIPT = SCRIPT || CUR_R.RDB$VIEW_SOURCE;
      SUSPEND;
    END
  END
```

Двунаправленные (прокручиваемые) курсоры

В Firebird 3 появилась возможность объявление в заголовке PSQL модуля (хранимые процедуры, хранимые функции, триггеры, анонимные PSQL блоки) двунаправленных курсоров. Такие курсоры позволяют двигаться по набору данных не только вперёд, но и назад, а также на N позиций относительно текущего положения. Двунаправленные курсоры используют буферизацию записей выборки, поэтому они работают несколько медленней однонаправленных курсоров.

Объявление двунаправленного курсора

Доступно: PSQL

Синтаксис (полный):

```
DECLARE [VARIABLE] <cursor_name> [SCROLL | NO SCROLL]
CURSOR FOR (<select_statement>);
```

Аргумент	Описание
cursor_name	Имя курсора.
select_statement	Оператор SELECT.

Описание:

Оператор DECLARE … CURSOR FOR объявляет именованный курсор, связывая его с набором данных, полученного в операторе SELECT, указанного в предложении CURSOR FOR. Необязательное предложение SCROLL делает курсор двунаправленным (прокручиваемым), предложение NO SCROLL – однонаправленным. По умолчанию курсоры являются однонаправленными.

Примечания:

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

Оператор FETCH

Доступно: PSQL

Синтаксис (полный):

```
FETCH <cursor_name> [INTO [:]<var_name> [, [:]<var_name> ...]];

или

FETCH {
  NEXT  |
  PRIOR |
  FIRST |
  LAST  |
  ABSOLUTE <n> |
  RELATIVE <n>
} FROM <cursor_name> [INTO [:]<var_name> [,[:]<var_name> ...]];
```

Аргумент	Описание
cursor_name	Имя курсора.
var_name	PSQL переменная.
n	Целое число.

Описание:

Оператор FETCH выбирает следующую строку данных из результирующего набора данных курсора и присваивает значения столбцов в переменные PSQL.

Оператор FETCH применим только к курсорам, объявленным в операторе DECLARE VARIABLE.

Во второй версии оператора FETCH вы можете указывать в каком направлении и на сколько записей продвинется позиция курсора. Предложение NEXT указывает, что указатель курсора должен продвинуться на 1 запись вперёд. Это предложение допустимо использовать как с прокручиваемыми, там и не прокручиваемыми курсорами. Остальные предложения допустимо использовать только с прокручиваемыми курсорами. Предложение PRIOR указывает, что указатель курсора должен продвинуться на 1 запись назад. Предложение FIRST позволяет переместить позицию курсора на первую запись, а предложение LAST – на последнюю. Предложение ABSOLTE позволяет указать номер позиции, на которую будет установлен курсор. Номер позиции должен быть в диапазоне от 1 до максимального количества записей извлекаемых запросом курсора. Предложение RELATIVE позволяет указать на какое количество записей относительно текущей позиции необходимо переместить указатель курсора. Если указано положительное число, то курсор перемещает вперёд на N позиций, если отрицательное, то назад.

Необязательное предложение INTO помещает данные из текущей строки курсора в PSQL переменные.

Для проверки того, что записи набора данных исчерпаны, используется контекстная переменная ROW_COUNT, которая возвращает количество считанных оператором строк. Если произошло чтение очередной записи из набора данных, то ROW_COUNT равняется единице, иначе нулю.

Примеры:

Пример работы с прокручиваемым курсором.

```
EXECUTE BLOCK
RETURNS (
    N INT,
    RNAME CHAR(31))
AS
DECLARE C SCROLL CURSOR FOR (
    SELECT
        ROW_NUMBER() OVER(ORDER BY RDB$RELATION_NAME) AS N,
        RDB$RELATION_NAME
    FROM
        RDB$RELATIONS
    ORDER BY RDB$RELATION_NAME);
BEGIN
  OPEN C;
  -- перемещаемся на первую запись (N=1)
  FETCH FIRST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на 1 запись вперёд (N=2)
```

```
  FETCH NEXT FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на пятую запись (N=5)
  FETCH ABSOLUTE 5 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на 1 запись назад (N=4)
  FETCH PRIOR FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на 3 записи вперёд (N=7)
  FETCH RELATIVE 3 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на 5 записей назад (N=2)
  FETCH RELATIVE -5 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на первую запись (N=1)
  FETCH FIRST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- перемещаемся на последнюю
  FETCH LAST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  CLOSE C;
END
```

Исключения с параметрами

Исключение теперь могут быть определены с сообщением, содержащим слоты для параметров, которые заполняются при возбуждении исключения.

Синтаксис:

```
EXCEPTION <name> USING (<value list>);

<value list> := <val> [, <val> [, <val> …]]
```

Замечания:

Статус вектор генерируется, используя комбинацию кодов isc_except, <exception number>, isc_formatted_exception, <formatted exception message>, <exception parameters>.

Поскольку используется новый код ошибки (isc_formatted_exception), клиент должен быть версии 3.0 или по крайней мере использовать firebird.msg от версии 3.0 для того чтобы правильно преобразовать статус вектор в строку.

Параметры рассматриваются слева направо. Каждый параметр передаётся в оператор возбуждающий исключение как N-ый, N начинается с 1:

- Если N-ый параметр не передан, его слот не заменяется;
- Если передано значение NULL, слот будет заменён на строку '***null***';
- Если количество передаваемых параметров будет больше, чем содержится в сообщении исключения, то лишние будут проигнорированы;
- Общая длина сообщения, включая значения параметров, все еще ограничена 1053 байтами.

Примеры:

1. Использование параметризованного исключения

```
create exception e_invalid_val
  'Invalid value @1 for the field @2';

...
if (val < 1000) then
  thing = val;
else
  exception e_invalid_val using (val, 'thing');
end
```

2. Использование параметризованного исключения в DDL триггере

```
CREATE EXCEPTION EX_BAD_SP_NAME
  'Name of procedures must start with ''@1'' : ''@2''';

...
CREATE TRIGGER TRG_SP_CREATE BEFORE CREATE PROCEDURE
AS
  DECLARE SP_NAME VARCHAR(255);
BEGIN
  SP_NAME = RDB$GET_CONTEXT('DDL_TRIGGER', 'OBJECT_NAME');
  IF (SP_NAME NOT STARTING 'SP_')
  THEN
    EXCEPTION EX_BAD_SP_NAME USING ('SP_', SP_NAME);
END^
```

Оператор CONTINUE

Оператор досрочного начала новой итерации цикла.

Доступно: PSQL

Синтаксис:

```
[label:]
{FOR <select_stmt> | WHILE (<condition>)}
DO
BEGIN
…
CONTINUE [label];
…
END
```

Аргумент	Описание
label	Метка.
select_stmt	Оператор SELECT.
condition	Логическое условие возвращающее TRUE, FALSE или UNKNOWN.

Описание:

Оператор CONTINUE моментально начинает новую итерацию внутреннего цикла операторов WHILE или FOR. С использованием опционального параметра метки LEAVE также может начинать новую итерацию для внешних циклов.

Примеры:

```
FOR
  SELECT A, D FROM ATABLE INTO :achar, :ddate
DO BEGIN
  IF (ddate < current_data - 30) THEN
    CONTINUE;
  ELSE
    /* do stuff */
  ...
END
```

SQLSTATE в обработчиках исключений

В качестве условия перехвата исключений теперь можно использовать SQLSTATE в операторе WHEN … DO.

Доступно: PSQL

Синтаксис:

```
WHEN {<error> [, <error> …] | ANY}
```

```
DO <compound_statement>

<error> ::= {
  EXCEPTION exception_name
| SQLCODE number
| GDSCODE errcode
| SQLSTATE 'sqlstate_code'}
```

Аргумент	Описание
exception_name	Имя исключения.
number	Код ошибки SQLCODE.
errcode	Символическое имя ошибки GDSCODE
sqlstate_code	Код ошибки SQLSTATE.
compound_statement	Оператор или блок операторов.

Примеры:

```
EXECUTE BLOCK
AS
    DECLARE VARIABLE I INT;
BEGIN
  BEGIN
    I = 1 / 0;
    WHEN SQLSTATE '22003' DO
      EXCEPTION E_CUSTOM_EXCEPTION
        'Numeric value out of range.';
    WHEN SQLSTATE '22012' DO
      EXCEPTION E_CUSTOM_EXCEPTION 'Division by zero.';
    WHEN SQLSTATE '23000' DO
      EXCEPTION E_CUSTOM_EXCEPTION
        'Integrity constraint violation.';
  END
END
```

Стабильность PSQL курсоров

Теперь PSQL курсоры, не содержащие оператор SUSPEND, стабильны.

```
FOR
  SELECT ID
  FROM T
  WHERE VAL IS NULL
  INTO :ID
DO BEGIN
  UPDATE T SET VAL = 1
  WHERE ID = :ID;
END
```

Ранее этот блок выполнялся бы в бесконечном цикле. Теперь, цикл не будет выбрать значение, если оно было установлено внутри цикла.

Замечание:

Это может изменить поведение унаследованного кода.

Если внутри блока есть оператор SUSPEND, старая нестабильность остается, следующий запрос все еще производит бесконечный цикл:

```
FOR SELECT ID FROM T INTO :ID
DO BEGIN
  INSERT INTO T (ID) VALUES (:ID);
  SUSPEND;
END
```

Удалены некоторые ограничения на размеры при использовании нового API

Только при использовании нового API:

- Размер тела хранимой процедуры или триггера может превышать традиционный предел в 32 КБ. Теоретический предел, обеспечиваемый новым API, составляет 4 ГБ. На данный момент, по соображениям безопасности, установлен предел 10MB.
- Общий размер всех входных или выходных параметров для хранимой процедуры или определяемого пользователем DSQL запроса больше не ограничивается традиционным размером (64K - overhead).

Новое в безопасности

Размещение списка пользователей

Теперь Firebird поддерживает неограниченное количество баз данных безопасности. Любая база данных может выступать в качестве базы данных безопасности, кроме того любая база данных может быть базой данных безопасности для самой себя.

Используйте файл database.conf чтобы установить базу данных безопасности отличную от той, что установлена по умолчанию. В следующем примере /mnt/storage/private.security.fdb установлена как база данных безопасности для баз first и second:

```
first = /mnt/storage/first.fdb
{
  SecurityDatabase = /mnt/storage/private.security.fdb
}

second = /mnt/storage/second.fdb
{
  SecurityDatabase = /mnt/storage/private.security.fdb
}
```

В этом примере база данных third является базой данных безопасности для самой себя:

```
third = /mnt/storage/third.fdb
{
  SecurityDatabase = third
}
```

Примечание.

Значением параметра SecurityDatabase может быть псевдоним базы данных или фактический путь к базе данных.

Управление пользователями с помощью SQL

Начиная с Firebird 2.5 были введены SQL операторы для управления пользователями. Firebird 3 расширяет эти команды следующими возможностями:

- Позволяется явно указывать с помощью какого плагина управления пользователями создавать учётную запись, из какого плагина модифицировать его или удалять.
- Добавление неограниченного множества пользовательских атрибутов для каждого пользователя (атрибуты TAGS).
- Включение и выключение пользователя без его удаления (атрибуты ACTIVE/INACTIVE).
- Изменение текущего пользователя ALTER CURRENT USER …
- Позволяется добавлять комментарии для пользователя

Важно.

Начиная с Firebird 3.0, поддерживается работа с несколькими базами данных безопасности и плагинами управления пользователями. Эта возможность не поддерживается gsec утилитой или сервисами API. Оба этих метода считаются устаревшими.

CREATE USER

Создание учётной записи пользователя Firebird.

Доступно: DSQL

Синтаксис:

```
CREATE USER username PASSWORD 'password'
[FIRSTNAME 'firstname']
[MIDDLENAME 'middlename']
[LASTNAME 'lastname']
[ACTIVE | INACTIVE]
[USING PLUGIN 'pluginname']
[TAGS (<tag> [, <tag> [, <tag> ...]] )]
[GRANT ADMIN ROLE];

<tag> ::= tagname = 'string value'
```

Аргумент	Описание
username	Имя пользователя. Максимальная длина 31 символ.
password	Пароль пользователя. Может включать в себя до 32 символов. Чувствительно к регистру.
firstname	Вспомогательная информация: имя пользователя. Максимальная длина 32 символа.
middlename	Вспомогательная информация: «второе имя» (отчество, «имя отца») пользователя. Максимальная длина 32 символа.
lastname	Вспомогательная информация: фамилия пользователя. Максимальная длина 32 символа.
tagname	Имя пользовательского атрибута. Максимальная длина 31 символ. Имя атрибута должно подчиняться правилам наименования SQL

	идентификаторов.
string value	Значение пользовательского атрибута. Максимальная длина 255 символов.
pluginname	Имя плагина управления пользователями, в котором необходимо создать нового пользователя.

Описание:

Оператор CREATE USER создаёт учётную запись пользователя Firebird. Пользователь должен отсутствовать в текущей базе данных безопасности Firebird иначе будет выдано соответствующее сообщение об ошибке.

Предложение PASSWORD задаёт пароль пользователя. Максимальная длина пароля зависит от того какой менеджер пользователей задействован (параметр UserManager). Для менеджера пользователей Srp максимальная длина пароля составляет 20 символов, для Legacy_UserManager – 8 символов.

Необязательные предложения FIRSTNAME, MIDDLENAME и LASTNAME задают дополнительные атрибуты пользователя, такие как имя пользователя (имя человека), отчество и фамилия соответственно.

Кроме того вы можете задать неограниченное количество пользовательских атрибутов с помощью необязательного предложения TAGS.

Если при создании учётной записи будет указан атрибут INACTIVE, то пользователь будет создан в «неактивном состоянии», т.е. подключиться с его учётной записью будет невозможно. При указании атрибута ACTIVE пользователь будет создан в активном состоянии. По умолчанию пользователь создаётся активным.

Если указана опция GRANT ADMIN ROLE, то новая учётная запись пользователя создаётся с правами роли RDB$ADMIN в текущей базе данных безопасности. Это позволяет вновь созданному пользователю управлять учётными записями пользователей, но не даёт ему специальных полномочий в обычных базах данных.

Необязательное предложение USING PLUGIN позволяет явно указывать какой плагин управления пользователями будет использован. По умолчанию используется тот плагин, который был указан первым в списке параметра UserManager в файле конфигурации firebird.conf. Допустимыми являются только значения, перечисленные в параметре UserManager.

Важно.

Учтите что одноимённые пользователи, созданные с помощью разных плагинов управления пользователями - это разные пользователи. Поэтому пользователя созданного с помощью одного плагина управления пользователями можно удалить или изменить, указав только тот же самый плагин.

Для создания учётной записи пользователя текущий пользователь должен обладать административными привилегиями.

Примеры:

1. Создание пользователя с именем bigshot.

```
CREATE USER bigshot PASSWORD 'buckshot';
```

2. Создание пользователя с именем godzilla с помощью плагина управления пользователями Legacy_UserManager.

```
CREATE USER godzilla PASSWORD 'robot'
USING PLUGIN Legacy_UserManager;
```

3. Создание пользователя John с дополнительными атрибутами (именем и фамилией).

```
CREATE USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe';
```

4. Создание пользователя John с дополнительными атрибутами (именем и фамилией) и пользовательскими атрибутами.

```
CREATE USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe'
TAGS (BIRTHYEAR = '1970', CITY = 'New York');
```

5. Создание пользователя John в неактивном состоянии.

```
CREATE USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe'
INACTIVE;
```

6. Создание пользователя superuser с возможностью управления пользователями.

```
CREATE USER superuser PASSWORD 'kMn8Kjh'
GRANT ADMIN ROLE;
```

См. также ALTER USER, CREATE OR ALTER USER, DROP USER

ALTER USER

Изменение учётной записи пользователя Firebird.

Доступно: DSQL

Синтаксис:

```
ALTER {USER username | CURRENT USER}
{
  [SET]
  [PASSWORD 'password']
  [FIRSTNAME 'firstname']
  [MIDDLENAME 'middlename']
  [LASTNAME 'lastname']
  [ACTIVE | INACTIVE]
  [TAGS (<tag> | DROP tagname [, <tag> | DROP tagname ...])]
}
[USING PLUGIN pluginname]
[{GRANT | REVOKE} ADMIN ROLE];


<tag> ::= tagname = 'string value'
```

Аргумент	Описание
username	Имя пользователя. Максимальная длина 31 символ.
password	Пароль пользователя. Может включать в себя до 32 символов. Чувствительно к регистру.
firstname	Вспомогательная информация: имя пользователя. Максимальная длина 32 символа.
middlename	Вспомогательная информация: «второе имя» (отчество, «имя отца») пользователя. Максимальная длина 32 символа.
lastname	Вспомогательная информация: фамилия пользователя. Максимальная длина 32 символа.
tagname	Имя пользовательского атрибута. Максимальная длина 31 символ. Имя атрибута должно подчиняться правилам наименования SQL идентификаторов.
string value	Значение пользовательского атрибута. Максимальная длина 255 символов.
pluginname	Имя плагина управления пользователями, в котором был создан данный пользователь

Описание:

Оператор ALTER USER изменяет данные учётной записи пользователя. В операторе ALTER USER должен присутствовать хотя бы одно из необязательных предложений.

Необязательное предложение PASSWORD задаёт новый пароль пользователя. Необязательные предложения FIRSTNAME, MIDDLENAME и LASTNAME позволяют изменить дополнительные атрибуты пользователя, такие как имя пользователя (имя человека), отчество и фамилия соответственно.

Атрибут INACTIVE позволяет сделать учётную запись неактивной. Это удобно когда необходимо временно отключить учётную запись без её удаления.

Атрибут ACTIVE позволяет вернуть неактивную учётную запись в активное состояние.

Необязательное предложение TAGS позволяет задать, изменить или удалить пользовательские атрибуты. Если в списке атрибутов, атрибута с заданным именем не было, то он будет добавлен, иначе его значение будет изменено. Атрибуты не указанные в списке не будут изменены. Для удаления пользовательского атрибута перед его именем в списке атрибутов необходимо указать ключевое слово DROP.

Предложение GRANT ADMIN ROLE предоставляет указанному пользователю привилегии роли RDB$ADMIN в текущей базе данных безопасности. Это позволяет указанному пользователю управлять учётными записями пользователей, но не даёт ему специальных полномочий в обычных базах данных.

Предложение REVOKE ADMIN ROLE отбирает у указанного пользователя привилегии роли RDB$ADMIN в текущей базе данных безопасности. Это запрещает указанному пользователю управлять учётными записями пользователей.

Необязательное предложение USING PLUGIN позволяет явно указывать какой плагин управления пользователями будет использован. По умолчанию используется тот плагин, который был указан первым в списке параметра UserManager в файле конфигурации firebird.conf. Допустимыми являются только значения, перечисленные в параметре UserManager.

Важно.

Учтите что одноимённые пользователи, созданные с помощью разных плагинов управления пользователями - это разные пользователи. Поэтому пользователя созданного с помощью одного плагина управления пользователями можно удалить или изменить, указав только тот же самый плагин.

Если требуется изменить свою учётную запись, то вместо указания имени текущего пользователя можно использовать предложение CURRENT USER.

Для модификации чужой учётной записи пользователя текущий пользователь должен обладать административными привилегиями. Свои собственные учётные записи могут изменять любые пользователи, однако это не относится к опциям GRANT/REVOKE ADMIN ROLE для изменения которых, необходимы административные привилегии.

Примеры:

1. Изменение пароля пользователя bobby и выдача ему привилегии управления пользователями.

```
ALTER USER bobby PASSWORD '67-UiT_G8'
GRANT ADMIN ROLE;
```

2. Изменение пароля пользователя Godzilla, созданного с помощью плагина управления пользователями Legacy_UserManager

```
ALTER USER godzilla PASSWORD 'robot12'
USING PLUGIN Legacy_UserManager;
```

3. Изменение дополнительных атрибутов (имени и фамилии) пользователя dan.

```
ALTER USER dan
FIRSTNAME 'No_Jack'
LASTNAME 'Kennedy';
```

4. Изменение дополнительных атрибутов своей учётной записи.

```
ALTER CURRENT USER
FIRSTNAME 'No_Jack'
LASTNAME 'Kennedy';
```

5. Отключение пользователя dan.

```
ALTER USER dan INACTIVE;
```

6. Отбор привилегии управления пользователями у пользователя dumbbell.

```
ALTER USER dumbbell
REVOKE ADMIN ROLE;
```

7. Изменение пользовательских атрибутов своей учётной записи. Атрибуту BIRTHDAY будет установлено новое значение, а атрибут CITY будет удалён.

```
CREATE CURRENT USER
TAGS (BIRTHYEAR = '1971', DROP CITY);
```

См. также CREATE USER, CREATE OR ALTER USER, DROP USER

CREATE OR ALTER USER

Создание или изменение учётной записи пользователя Firebird.

Доступно: DSQL

Синтаксис:

```
CREATE OR ALTER USER username
{
  [SET]
  [PASSWORD 'password']
  [FIRSTNAME 'firstname']
  [MIDDLENAME 'middlename']
  [LASTNAME 'lastname']
  [ACTIVE | INACTIVE]
  [TAGS (<tag> | DROP tagname [, <tag> | DROP tagname ...])]
}
[USING PLUGIN pluginname]
[{GRANT | REVOKE} ADMIN ROLE];


<tag> ::= tagname = 'string value'
```

Аргумент	Описание
username	Имя пользователя. Максимальная длина 31 символ.
password	Пароль пользователя. Может включать в себя до 32 символов. Чувствительно к регистру.
firstname	Вспомогательная информация: имя пользователя. Максимальная длина 32 символа.
middlename	Вспомогательная информация: «второе имя» (отчество, «имя отца») пользователя. Максимальная длина 32 символа.
lastname	Вспомогательная информация: фамилия пользователя. Максимальная длина 32 символа.
tagname	Имя пользовательского атрибута. Максимальная длина 31 символ. Имя атрибута должно подчиняться правилам наименования SQL идентификаторов.
string value	Значение пользовательского атрибута. Максимальная длина 255 символов.
pluginname	Имя плагина управления пользователями, в котором необходимо создать нового пользователя или в котором был ранее создан данный пользователь

Описание:

Оператор CREATE OR ALTER USER создает новую или изменяет учётную запись. Если пользователя не существует, то он будет создан с использованием предложения CREATE USER. Если он уже существует, то он будет изменен, при этом существующие привилегии сохраняются.

Примеры:

Создание или изменение пользователя john.

```
CREATE OR ALTER USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe'
INACTIVE;
```

См. также CREATE USER, ALTER USER

DROP USER

Удаление учётной записи пользователя Firebird.

Доступно: DSQL

Синтаксис:

```
DROP USER username
[USING PLUGIN pluginname]
```

Аргумент	Описание
username	Имя пользователя. Максимальная длина 31 символ.
pluginname	Имя плагина управления пользователями, в котором был создан данный пользователь

Описание:

Оператор DROP USER удаляет учётную запись пользователя Firebird.

Необязательное предложение USING PLUGIN позволяет явно указывать какой плагин управления пользователями будет использован. По умолчанию используется тот плагин, который был указан первым в списке параметра UserManager в файле конфигурации firebird.conf. Допустимыми являются только значения, перечисленные в параметре UserManager.

Важно.

Учтите что одноимённые пользователи, созданные с помощью разных плагинов управления пользователями - это разные пользователи. Поэтому пользователя созданного с помощью одного плагина управления пользователями можно удалить или изменить, указав только тот же самый плагин.

Для удаления учётной записи пользователя текущий пользователь должен обладать административными привилегиями.

Примеры:

1. Удаление пользователя bobby.

```
DROP USER bobby;
```

2. Удаление пользователя Godzilla, созданного с помощью плагина управления пользователями Legacy_UserManager.

```
DROP USER Godzilla USING PLUGIN Legacy_UserManager;
```

См. также CREATE USER, ALTER USER

COMMENT ON USER

Добавление комментария для учётной записи пользователя.

Доступно: DSQL

Синтаксис:

```
COMMENT ON USER username IS 'text'
```

Аргумент	Описание
username	Имя пользователя. Максимальная длина 31 символ.
text	Текст комментария

Описание:

Оператор COMMENT ON USER добавляет комментарии для учётных данных пользователей. В комментариях вы можете добавлять очень длинные текстовые примечания.

Примеры:

1. Добавление комментария для текущей базы данных

```
COMMENT ON USER John IS 'Это пользователь Джон';
```

Псевдо таблицы со списком пользователей

Для получения списка пользователей и их атрибутов были введены две новые виртуальные таблицы SEC$USERS и SEC$USER_ATTRIBUTES.

Важно.

Эта функция во многом зависит от плагина управления пользователями. Имейте в виду, что некоторые опции игнорируются при использовании устаревшего плагина управления пользователями.

Эти псевдо-таблицы подобны таблицам семейства MON$, используемых для мониторинга сервера. Таблицы создаются по требованию при запуске оператора

SELECT * FROM SEC$USERS

или

SELECT * FROM SEC$USER_ATTRIBUTES

Запрос выводит список пользователей (или их атрибутов) из базы данных безопасности, сконфигурированной для текущей базы данных и доступных для управления данным пользователем.

SEC$USERS

Таблица «содержит» список (или их атрибутов) из базы данных безопасности, сконфигурированной для текущей базы данных и доступных для управления данным пользователем.

Имя поля	Тип	Описание
SEC$USER_NAME	CHAR(31)	Имя пользователя
SEC$FIRST_NAME	VARCHAR(32)	Первое имя (имя).
SEC$MIDDLE_NAME	VARCHAR(32)	Среднее имя (отчество).
SEC$LAST_NAME	VARCHAR(32)	Последнее имя (фамилия).
SEC$ACTIVE	BOOLEAN	Флаг активности пользователя.
SEC$ADMIN	BOOLEAN	Отражает, имеет ли пользователь права RDB$ADMIN в базе данных безопасности.
SEC$DESCRIPTION	BLOB SUB_TYPE TEXT	Комментарий к пользователю.

SEC$USER_ATTRIBUTES

Таблица «содержит» атрибуты пользователей из базы данных безопасности, сконфигурированной для текущей базы данных и доступных для управления данным пользователем.

Имя поля	Тип	Описание
SEC$USER_NAME	CHAR(31)	Имя пользователя
SEC$KEY	VARCHAR(31)	Имя атрибута.
SEC$VALUE	VARCHAR(255)	Значение атрибута.

SET ROLE

Согласно стандарту SQL-2008 оператор SET ROLE установить контекстной переменной CURRENT ROLE одну из назначенных ролей для пользователя

CURRENT_USER или роль, полученную в результате доверительной аутентификации (в этом случае оператор принимает вид SET TRUSTED ROLE).

Доступно: DSQL

Синтаксис:

```
SET ROLE rolename
```

Аргумент	Описание
rolename	Имя роли.

Примеры:

Установить для текущего пользователя роль manager.

```
SET ROLE manager;

select current_role from rdb$database;

ROLE
=======================
MANAGER
```

SET TRUSTED ROLE

Идея отдельной команды SET TRUSTED ROLE состоит в том, чтобы при подключении доверенного пользователя не указывать никакой дополнительной информации о роли, SET TRUSTED ROLE делает доверенную роль (если таковая существует) текущей ролью без дополнительной деятельности, связанной с установкой параметров DBP.

Доверенная роль это не специальный тип роли, ей может быть любая роль созданная с помощью оператора CREATE ROLE или предопределённая системная роль RDB$ADMIN. Она становится доверенной ролью для подключения, когда подсистема отображения объектов безопасности (security objects mapping subsystem) находит соответствие между результатом аутентификации, полученным от плагина и локальным или глобальным отображением (mapping) для текущей базы данных. Роль даже может быть той, которая не предоставлена явно этому доверенному пользователю.

Примечания.

- Доверенная роль не назначается при подключении по умолчанию. Можно изменить это поведение, используя соответствующий плагин аутентификации и команды CREATE/ALTER MAPPING.
- В то время как CURRENT_ROLE возможно изменить с помощью оператора SET ROLE, её не всегда возможно вернуть обратно той же командой, потому что она выполняет проверку прав доступа.

Доступно: DSQL

Синтаксис:

Включает доступ доверенной роли, при условии, что CURRENT_USER получен с помощью доверительной аутентификации и роль доступна.

```
SET TRUSTED ROLE
```

Примером использования доверенной роли является назначение системной роли RDB$ADMIN для администраторов Windows, когда используется доверительная аутентификация Windows.

Новые права на объекты базы данных

Привилегия EXECUTE для функций и пакетов

Привилегия EXECUTE теперь поддерживается для PSQL функций, пакетов и унаследованных внешних функций (DECLARE EXTERNAL FUNCTION). Привилегия может быть назначена только для всего пакета, а не для отдельных его подпрограмм.

Доступно: DSQL

Синтаксис:

```
GRANT EXECUTE ON {
  PROCEDURE procname |
  FUNCTION funcname |
  PACKAGE packagename
}
TO {<object_list> | <user_list> [WITH GRANT OPTION]}
[{GRANTED BY | AS} [USER] grantor];

REVOKE [GRANT OPTION FOR] EXECUTE ON {
  PROCEDURE procname |
  FUNCTION funcname |
  PACKAGE packagename
}
FROM {<object_list> | <user_list>}
[{GRANTED BY | AS} [USER] grantor];


<object_list> ::= {
```

```
    PROCEDURE procname |
    FUNCTION funcname |
    PACKAGE packagename |
    TRIGGER trigname |
    VIEW viewname |
    PUBLIC
  } [, <object_list> …]

  <user_list> ::= {
    [USER] username |
    [ROLE] rolename |
    Unix_user
  } [, <user_list> …]
```

Аргумент	Описание
procname	Имя хранимой процедуры, для которой назначается или отбирается привилегия EXECUTE, или которой будут даны или отобраны привилегии.
funcname	Имя PSQL или внешней функции, для которой назначается или отбирается привилегия EXECUTE, или которой будут даны или отобраны привилегии.
packagename	Имя пакета, для которого назначается или отбирается привилегия EXECUTE, или которому будут даны или отобраны привилегии.
Unix_group	Имя группы пользователей в операционных системах семейства UNIX.
username	Имя пользователя, которому будут даны или отобраны привилегии.
rolename	Имя роли, которой будут даны или отобраны привилегии.
viewname	Имя представления, которому будут даны или отобраны привилегии.
trigname	Имя триггера, которому будут даны или отобраны привилегии.
grantor	Пользователь от имени, которого предоставляются или отбираются привилегии.

Описание (не полное):

Оператор GRANT предоставляет одну или несколько привилегий для объектов базы данных пользователям, ролям, хранимым процедурам, хранимым или внешним функциям, пакетам, триггерам и представлениям.

Оператор REVOKE отменяет привилегии для пользователей, ролей, хранимых процедур, хранимых или внешних функций, пакетов, триггеров и представлений выданные оператором GRANT.

Авторизованный пользователь не имеет никаких привилегий до тех пор, пока какие либо права не будут предоставлены ему явно. При создании объекта только его создатель и SYSDBA имеет привилегии на него и может назначать привилегии другим пользователям, ролям или объектам.

Для выполнения хранимых процедур, функций существует отдельная привилегия EXECUTE. Она позволяет выполнять хранимые процедуры или функции и делать выборку данных из процедур выбора (с помощью оператора

SELECT). Для упакованных процедур и функций, невозможно назначить отдельные привилегии, привилегия может быть предоставлена только для пакета в целом, при этом она распространяется на все процедуры и функции пакета.

Начиная с Firebird 3 для рекурсивных процедур (функций) привилегия EXECUTE не требуется для вызова самой хранимой процедуру (функции).

Необязательное предложение WITH GRANT OPTION оператора GRANT позволяет пользователям, указанным в списке пользователей, передавать другим пользователям привилегии указанные в списке привилегий.

Необязательное предложение GRANT OPTION FOR оператора REVOKE отменяет для соответствующего пользователя или роли право предоставления другим пользователям или ролям указанную привилегию.

Примеры:

1. Выдача привилегий на выполнении функции GET_BEGIN_DATE для пользователя вошедшего (или установившего позже) под ролью MANAGER.

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE TO ROLE MANAGER;
```

2. Выдача привилегий на выполнение всех публичных процедур и функций пакета APP_VAR для всех пользователей.

```
GRANT EXECUTE ON PACKAGE APP_VAR TO PUBLIC;
```

3. Выдача привилегий на выполнение функции GET_BEGIN_DATE для пакета APP_VAR.

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE
TO PACKAGE APP_VAR;
```

4. Отбираем привилегии на выполнение функции GET_BEGIN_DATE и право передавать эту привилегию у роли MANAGER.

```
REVOKE GRANT OPTION FOR
EXECUTE ON FUNCTION GET_BEGIN_DATE
FROM ROLE MANAGER;
```

Привилегии для защиты других объектов метаданных

Новая SQL-2008 совместимая привилегия USAGE введена для защиты объектов метаданных, отличных от таблиц, представлений, хранимых процедур и функций, триггеров и пакетов.

Примечание.

В Firebird 3 привилегия USAGE проверяется только для исключений (exception) и генераторов/последовательностей (в **gen_id**(*gen_name*, 1) или **next value for** *gen_name*). Привилегии для других объектов метаданных могут быть включены в следующих релизах, если покажется целесообразным.

Доступно: DSQL

Синтаксис:

```
GRANT USAGE ON {
  DOMAIN domainname |
  EXCEPTION exceptionname |
  {GENERATOR | SEQUENCE} generatorname |
  CHARACTER SET charsetname
  COLLATION collationname
}
TO {<object_list> | <user_list> [WITH GRANT OPTION]}
[{GRANTED BY | AS} [USER] grantor];

REVOKE [GRANT OPTION FOR] USAGE ON {
  DOMAIN domainname |
  EXCEPTION exceptionname |
  {GENERATOR | SEQUENCE} generatorname |
  CHARACTER SET charsetname
  COLLATION collationname
}
FROM {<object_list> | <user_list>}
[{GRANTED BY | AS} [USER] grantor];

<object_list> ::= {
  PROCEDURE procname |
  FUNCTION funcname |
  PACKAGE packagename |
  TRIGGER trigname |
  VIEW viewname |
  PUBLIC
} [, <object_list> …]

<user_list> ::= {
  [USER] username |
  [ROLE] rolename |
  Unix_user
} [, <user_list> …]
```

Аргумент	Описание
domainname	Имя домена, для которого назначается или отнимается привилегия USAGE.
exceptionname	Имя исключения, для которого назначается или отнимается привилегия USAGE.

generatorname	Имя генератора или последовательности, для которого назначается или отнимается привилегия USAGE.
charsetname	Имя набора символов, для которого назначается или отнимается привилегия USAGE.
collationname	Имя сортировки, для которого назначается или отнимается привилегия USAGE.
Unix_group	Имя группы пользователей в операционных системах семейства UNIX.
username	Имя пользователя, которому будут даны или отобраны привилегии.
rolename	Имя роли, которой будут даны или отобраны привилегии.
viewname	Имя представления, которому будут даны или отобраны привилегии.
trigname	Имя триггера, которому будут даны или отобраны привилегии.
procname	Имя процедуры, которой будут даны или отобраны привилегии.
funcname	Имя функции, которой будут даны или отобраны привилегии.
packagename	Имя пакета, которому будут даны или отобраны привилегии.
grantor	Пользователь от имени, которого предоставляются или отбираются привилегии.

Оператор GRANT предоставляет одну или несколько привилегий для объектов базы данных пользователям, ролям, хранимым процедурам, хранимым или внешним функциям, пакетам, триггерам и представлениям.

Оператор REVOKE отменяет привилегии для пользователей, ролей, хранимых процедур, хранимых или внешних функций, пакетов, триггеров и представлений выданные оператором GRANT.

Авторизованный пользователь не имеет никаких привилегий до тех пор, пока какие либо права не будут предоставлены ему явно. При создании объекта только его создатель и SYSDBA имеет привилегии на него и может назначать привилегии другим пользователям, ролям или объектам.

Для использования объектов метаданных, отличных от таблиц, представлений, хранимых процедур и функций, триггеров и пакетов, в пользовательских запросах необходимо предоставить пользователю привилегию USAGE для этих объектов. Поскольку в Firebird хранимые процедуры и функции, триггеры и подпрограммы пакетов выполняются с привилегиями вызывающего пользователя, то при использовании таких объектов метаданных в них, может потребоваться назначить привилегию USAGE и для них.

Необязательное предложение WITH GRANT OPTION оператора GRANT позволяет пользователям, указанным в списке пользователей, передавать другим пользователям привилегии указанные в списке привилегий.

Необязательное предложение GRANT OPTION FOR оператора REVOKE отменяет для соответствующего пользователя или роли право предоставления другим пользователям или ролям указанную привилегию.

Примеры:

1. Выдача привилегий на использование генератора GEN_AGE для пользователей, вошедших с ролью MANAGER.

```
GRANT USAGE ON SEQUENCE GEN_AGE TO ROLE MANAGER;
```

2. Выдача привилегий триггеру на использование генератора GEN_AGE.

```
GRANT USAGE ON SEQUENCE GEN_AGE TO TRIGGER TR_AGE_BI;
```

3. Выдача привилегий пакету PKG_BILL на использование исключения E_ACCESS_DENIED.

```
GRANT USAGE ON EXCEPTION E_ACCESS_DENIED
TO PACKAGE PKG_BILL;
```

Привилегии на изменение метаданных (DDL)

В предыдущих версиях Firebird любой авторизованный пользователь мог создать любой объект базы данных и даже саму базу данных. Кроме того, некоторые объекты базы данных (домены, внешние функции, BLOB фильтры, генераторы, исключения, сортировки) не были защищены от редактирования и удаления, поскольку у этих объектов не хранилась информация о создателе. В Firebird 3 эти недостатки устранены.

Начиная с Firebird 3 создать новый объект метаданных могут:

- SYSDBA;
- владелец базы данных;
- любой пользователь, подключённый с ролью RDB$ADMIN;
- пользователь, получивший привилегии на создание объекта метаданных данного типа.

Изменить/удалить объект метаданных могут:

- SYSDBA;
- владелец базы данных;
- владелец объекта метаданных;
- любой пользователь, подключённый с ролью RDB$ADMIN;
- пользователь, получивший привилегии на изменение/удаление объектов метаданных данного типа.

Создать новую базу данных могут:

- SYSDBA;
- любой пользователь, подключённый с ролью RDB$ADMIN;
- пользователь, получивший привилегии на создание базы данных.

Изменить/удалить базу данных могут:

- SYSDBA;
- Владелец базы данных;
- любой пользователь, подключённый с ролью RDB$ADMIN;
- пользователь, получивший привилегии на создание базы данных.

Доступно: DSQL

Синтаксис:

Назначение привилегий на изменение метаданных.

```
GRANT CREATE <object-type>
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];

GRANT ALTER ANY <object-type>
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];

GRANT DROP ANY <object-type>
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];
```

Отзыв привилегий на изменение метаданных.

```
REVOKE [GRANT OPTION FOR] CREATE <object-type>
FROM {[USER] user-name | [ROLE] role-name};

REVOKE [GRANT OPTION FOR] ALTER ANY <object-type>
FROM {[USER] user-name | [ROLE] role-name};

REVOKE [GRANT OPTION FOR] DROP ANY <object-type>
FROM {[USER] user-name | [ROLE] role-name};
```

Специальная форма для создания/изменения/удаления базы данных.

```
GRANT CREATE DATABASE
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];

GRANT ALTER DATABASE
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];

GRANT DROP DATABASE
TO {[USER] user-name | [ROLE] role-name}
[WITH GRANT OPTION];
```

```
REVOKE [GRANT OPTION FOR] CREATE DATABASE
FROM {[USER] user-name | [ROLE] role-name};

REVOKE [GRANT OPTION FOR] ALTER DATABASE
FROM {[USER] user-name | [ROLE] role-name};

REVOKE [GRANT OPTION FOR] DROP DATABASE
FROM {[USER] user-name | [ROLE] role-name};

<object-type> ::= {
  CHARACTER SET |
  COLLATION |
  DOMAIN |
  EXCEPTION |
  FILTER |
  FUNCTION |
  GENERATOR |
  PACKAGE |
  PROCEDURE |
  ROLE |
  SEQUENCE |
  TABLE |
  VIEW
}
```

Аргумент	Описание
user-name	Имя пользователя.
role-name	Имя роли.

Примечание.

Метаданные триггеров и индексов наследуют привилегии таблиц, которые владеют ими.

Примеры:

1. Выдача привилегий на создание и изменение любой таблицы пользователю Joe.

```
GRANT CREATE TABLE TO Joe;
GRANT ALTER ANY TABLE TO Joe;
```

2. Отозвать привилегию на создание таблиц у пользователя Joe.

```
REVOKE CREATE TABLE FROM Joe;
```

Управление отображением объектов безопасности

В Firebird 3 введены новые SQL полномочия для отображения между пользователями, группами и объектами безопасности и между базами данных.

С введением поддержки множества баз данных безопасности в Firebird появились новые проблемы, которые не могли произойти с единой глобальной базой данных безопасности. Кластеры баз данных, использующие одну и ту же базу данных безопасности, были эффективно разделены. Отображения предоставляют средства для достижения той же эффективности, когда множество баз данных используют каждая свою базу данных безопасности. В некоторых случаях требуется управление для ограничения взаимодействия между такими кластерами. Например:

- когда EXECUTE STATEMENT ON EXTERNAL DATA SOURCE требует обмена данными между кластерами
- когда обще серверный SYSDBA доступ к базам данных необходим от других кластеров, использующих службы
- аналогичные проблемы существовали в Firebird 2.1 и 2.5 под Windows, из-за поддержки доверительной аутентификации: два отдельных списка пользователей – один в базе данных безопасности, а другой в Windows, и необходимо связать их.

Единое решение для всех этих случаев является отображение информации о пользователе, входящего в систему, на внутренние объекты безопасности – CURRENT_USER и CURRENT_ROLE.

Правила отображения

Правила отображения состоит из четырех частей информации:

1. Сфера отображения – будет ли отображение локальным для текущей базы данных или его эффект должен быть глобальным, затрагивая все базы данных в кластере, в том числе и базы данных безопасности.
2. Имя отображения – идентификатор SQL, поскольку отображения являются объектами базы данных.
3. Объект FROM который отображается. Он состоит из четырёх элементов:
   - Источник аутентификации

     имя плагина или
     объект отображения в другой базе данных или
     обще серверная аутентификация или
     любой метод (any method)

   - Имя базы данных, в которой прошла аутентификация
   - Имя объекта, из которого выполняется отображение
   - Тип этого имени - user name | role | OS group - в зависимости от плагина, который добавил это имя во время аутентификации

   Любой из этих элементов допустим, обязательным элементом является только тип.
4. Объект TO на который происходит отображение. Он состоит из двух элементов:

- Имя объекта, на который происходит отображение
- Тип объекта, допускается только USER и ROLE

Синтаксис для отображения объектов

Отображения определяются, используя следующий набор DDL команд:

```
{CREATE | ALTER | CREATE OR ALTER} [GLOBAL] MAPPING name
USING {
  PLUGIN name [IN database] |
  ANY PLUGIN [IN database | SERVERWIDE] |
  MAPPING [IN database] |
  '*' [IN database]}
FROM {ANY type | type name}
TO {USER | ROLE} [name]


DROP [GLOBAL] MAPPING name
```

Описание:

- Любое отображение может иметь атрибут GLOBAL.

  Глобальное отображение работает лучше, если в качестве базы данных безопасности используется база данных Firebird 3 или более высокой версии. Если вы планируете использовать другую базу данных, например, для целей использования собственного поставщика, то вам необходимо создать таблицу в ней и назвать её RDB$MAP с той же структурой, что и RDB$MAP в базе данных Firebird 3 и дать доступ на запись только для SYSDBA.

Осторожно.

Если существуют одноимённые глобальные и локальные отображение, то вам следует знать, что это разные объекты.

- Операторы CREATE, ALTER и CREATE OR ALTER используют один и тот же набор опций. Имя (идентификатор) отображения используется, чтобы идентифицировать его, как и в других наборах команд DDL.
- Предложение USING имеет весьма сложный набор опций:
  - явное указание имени плагина означает, что оно будет работать только с этим плагином;
  - оно может использовать любой доступный плагин, даже если источник является продуктом предыдущего отображения;
  - оно может быть сделано так, чтобы работать только с обще серверными плагинами;
  - оно может быть сделано так, чтобы работать только с результатами предыдущего отображения;

- вы можете опустить использование любого из методов, используя звёздочку (*) в качестве аргумента;
- оно может содержать имя базы данных, из которой происходит отображение объекта FROM.

> **Примечание**.
>
> Этот аргумент не является допустимым для отображения обще серверной аутентификацией.

- Предложение FROM принимает обязательный аргумент – тип именованного объекта.
  - при отображении имен из плагинов, тип определяется плагином;
  - при отображении продукта предыдущего отображения, типом может быть только USER и ROLE;
  - если имя будет указано явно, то оно будет учитываться при отображении;
  - используйте ключевое слово ANY для того чтобы работать с любыми именами данного типа.
- В предложении TO указывается пользователь или роль, на которого будет произведено отображение. NAME является не обязательным аргументом. Если он не указан, то в качестве имени объекта будет использовано оригинальное имя из отображаемого объекта.

Примеры:

В примерах используется синтаксис CREATE. Использование ALTER точно такое же, а использование DROP очевидно.

1. Включить использование доверительной аутентификации Windows во всех базах данных, которые используют текущую базу данных безопасности:

```
CREATE GLOBAL MAPPING TRUSTED_AUTH
USING PLUGIN WIN_SSPI
FROM ANY USER
TO USER;
```

2. Включить SYSDBA подобный доступ для администраторов Windows в текущей базе данных:

```
CREATE MAPPING WIN_ADMINS
USING PLUGIN WIN_SSPI
FROM Predefined_Group
DOMAIN_ANY_RID_ADMINS
TO ROLE RDB$ADMIN;
```

> **Примечание**.

Группа DOMAIN_ANY_RID_ADMINS не существует в Windows, но такое имя будет добавлено плагином win_sspi для обеспечения точной обратной совместимости.

3. Включение доступа определённому пользователю из другой базы данных к текущей базе данных под другим именем:

```
CREATE MAPPING FROM_RT
USING PLUGIN SRP IN "rt"
FROM USER U1 TO USER U2;
```

Важно.

Имена баз данных должны быть заключены в двойные кавычки на операционных системах, которые имею регистр чувствительные имена файлов.

4. Включить SYSDBA сервера (от основной базы данных безопасности) для доступа к текущей базе данных. (Предположим, что база данных использует базу данных безопасности не по умолчанию):

```
CREATE MAPPING DEF_SYSDBA
USING PLUGIN SRP IN "security.db"
FROM USER SYSDBA
TO USER;
```

5. Гарантировать, что у пользователей, которые подключаются унаследованным плагином аутентификации не слишком много прав:

```
CREATE MAPPING LEGACY_2_GUEST
USING PLUGIN legacy_auth
FROM ANY USER
TO USER GUEST;
```

Унаследованные правила отображения

В предыдущих версиях Firebird имелось одно встроенное глобальное правило, действующее по умолчанию: пользователи прошедшие проверку в базе данных безопасности всегда отображается в любую базу данных один к одному. Это безопасное правило: для базы данных безопасности не имеет смысла не доверять себе.

Для обеспечения обратной совместимости это правило сохраняется и в Firebird 3.

Отображение пользователей Windows на CURRENT_USER

В Firebird версии 2.1 и 2.5 при включенной доверительной аутентификации, пользователи прошедшие проверку по умолчанию автоматически отображались в CURRENT_USER. В Firebird 3 отображение должно быть сделано явно для систем с несколькими базами данных безопасности и включенной доверительной аутентификацией.

Шифрование базы данных

В Firebird 3 появилась возможность зашифровать данные хранимые в базе данных. Не весь файл базы данных шифруется: только страницы данных, индексов и blob.

Для того чтобы сделать шифрование базы данных возможным необходимо получить или написать плагин шифрования базы данных.

Примечание.

Пример плагина шифрования в examples/dbcrypt не производит реального шифрования, это просто пример того, как можно написать этот плагин.

Секретный ключ

Основная проблема с шифрованием базы данных состоит в том, как хранить секретный ключ. Firebird предоставляет помощника для передачи этого ключа от клиента, но это вовсе не означает, что хранение ключей на клиенте является лучшим способом: это не более, чем одна возможных альтернатив. Хранение ключей на том же диске что и база данных является очень плохим вариантом.

Задачи

Для эффективного разделения шифрования и доступа к ключу, плагин шифрования базы данных разделён на две части: само шифрование и держатель секретного ключа. Это может быть эффективным подходом, когда вы хотите использовать некоторый хороший алгоритм шифрования, но у вас есть собственный секретный способ хранение ключей.

После того как вы определитесь с плагином и ключом, вы можете включить процесс шифрования:

```
ALTER DATABASE ENCRYPT WITH plugin_name
```

Аргумент	Описание

plugin_name	Имя плагина шифрования.

Шифрование начинается сразу после этого оператора и будет выполняться в фоновом режиме. Нормальная работа с базами данных не нарушается во время шифрования.

Подсказка.

Процесс шифрования может быть проконтролирован с помощью поля MON$CRYPT_PAGE в псевдо-таблице MON$DATABASE или смотреть страницу заголовка базы данных с помощью gstat -e.

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

Для дешифрования базы данных выполните:

```
ALTER DATABASE DECRYPT
```

Для Linux пример плагина с именем libDbCrypt_example.so можно найти в поддиректории /plugins/.