Что нового в Firebird 5.0. SQL

Симонов Денис

Version 1.0 от 03.12.2023

Этот материал был создан при поддержке и спонсорстве компании iBase.ru, которая разрабатывает инструменты Firebird SQL для предприятий и предоставляет сервис технической поддержки для Firebird SQL.

Предисловие

Недавно вышел Release Candidate СУБД Firebird 5.0, а это обозначает, что пришло время ознакомиться с новыми возможностями предстоящего релиза. Это восьмой основной выпуск СУБД Firebird, разработка которого началась в мае 2021 года.

В Firebird 5.0 команда разработчиков сосредоточила свои усилия на повышение производительности СУБД в различных аспектах, таких как:

- параллельное выполнение для распространённых задач: backup, restore, sweep, создание и перестроение индекса;
- улучшение масштабирования в многопользовательской среде;
- ускорение повторной подготовки запросов (кеш компилированных запросов);
- улучшение оптимизатора;
- улучшение алгоритма сжатия записей;
- поиск узких мест с помощью плагина профилирования.

Поскольку объём материала довольно большой, то я разделю описание новых функций на несколько частей:

- улучшение в оптимизаторе запросов;
- новые возможности в языке SQL Firebird 5.0;
- другие функции появившиеся в Firebird 5.0;
- поиск узких места с помощью плагина PSQL профилирования.

В прошлый раз я рассказал о улучшениях в оптимизаторе запросов. Теперь посмотрим что нового появилось в языке SQL Firebird 5.0.

Новые возможности в языке SQL

Поддержка предложения WHEN NOT MATCHED BY SOURCE в операторе MERGE

Оператор MERGE производит слияние записей источника и целевой таблицы (или обновляемым представлением). В процессе выполнения оператора MERGE читаются записи источника и выполняются INSERT, UPDATE или DELETE для целевой таблицы в зависимости от условий.

Синтаксис оператора MERGE выглядит следующим образом:

```
MERGE
  INTO target [[AS] target_alias]
  USING <source> [[AS] source_alias]
  ON <join condition>
  <merge when> [<merge when> ...]
  [<plan clause>]
  [<order by clause>]
  [<returning clause>]

<source> ::= tablename | (<select_stmt>)

<merge when> ::=
    <merge when matched>
  | <merge when not matched by target>
  | <merge when not matched by source>

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

<merge when not matched by target> ::=
  WHEN NOT MATCHED [ BY TARGET ] [ AND <condition> ]
  THEN INSERT [ <left paren> <column_list> <right paren> ]
  VALUES <left paren> <value_list> <right paren>

<merge when not matched by source> ::=
  WHEN NOT MATCHED BY SOURCE [ AND <condition> ] THEN
  { UPDATE SET <assignment list> | DELETE }
```

В Firebird 5.0 появились условные ветки <merge when not matched by source>, которые позволяют обновить или удалить записи из целевой таблицы, если они отсутствуют в источнике данных.

Теперь оператор MERGE является по настоящему универсальным комбайном для любых модификаций целевой таблицы по некоторому набору данных.

Источником данных может быть таблица, представление, хранимая процедура или производная таблица. При выполнении оператора MERGE производится соединение между источником (USING) и целевой таблицей. Тип соединения зависит от присутствия предложений WHEN NOT MATCHED:

- <merge when not matched by target> и <merge when not matched by source> — FULL JOIN
- <merge when not matched by source> — RIGHT JOIN
- <merge when not matched by target> — LEFT JOIN
- только <merge when matched> — INNER JOIN

Действие над целевой таблицей, а также условие при котором оно выполняется, описывается в предложении WHEN. Допускается несколько предложений WHEN MATCHED, WHEN NOT MATCHED [BY TARGET] и WHEN NOT MATCHED BY SOURCE.

Если условие в предложении WHEN не выполняется, то Firebird пропускает 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 запись результата соединения между источником (USING) и целевой таблицей. Для каждой записи результата соединения выполняется только одно действие.

WHEN MATCHED

Указывает, что все строки *target*, которые соответствуют строкам, возвращенным выражением <source> ON <join condition>, и удовлетворяют дополнительным условиям поиска, обновляются (предложение UPDATE) или удаляются (предложение DELETE) в соответствии с предложением <merge when matched>.

Допускается указывать несколько предложений WHEN MATCHED. Если указано более одного предложения WHEN MATCHED, то все их следует дополнять дополнительными условиями поиска, за исключением последнего.

Инструкция MERGE не может обновить одну строку более одного раза или одновременно обновить и удалить одну и ту же строку.

WHEN NOT MATCHED [BY TARGET]

Указывает, что все строки *target*, которые не соответствуют строкам, возвращенным выражением <source> ON <join condition>, и удовлетворяют дополнительным условиям поиска, вставляются в целевую таблицу (предложение INSERT) в соответствии с предложением <merge when not matched by target>.

Допускается указывать несколько предложений WHEN NOT MATCHED [BY TARGET]. Если указано более одного предложения WHEN NOT MATCHED [BY TARGET], то все их следует дополнять дополнительными условиями поиска, за исключением последнего.

WHEN NOT MATCHED BY SOURCE

Указывает, что все строки *target*, которые не соответствуют строкам, возвращенным

выражением <source> ON <join condition>, и удовлетворяют дополнительным условиям поиска, (предложение UPDATE) или удаляются (предложение DELETE) в соответствии с предложением <merge when not matched by source>.

Предложение WHEN NOT MATCHED BY SOURCE доступно начиная с Firebird 5.0.

Допускается указывать несколько предложений WHEN NOT MATCHED BY SOURCE. Если указано более одного предложения WHEN NOT MATCHED BY SOURCE, то все их следует дополнять дополнительными условиями поиска, за исключением последнего.

NOTE Обратите внимание! В списке SET предложения UPDATE не имеет смысла использовать выражения со ссылкой на <source>, поскольку ни одна запись из <source> не соответствует записям *target*.

Пример использование MERGE с предложением WHEN NOT MATCHED BY SOURCE

Допустим у вас есть некоторый прайс во временной таблице tmp_price и необходимо обновить текущий прайс так чтобы:

- если товара в текущем прайсе нет, то добавить его;
- если товар в текущем прайсе есть, то обновить для него цену;
- если товар присутствует в текущем прайсе. но его нет в новом, то удалить эту строку прайса.

Все эти действия можно сделать одним запросом:

```
MERGE INTO price
USING tmp_price
ON price.good_id = tmp_price.good_id
WHEN NOT MATCHED
  -- добавляем если не было
  THEN INSERT(good_id, name, cost)
  VALUES(tmp_price.good_id, tmp_price.name, tmp_price.cost)
WHEN MATCHED AND price.cost <> tmp_price.cost THEN
  -- обновляем цену, если товар есть в новом прайсе и цена отличается
  UPDATE SET cost = tmp_price.cost
WHEN NOT MATCHED BY SOURCE
  -- если в новом прайсе товара нет, то удаляем его из текущего прайса
  DELETE;
```

NOTE В этом примере вместо временной таблицы tmp_price может быть сколь угодно сложный SELECT запрос или хранимая процедура. Но учтите, что поскольку присутствуют оба предложения WHEN NOT MATCHED [BY TARGET] и WHEN NOT MATCHED BY SOURCE, то соединение целевой таблицы и источника данных будет происходить с помощью FULL JOIN. В текущей версии Firebird FULL JOIN при невозможности использовать индексы как справа, так и слева будет выполняться очень медленно.

Предложение SKIP LOCKED

В Firebird 5.0 появилось предложение SKIP LOCKED, которое может использоваться в операторах SELECT .. WITH LOCK, UPDATE и DELETE. Использование этого предложения заставляет движок пропускать записи, заблокированные другими транзакциями, вместо того, чтобы ждать их, или вызывать ошибки конфликта обновления.

Использование SKIP LOCKED полезно для реализации рабочих очередей, в которых один или несколько процессов отправляют работу в таблицу и выдают событие, в то время как рабочие потоки прослушивают события и читают/удаляют элементы из таблицы. Используя SKIP LOCKED, несколько работников могут получать эксклюзивные задания из таблицы без конфликтов.

```
SELECT
  [FIRST ...]
  [SKIP ...]
  FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [{ ROWS ... } | {OFFSET ...} | {FETCH ...}]
  [FOR UPDATE [OF ...]]
  [WITH LOCK [SKIP LOCKED]]
```

```
UPDATE <sometable>
  SET ...
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

```
DELETE FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

NOTE В случае использования предложения SKIP LOCKED сначала пропускаются заблокированные записи, а затем применяются ограничители FIRST/SKIP/ROWS/OFFSET/FETCH к оставшимся записям.

Пример использования:

- Подготовка метаданных

```
create table emails_queue (
  subject varchar(60) not null,
  text blob sub_type text not null
);

set term !;

create trigger emails_queue_ins after insert on emails_queue
as
begin
  post_event('EMAILS_QUEUE');
end!

set term ;!
```

- Отправка приложением или подпрограммой

```
insert into emails_queue (subject, text)
values ('E-mail subject', 'E-mail text...');

commit;
```

- Клиентское приложение

```
-- Клиентское приложение может прослушивать событие EMAILS_QUEUE,
-- чтобы отправлять электронные письма, используя этот запрос:

delete from emails_queue
  rows 10
  skip locked
  returning subject, text;
```

Может быть запущено более одного экземпляра приложения, например, для балансировки нагрузки.

NOTE	Подробнее о практическом использованию `SKIP LOCKED` для организации мы поговорим в следующий раз.

Поддержка возврата множества записей операторами с `RETURNING`

Начиная с Firebird 5.0 клиентские модифицирующие операторы `INSERT .. SELECT`, `UPDATE`, `DELETE`, `UPDATE OR INSERT` и `MERGE`, содержащие предложение `RETURNING` возвращают курсор, то

есть они способны вернуть множество записей вместо выдачи ошибки "multiple rows in singleton select", как это происходило ранее.

Теперь эти запросы во время подготовки описываются как `isc_info_sql_stmt_select`, тогда как в предыдущих версии они были описаны как `isc_info_sql_stmt_exec_procedure`.

Сингелтон-операторы `INSERT .. VALUES`, а также позиционированные операторы `UPDATE` и `DELETE` (то есть, которые содержат предложение `WHERE CURRENT OF`) сохраняют существующее поведение и описываются как `isc_info_sql_stmt_exec_procedure`.

Однако все эти запросы, если они используются в PSQL и применяется предложение RETURNING, по-прежнему рассматриваются как сингелтоны.

Примеры модифицирующие операторов содержащих RETURNING и возвращающих курсор:

```
INSERT INTO dest(name, val)
SELECT desc, num + 1 FROM src WHERE id_parent = 5
RETURNING id, name, val;

UPDATE dest
SET a = a + 1
RETURNING id, a;

DELETE FROM dest
WHERE price < 0.52
RETURNING id;

MERGE INTO PRODUCT_INVENTORY 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 SRC(ID_PRODUCT, QUANTITY)
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
RETURNING OLD.QUANTITY, NEW.QUANTITY, SRC.QUANTITY;
```

Частичные индексы

В Firebird 5.0 при создании индекса появилась возможность указать необязательное предложение WHERE, которое определяет условие поиска, ограничивающее подмножество записей таблицы для индексирования. Такие индексы называются частичными индексами. Условие поиска должно содержать один или несколько столбцов таблицы.

Определение частичного индекса может включать спецификацию UNIQUE. В этом случае каждый ключ в индексе должен быть уникальным. Это позволяет обеспечить уникальность для некоторого подмножества строк таблицы.

Определение частичного индекса также может включать предложение COMPUTED BY, таким образом частичный индекс может быть вычисляемым.

Таким образом, полный синтаксис создания индекса выглядит следующим образом:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX indexname ON tablename
{(<column_list>) | COMPUTED [BY] (<value_expression>)}
[WHERE <search_condition>]

<column_list> ::= col [, col ...]
```

Оптимизатор может использовать частичный индекс можно использовать только в следующих случаях:

- условие WHERE включает точно такое же логическое выражение, как и определенное для индекса;
- условие поиска, определенное для индекса, содержит логические выражения, объединенные OR, и одно из них явно включено в условие WHERE;
- условие поиска, определенное для индекса, указывает IS NOT NULL, а условие WHERE включает выражение для того же поля, которое, как известно, игнорирует NULL.

Если для одного и того же набора полей существует обычный индекс и частичный индекс, то оптимизатор выберет обычный индекс, даже если условие WHERE включает тоже самое выражение, что определено в частичном индексе. Причина такого поведения состоит в том, что у обычного индекса селективность лучше, чем у частичного. Но существуют исключения из этого правила: использование для индексируемых полей предикатов с плохой избирательностью, таких как <>, IS DISTINCT FROM или IS NOT NULL, при условии что данный предикат используется в частичном индексе.

NOTE	Частичные индексы не могут быть использованы для ограничения первичного и внешнего ключа, то есть в выражении USING INDEX нельзя указать определение частичного индекса.

Давайте посмотрим в какие случаях частичные индексы полезны.

Example 1. Обеспечение частичной уникальности

Предположим у нас есть таблица хранящая email адреса человека.

```
CREATE TABLE MAN_EMAILS (
  CODE_MAN_EMAIL BIGINT GENERATED BY DEFAULT AS IDENTITY,
  CODE_MAN BIGINT NOT NULL,
  EMAIL VARCHAR(50) NOT NULL,
  DEFAULT_FLAG BOOLEAN DEFAULT FALSE NOT NULL,
  CONSTRAINT PK_MAN_EMAILS PRIMARY KEY(CODE_MAN_EMAIL),
  CONSTRAINT FK_EMAILS_REF_MAN FOREIGN KEY(CODE_MAN) REFERENCES MAN(CODE_MAN)
);
```

У одного человека может быть много email адресов, но только один может быть адресом по умолчанию. Обычный уникальный индекс или ограничение в данном случае не подойдёт, поскольку в этом случае мы будем ограничены всего двумя адресами.

Здесь нам на помощь придёт частичный уникальный индекс:

```
CREATE UNIQUE INDEX IDX_UNIQUE_DEFAULT_MAN_EMAIL
ON MAN_EMAILS(CODE_MAN) WHERE DEFAULT_FLAG IS TRUE;
```

Таким образом для одного человека мы позволяем сколько угодно адресов с `DEFAULT_FLAG=FALSE` и только один адрес с `DEFAULT_FLAG=TRUE`.

Частичные индексы можно использовать просто для того чтобы индекс был более компактным.

Example 2. Уменьшение размера индекса

Предположим у вас в базе данных есть таблица лошадей HORSE и в ней есть поле IS_ANCESTOR, которое используется для отметки является ли лошадь родоначальником линии или семейства. Очевидно, что родоначальников в сотни раз меньше, чем других лошадей.

К примеру если мы выполним следующий запрос запрос, то получим:

```
SELECT
  COUNT(*) FILTER(WHERE IS_ANCESTOR IS TRUE) AS CNT_ANCESTOR,
  COUNT(*) FILTER(WHERE IS_ANCESTOR IS FALSE) AS CNT_OTHER
FROM HORSE
```

```
         CNT_ANCESTOR             CNT_OTHER
==================== ====================
                1426                518197
```

Задача состоит в том, чтобы быстро получать список родоначальников. Из приведённой статистики также очевидно, что для варианта IS_ANCESTOR IS FALSE использование индексов практически бесполезно.

В принципе мы можем создать обыкновенный индекс

```
CREATE INDEX IDX_HORSE_ANCESTOR ON HORSE(IS_ANCESTOR);
```

Но в данном случае такой индекс будет избыточным. Давайте посмотрим его статистику:

```
    Index IDX_HORSE_ANCESTOR (26)
        Root page: 163419, depth: 2, leaf buckets: 159, nodes: 519623
        Average node length: 4.94, total dup: 519621, max dup: 518196
        Average key length: 2.00, compression ratio: 0.50
        Average prefix length: 1.00, average data length: 0.00
        Clustering factor: 9809, ratio: 0.02
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 1
            40 - 59% = 0
            60 - 79% = 0
            80 - 99% = 158
```

Вместо обычного индекса мы можем создать частичный индекс (предыдущий надо удалить):

```
CREATE INDEX IDX_HORSE_ANCESTOR ON HORSE(IS_ANCESTOR) WHERE IS_ANCESTOR IS TRUE;
```

Сравним статистику:

```
    Index IDX_HORSE_ANCESTOR (26)
        Root page: 163417, depth: 1, leaf buckets: 1, nodes: 1426
        Average node length: 4.75, total dup: 1425, max dup: 1425
        Average key length: 2.00, compression ratio: 0.50
        Average prefix length: 1.00, average data length: 0.00
        Clustering factor: 764, ratio: 0.54
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 0
            40 - 59% = 1
            60 - 79% = 0
            80 - 99% = 0
```

Как видите частичный индекс намного более компактный.

Проверим что он может быть использован для получения родоначальников:

```
SELECT COUNT(*)
FROM HORSE
WHERE IS_ANCESTOR IS TRUE;
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_ANCESTOR" Full Scan

               COUNT
=====================
                1426

Current memory = 556868928
Delta memory = 176
Max memory = 575376064
Elapsed time = 0.007 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 2192
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |    1426|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Обратите внимание, если в запросе вы укажете `WHERE IS_ANCESTOR` или `WHERE IS_ANCESTOR =`

`TRUE`, то индекс не будет использован. Необходимо, чтобы выражение указанное для фильтрации индекса полностью совпадало с выражением в `WHERE` вашего запроса.

Другим случаем, когда частичные индексы могут быть полезны это использование их с неселективными предикатами.

Example 3. Использование частичных индексов с неселективными предикатами

Предположим нам необходимо получить всех мёртвых лошадей у которых известна дата смерти. Лошадь точно является мёртвой, если у неё выставлена дата смерти, но часто бывает так, что её не выставляют или просто она неизвестна. Причём количество неизвестных дат смерти намного больше, чем известных. Для этого напишем следующий запрос:

```
SELECT COUNT(*)
FROM HORSE
WHERE DEATHDATE IS NOT NULL;
```

Нам хочется, получать этот список максимально быстро, поэтому попытаемся создать индекс на поле DEATHDATE.

```
CREATE INDEX IDX_HORSE_DEATHDATE
ON HORSE(DEATHDATE);
```

Теперь попытаемся выполнить запрос выше и посмотреть на его план и статистику:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Full Scan

                COUNT
=====================
                16234

Current memory = 2579550800
Delta memory = 176
Max memory = 2596993840
Elapsed time = 0.196 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 555810
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |  519623|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Как видите индекс задействовать не получилось. Причина в том, что предикаты IS NOT NULL, <>, IS DISTINCT FROM являются малоселективными. В настоящее время в Firebird нет гистограмм с распределением значений ключей индекса, а потому распределение считается равномерным. При равномерном распределении для таких предикатов нет смысла использовать индекс, что и делается.

А теперь попробуем удалить ранее созданный индекс и создать вместо него частичный индекс:

```
DROP INDEX IDX_HORSE_DEATHDATE;

CREATE INDEX IDX_HORSE_DEATHDATE
ON HORSE(DEATHDATE) WHERE DEATHDATE IS NOT NULL;
```

И попробуем повторить запрос выше:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_DEATHDATE" Full Scan

                COUNT
====================
                16234

Current memory = 2579766848
Delta memory = 176
Max memory = 2596993840
Elapsed time = 0.017 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 21525
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |    16234|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Как видите оптимизатору удалось задействовать наш индекс. Но самое интересное. наш индекс будет продолжать работать и с другими предикатами сравнения с датой (для `IS NULL` не будет).

```
SELECT COUNT(*)
FROM HORSE
WHERE DEATHDATE = DATE'01.01.2005';
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_DEATHDATE" Range Scan (full match)

                COUNT
====================
                  190

Current memory = 2579872992
Delta memory = 192
Max memory = 2596993840
Elapsed time = 0.004 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 376
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |     190|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Всё потому, что оптимизатор в этом случае догадался, что условие фильтрации `IS NOT NULL` в частичном индексе покрывает любые другие предикаты не сравнивающие с `NULL`.

Важно отметить, что если вы в частичном индексе укажете условие `FIELD > 2`, а в запросе будет условие поиска `FIELD > 1`, то несмотря на то, что любое число больше 2, также больше 1, частичный индекс задействован не будет. Оптимизатор не настолько умён, чтобы вывести данное условие эквивалентности.

Функции `UNICODE_CHAR` и `UNICODE_VAL`

В Firebird 2.1 появилась пара функций `ASCII_CHAR` - возвращающая символ по его коду в таблице ASCII, и `ASCII_VAL` - возвращающая код в таблице ASCII по символу. Эти функции применимы только к однобайтным кодировкам, для UTF-8 ничего подобного не было.

В Firebird 5.0 добавили ещё две функции, которые работают с многобайтными кодировками:

```
UNICODE_CHAR (number)

UNICODE_VAL (string)
```

Функция `UNICODE_CHAR` возвращает UNICODE символ для заданной кодовой точки.

Функция `UNICODE_VAL` возвращает UTF-32 кодовую точку для первого символа в строке. Для пустой строки возвращается 0.

```
SELECT
  UNICODE_VAL(UNICODE_CHAR(0x1F601)) AS CP_VAL,
  UNICODE_CHAR(0x1F601) AS CH
FROM RDB$DATABASE
```

Выражения запросов в скобках

Синтаксис DML был расширен, чтобы разрешить использование выражения запроса в круглых скобках (select, включая предложения `order by`, `offset` и `fetch`, но без предложения `with`), где ранее была разрешена только спецификация запроса (select без предложений `with`, `order by`, `offset` и `fetch`).

Это обеспечивает более выразительные запросы, особенно в операторах `UNION`, и обеспечивает большую совместимость с операторами, генерируемыми определенными ORM.

NOTE	Использование выражений запроса в скобках обходится дорого, поскольку они требуют дополнительного контекста запроса по сравнению с простой спецификацией запроса. Максимальное количество контекстов запроса в операторе — 255.

Пример:

```
(
  select emp_no, salary, 'lowest' as type
  from employee
  order by salary asc
  fetch first row only
)
union all
(
  select emp_no, salary, 'highest' as type
  from employee
  order by salary desc
  fetch first row only
);
```

Улучшение литералов

Полный синтаксис строковых литералов

Синтаксис литералов символьных строк был изменен для поддержки полного стандартного синтаксиса SQL. Это означает, что литерал может быть «прерван» пробелами или комментариями. Это можно использовать, например, для разбиения длинного литерала на несколько строк или для предоставления встроенных комментариев.

Синтаксис строкового литерала согласно ISO/IEC 9075-2:2016 SQL - Part 2: Foundation

```
<character string literal> ::=
  [ <introducer> <character set specification> ]
    <quote> [ <character representation>... ] <quote>
    [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<separator> ::=
  { <comment> | <white space> }...
```

Пример:

```
-- пробелы между литералами
select 'ab'
       'cd'
from RDB$DATABASE;
-- output: 'abcd'

-- комментарий и пробелы между литералами
select 'ab' /* comment */ 'cd'
from RDB$DATABASE;
-- output: 'abcd'
```

Полный синтаксис двоичных литералов

Синтаксис двоичных строковых литералов был изменен для поддержки полного стандартного синтаксиса SQL. Это означает, что литерал может содержать пробелы для разделения шестнадцатеричных символов и может быть «прерван» пробелами или комментариями. Это можно использовать, например, для того, чтобы сделать шестнадцатеричную строку более читабельной за счет группировки символов, или для разбиения длинного литерала на несколько строк, или для предоставления встроенных комментариев.

Синтаксис двоичного литерала согласно ISO/IEC 9075-2:2016 SQL - Part 2: Foundation

```
<binary string literal> ::=
  {X|x} <quote> [ <space>... ] [ { <hexit> [ <space>... ] <hexit> [ <space>... ] }...
      ] <quote>
    [ { <separator> <quote> [ <space>... ] [ { <hexit> [ <space>... ]
    <hexit> [ <space>... ] }... ] <quote> }... ]
```

Примеры:

```
-- Группировка по байтам (пробелы внутри литерала)
select _win1252 x'42 49 4e 41 52 59'
from RDB$DATABASE;
-- output: BINARY

-- пробелы между литералами
select _win1252 x'42494e'
                 '415259'
from RDB$DATABASE;
-- output: BINARY
```

Улучшение предиката IN

До Firebird 5.0 предикат IN со списком констант был ограничен 1500 элементами, поскольку обрабатывался рекурсивно преобразуя исходное выражение в эквивалентную форму.

То есть,

```
F IN (V1, V2, ... VN)
```

преобразуется в

```
F = V1 OR F = V2 OR .... F = VN
```

Начиная с Firebird 5.0 обработка предикатов `IN <list>` является линейной. Лимит в 1500 элементов увеличен до 65535 элементов. Кроме того, запросы использующие предикат `IN` со списком констант обрабатываются значительно быстрее. Подробно об этом было рассказано в главе про улучшение оптимизатора.

Пакет `RDB$BLOB_UTIL`

Традиционно работа с BLOB внутри PSQL кода обходилась дорого, поскольку при любой модификации BLOB всегда создаётся новый временный BLOB, это приводит к дополнительному потреблению памяти, а в ряде случае и к разрастанию файла базы данных для хранения временных BLOB.

В Firebird 4.0.2 для решения проблем конкатенации BLOB была добавлена встроенная функция `BLOB_APPEND`. В Firebird 5.0 был добавлен встроенный пакет `RDB$BLOB_UTIL` с процедурами и функциями для более эффективной манипуляции над BLOB.

Здесь я не буду описывать этот пакет целиком, поскольку это вы можете найти в Release Notes и в "Руководстве по языку SQL Firebird 5.0", а лишь покажу примеры для практического использования.

Использование функции RDB$BLOB_UTIL.NEW_BLOB

Функция RDB$BLOB_UTIL.NEW_BLOB создает новый BLOB SUB_TYPE BINARY. Она возвращает BLOB, подходящий для добавления данных, аналогично BLOB_APPEND.

Преимущество перед BLOB_APPEND заключается в том, что можно установить собственные параметры SEGMENTED и TEMP_STORAGE.

Функция BLOB_APPEND всегда создает BLOB-объекты во временном хранилище, что не всегда может быть лучшим подходом, если созданный BLOB-объект будет храниться в постоянной таблице, поскольку для этого потребуется операция копирования.

BLOB, возвращаемый этой функцией, даже если TEMP_STORAGE = FALSE, может использоваться с BLOB_APPEND для добавления данных.

Table 1. Входные параметры функции RDB$BLOB_UTIL.NEW_BLOB

Параметр	Тип	Описание
SEGMENTED	BOOLEAN NOT NULL	Тип BLOB. Если TRUE - будет создан сегментированный BLOB, FALSE - потоковый.
TEMP_STORAGE	BOOLEAN NOT NULL	В каком хранилище создаётся BLOB. TRUE - во временном, FALSE - в постоянном (для записи в обычную таблицу).

Тип возвращаемого результата
BLOB SUB_TYPE BINARY

Пример:

```
execute block
declare b blob sub_type text;
as
begin
  -- создаём потоковый не временный BLOB, поскольку далее он будет добавлен в таблицу
  b = rdb$blob_util.new_blob(false, false);

  b = blob_append(b, 'abcde');
  b = blob_append(b, 'fghikj');

  update t
  set some_field = :b
  where id = 1;
end
```

Чтение BLOB порциями

Когда надо было прочитать часть BLOB вы пользовались функцией SUBSTRING, но у этой

функции есть один существенный недостаток, она всегда возвращает новый временный BLOB.

Начиная с Firebird 5.0 вы можете использовать для этой цели функцию `RDB$BLOB_UTIL.READ_DATA`.

Table 2. Входные параметры функции `RDB$BLOB_UTIL.READ_DATA`

Параметр	Тип	Описание
`HANDLE`	`INTEGER NOT NULL`	Дескриптор открытого BLOB.
`LENGTH`	`INTEGER`	Количество байт, которое необходимо прочитать.

Тип возвращаемого результата
`VARBINARY(32765)`

Функция `RDB$BLOB_UTIL.READ_DATA` используется для чтения фрагментов данных из дескриптора BLOB, открытого с помощью `RDB$BLOB_UTIL.OPEN_BLOB`. Когда BLOB полностью прочитан и данных больше нет, она возвращает `NULL`.

Если в `LENGTH` передается положительное число, то возвращается `VARBINARY` максимальной длины `LENGTH`.

Если в `LENGTH` передается `NULL`, то возвращается только сегмент BLOB с максимальной длиной 32765.

Когда работа с дескриптором BLOB окончена, его необходимо закрыть с помощью процедуры `RDB$BLOB_UTIL.CLOSE_HANDLE`.

Example 4. Открытие BLOB и его возврат по частям в EXECUTE BLOCK

```
execute block returns (s varchar(10))
as
    declare b blob = '1234567';
    declare bhandle integer;
begin
    -- открывает BLOB для чтения и возвращает его хендл.
    bhandle = rdb$blob_util.open_blob(b);

    -- Получаем blob частями
    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    -- Когда данных больше нет возращается NULL.
    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    -- Закрываем BLOB хендл.
    execute procedure rdb$blob_util.close_handle(bhandle);
end
```

Передав в качестве параметра LENGTH значение NULL можно сделать посегментное чтение BLOB, если сегменты не превышают 32765 байт.

Напишем процедуру для посегментного возврата BLOB

```
CREATE OR ALTER PROCEDURE SP_GET_BLOB_SEGEMENTS (
  TXT BLOB SUB_TYPE TEXT CHARACTER SET NONE
)
RETURNS (
  SEG VARCHAR(32765) CHARACTER SET NONE
)
AS
  DECLARE H INTEGER;
BEGIN
  H = RDB$BLOB_UTIL.OPEN_BLOB(TXT);
  SEG = RDB$BLOB_UTIL.READ_DATA(H, NULL);
  WHILE (SEG IS NOT NULL) DO
  BEGIN
    SUSPEND;
    SEG = RDB$BLOB_UTIL.READ_DATA(H, NULL);
  END
  EXECUTE PROCEDURE RDB$BLOB_UTIL.CLOSE_HANDLE(H);
END
```

Её можно применить, например вот так:

```
WITH
  T AS (
    SELECT LIST(CODE_HORSE) AS B
    FROM HORSE
  )
SELECT
  S.SEG
FROM T
  LEFT JOIN SP_GET_BLOB_SEGEMENTS(T.B) S ON TRUE
```