Что нового в Firebird 5.0. Оптимизатор

Симонов Денис

Version 1.4 от 14.11.2023

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 этой статье мы познакомимся с улучшениями в оптимизаторе запросов.

Chapter 1. Улучшение оптимизатора

В Firebird 5.0 оптимизатор запросов подвергся самым значительным изменениям со времён Firebird 2.0. Далее я опишу, что именно изменилось и приведу примеры с замерами производительности.

1.1. Стоимостная оценка HASH vs NESTED LOOP JOIN

Соединение потоков с помощью алгоритма HASH JOIN появилось в Firebird 3.0.

При соединение методом HASH JOIN входные потоки всегда делятся на ведущий и ведомый, при этом ведомым обычно выбирается поток с наименьшей кардинальностью. Сначала меньший (ведомый) поток целиком вычитывается во внутренний буфер. В процессе чтения к каждому ключу связи применяется хеш-функция и пара {хеш, указатель в буфере} записывается в хеш-таблицу. После чего читается ведущий поток и его ключ связи опробуется в хеш-таблице. Если соответствие найдено, то записи обоих потоков соединяются и выдаются на выход. В случае нескольких дубликатов данного ключа в ведомой таблице на выход будут выданы несколько записей. Если вхождения ключа в хеш-таблицу нет, переходим к следующей записи ведущего потока и так далее.

Этот алгоритм соединения работает только при сравнении по строгому равенству ключей, и допускает выражения над ключами.

До Firebird 5.0 метод соединения HASH JOIN применялся только при отсутствии индексов по условию связи или их неприменимости, в противном случае оптимизатор выбирал алгоритм NESTED LOOP JOIN с использованием индексов. На самом деле это не всегда оптимально. Если большой поток соединяется с маленькой таблицей по первичному ключу, то каждая запись такой таблицы будет читаться многократно, кроме того многократно будут прочтены и страницы индексов, если они используются. При использовании соединения HASH JOIN меньшая таблица будет прочитана ровно один раз. Естественно стоимость хеширования и пробирования не бесплатны, поэтому выбор какой алгоритм применять происходит на основе стоимости.

Далее посмотрим как один и тот же запрос будет выполнен в Firebird 4.0 и Firebird 5.0. Чтобы было понятно что происходит я приведу explain план, обычную статистику и по-табличную статистику.

Допустим у вас есть одна большая таблица и к ней по первичному ключу присоединяется много небольших справочных таблиц.

```
SELECT
  COUNT(*)
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
```

В данном случае я использую `COUNT(*)` чтобы исключить время фетча записей на клиента, а так же гарантировать что будут извлечены именно все записи.

Результат в Firebird 4.0.

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Table "COLOR" Full Scan
            -> Filter
                -> Table "HORSE" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_COLOR" Range Scan (full match)
            -> Filter
                -> Table "SEX" Access By ID
                    -> Bitmap
                        -> Index "PK_SEX" Unique Scan
            -> Filter
                -> Table "BREED" Access By ID
                    -> Bitmap
                        -> Index "PK_BREED" Unique Scan
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               519623

Current memory = 2614108752
Delta memory = 438016
Max memory = 2614392048
Elapsed time = 2.642 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 5857109
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
BREED                           |         |   519623|         |         |         |
COLOR                           |      239|         |         |         |         |
FARM                            |         |   519623|         |         |         |
HORSE                           |         |   519623|         |         |         |
SEX                             |         |   519623|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Теперь выполним тот же самый запрос в Firebird 5.0.

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Hash Join (inner)
                        -> Nested Loop Join (inner)
                            -> Table "COLOR" Full Scan
                            -> Filter
                                -> Table "HORSE" Access By ID
                                    -> Bitmap
                                        -> Index "FK_HORSE_COLOR" Range Scan (full match)
                        -> Record Buffer (record length: 25)
                            -> Table "SEX" Full Scan
                    -> Record Buffer (record length: 25)
                        -> Table "BREED" Full Scan
                -> Record Buffer (record length: 33)
                    -> Table "FARM" Full Scan

                COUNT
=====================
               519623

Current memory = 2579749376
Delta memory = 352
Max memory = 2582802608
Elapsed time = 0.702 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 645256
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
BREED                           |      282|         |         |         |         |
COLOR                           |      239|         |         |         |         |
FARM                            |    36805|         |         |         |         |
HORSE                           |         |   519623|         |         |         |
SEX                             |        4|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Как видите разница времени выполнения в 3,5 раза!

1.2. Стоимостная оценка HASH vs MERGE JOIN

Алгоритм соединения слиянием MERGE JOIN был временно отключен в Firebird 3.0 в пользу соединения алгоритмом HASH JOIN. Обычно он применялся в тех случаях когда использование алгоритма NESTED LOOP JOIN было неоптимальным, то есть в первую очередь при отсутствии индексов по условию связи или их неприменимости, а также при

отсутствии зависимости между входными потоками.

В большинстве случаев соединение методом HASH JOIN более эффективно, поскольку не требуется выполнять предварительную сортировку потоков по ключам соединения, но есть случаи когда MERGE JOIN более эффективен:

- соединяемые потоки уже отсортированы по ключам соединения, например производится соединение результатов двух подзапросов по ключам указанным в GROUP BY:

```
select count(*)
from
(
    select code_father+0 as code_father, count(*) as cnt
    from horse group by 1
) h
join (
    select code_father+0 as code_father, count(*) as cnt
    from cover group by 1
) c on h.code_father = c.code_father
```

  В данном случае соединяемые потоки уже отсортированы по ключу `code_father`, поэтому их повторная сортировка не требуется, а значит алгоритм соединения MERGE JOIN будет наиболее эффективным.

  К сожалению оптимизатор Firebird 5.0 не умеет распознавать такие случаи.

- соединяемые потоки очень велики. В этом случае хеш-таблица становится очень велика и уже не помещаются целиком в память. Оптимизатор Firebird 5.0 проверяет кардинальности соединяемых потоков, и если меньшая из них более миллиона записей (более точная цифра 1009 слотов * 1000 коллизий = 1009000 записей), то выбирается алгоритм соединения MERGE JOIN. В explain плане он выглядит следующим образом:

```
SELECT
  *
FROM
  BIG_1
  JOIN BIG_2 ON BIG_2.F_2 = BIG_1.F_1
```

```
Select Expression
    -> Filter
        -> Merge Join (inner)
            -> Sort (record length: 44, key length: 12)
                -> Table "BIG_2" Full Scan
            -> Sort (record length: 44, key length: 12)
                -> Table "BIG_1" Full Scan
```

1.3. Трансформация OUTER JOIN в INNER JOIN

Традиционно OUTER JOIN довольно плохо оптимизированы в Firebird.

Во-первых, в настоящее время OUTER JOIN может быть выполнен только одним алгоритмом соединения NESTED LOOP JOIN, что может быть изменено в следующих версиях. Если возможно, то будет использован индекс по ключу присоединяемой таблицы, но как мы уже видели выше - это не есть гарантия наиболее быстрого выполнения.

Во-вторых, при соединении потоков внешними соединениями порядок соединения строго фиксирован, то есть оптимизатор не может изменить его, чтобы результат оставался правильным.

Однако, если в условии WHERE существует предикат для поля "правой" (присоединяемой) таблицы, который явно не обрабатывает значение NULL, то во внешнем соединении нет смысла. В этом случае начиная с Firebird 5.0 такое соединение будет преобразовано во внутреннее, что позволяет оптимизатору применять весь спектр доступных алгоритмов соединения.

Допустим у вас есть следующий запрос:

```
SELECT
  COUNT(*)
FROM
  HORSE
  LEFT JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE FARM.CODE_COUNTRY = 1
```

Результат выполнения в Firebird 4.0:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (outer)
                -> Table "HORSE" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               345525

Current memory = 2612613792
Delta memory = 0
Max memory = 2614392048
Elapsed time = 1.524 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 2671475
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |   519623|         |         |         |
HORSE                           |   519623|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Результат выполнения в Firebird 5.0:

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "FK_FARM_COUNTRY" Range Scan (full match)
            -> Filter
                -> Table "HORSE" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_FARMBORN" Range Scan (full match)

                COUNT
=====================
               345525

Current memory = 2580089760
Delta memory = 240
Max memory = 2582802608
Elapsed time = 0.294 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 563801
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |    32787|         |         |         |
HORSE                           |         |   345525|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Выигрыш в 4 раза! Это произошло потому, что тип соединения был изменён на внутреннее, а это значит потоки можно переставлять местами.

Некоторые из вас могут возразить, а зачем мне писать изначально не эффективный запрос? Дело в том, что многие запросы пишутся динамически. Например, условие `FARM.CODE_COUNTRY = 1` может быть динамически добавлено приложением к уже существующему запросу, или запрос может быть целиком написан с помощью ORM.

Некоторые разработчики используют `LEFT JOIN` вместо `INNER JOIN` как подсказку оптимизатору: в каком порядке производить соединение таблиц. При этом используется некоторое "фейковое" условие в `WHERE` по полю "правой" таблицы, которое всегда истинно, если в правой таблице найдено соответствие по условию соединения. Для них остался обходной вариант: если в `WHERE` есть условие `IS NOT NULL` по полю "правой" таблицы, то внешние соединение не трансформируется во внутреннее. В этом случае, необходимо заменить такое "фейковое" условие на `IS NOT NULL`.

```
SELECT
  COUNT(*)
FROM
  HORSE
  LEFT JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE FARM.CODE_FARM IS NOT NULL
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (outer)
                -> Table "HORSE" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               519623

Current memory = 2580315664
Delta memory = 240
Max memory = 2582802608
Elapsed time = 1.151 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 2676533
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |   519623|         |         |         |
HORSE                           |   519623|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

1.4. Раннее вычисление инвариантных предикатов

Начиная с Firebird 5.0, если фильтрующий предикат инвариантен, и его значение равно FALSE, то извлечение записей из входного потока немедленно прекращается. Предикат является инвариантным, если его значение не зависит от полей фильтруемых потоков.

Простейшим случаем инвариантного предиката является фейковое ложное условие фильтрации 1=0. Посмотрим как запрос с таким условием выполняется в Firebird 4.0 и Firebird 5.0.

```
SELECT COUNT(*) FROM HORSE
WHERE 1=0;
```

Результат в Firebird 4.0

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

                COUNT
=====================
                    0

Current memory = 2612572768
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.137 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 552573
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
HORSE                           |   519623|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Результат в Firebird 5.0

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

                COUNT
=====================
                    0

Current memory = 2580339248
Delta memory = 176
Max memory = 2582802608
Elapsed time = 0.005 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 0
```

Как видно из приведённой выше статистики Firebird 4.0 в холостую молотил 500000 записей таблицы `HORSE`, в то время как Firebird 5.0 не обращался к ней вовсе. Это произошло потому, что Firebird 5.0 вычислил значение инвариантного предиката перед чтением таблицы `HORSE` и исключил это чтение.

Предварительное вычисление инвариантных предикатов в explain плане отображается как `Filter (preliminary)`.

Казалось бы, а какая нам польза от того, что запрос с ложным условием стал выполняться быстро? Кто будет писать такие запросы? Не забывайте запросы могут формироваться динамически и тогда польза становится очевидной.

Приведу пример более практичного применения данной оптимизации. Допустим у нас есть запрос с параметром:

```
SELECT * FROM HORSE
WHERE :A=1;
```

Здесь параметр `A` не зависит от полей фильтруемого потока, поэтому предикат `:A=1` можно вычислить предварительно. Таким образом, мы получаем эффективное включение и выключение полной выборки значений из некоторого запроса с помощью параметра.

Приведу ещё один пример, в котором используется ранее вычисление инвариантных предикатов. Допустим у нас есть таблица лошадей `HORSE`, и необходимо получить родословную лошади на глубину 5 рядов. Для этого напишем следующий рекурсивный запрос:

```
WITH RECURSIVE
  R AS (
    SELECT
      CODE_HORSE,
      CODE_FATHER,
      CODE_MOTHER,
      0 AS DEPTH
    FROM HORSE
    WHERE CODE_HORSE = ?
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_MOTHER,
      HORSE.CODE_FATHER,
      R.DEPTH + 1
    FROM R
      JOIN HORSE ON HORSE.CODE_HORSE = R.CODE_FATHER
    WHERE R.DEPTH < 5
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_MOTHER,
      HORSE.CODE_FATHER,
      R.DEPTH + 1
    FROM R
      JOIN HORSE ON HORSE.CODE_HORSE = R.CODE_MOTHER
    WHERE R.DEPTH < 5
  )
SELECT *
FROM R
```

Статистика выполнения в Firebird 4.0 выглядит так (план опущен):

```
Current memory = 2612639872
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.027 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 610
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
HORSE                           |         |     127|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Сравним с Firebird 5.0

```
Select Expression
    -> Recursion
        -> Filter
            -> Table "HORSE" as "R HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
        -> Union
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "R HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "R HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan

Current memory = 2580444832
Delta memory = 768
Max memory = 2582802608
Elapsed time = 0.024 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 252
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
HORSE                           |         |      63|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Firebird 5.0 потребовалось вдвое меньше чтений таблицы `HORSE`. Всё потому, что условие `R.DEPTH < 5` тоже является инвариантом на каждом шаге рекурсивного запроса.

1.5. Эффективное выполнение IN со списком констант

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.

Списки констант в IN, предварительно оцениваются как инварианты и кэшируются как двоичное дерево поиска, что ускоряет сравнение, если условие необходимо проверить для многих записей или если список значений длинный.

Продемонстрируем это следующим запросом:

```
SELECT
  COUNT(*)
FROM COVER
WHERE CODE_COVERRESULT+0 IN (151, 152, 156, 158, 159, 168, 170, 200, 202)
```

В данном случае CODE_COVERRESULT+0 написан умышленно, чтобы отключить использование индекса.

Результат в Firebird 4.0

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

                COUNT
=====================
                45231

Current memory = 2612795072
Delta memory = -288
Max memory = 2614392048
Elapsed time = 0.877 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 738452
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |   713407|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Результат в Firebird 5.0

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

                COUNT
=====================
                45231

Current memory = 2580573216
Delta memory = 224
Max memory = 2582802608
Elapsed time = 0.332 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 743126
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |   713407|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Несмотря на то, что количество чтений таблицы `COVER` не изменилось, запрос выполняется в 2,5 раза быстрее.

Если список очень длинный или если предикат IN не является избирательным, то сканирование индекса поддерживает поиск групп с использованием указателя одного уровня (т. е. по горизонтали), а не поиск каждой группы от корня (т. е. по вертикали), таким образом, используя одно сканирование индекса для всего списка IN.

Продемонстрируем это следующим запросом:

```
SELECT
  COUNT(*)
FROM LAB_LINE
WHERE CODE_LABTYPE IN (4, 5)
```

Результат в Firebird 4.0:

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

                COUNT
=====================
               985594

Current memory = 2614023968
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.361 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 992519
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
LAB_LINE                        |         |   985594|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Результат в Firebird 5.0:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "LAB_LINE" Access By ID
                -> Bitmap
                    -> Index "FK_LAB_LINE_LABTYPE" List Scan (full match)

                COUNT
=====================
               985594

Current memory = 2582983152
Delta memory = 176
Max memory = 2583119072
Elapsed time = 0.306 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 993103
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
LAB_LINE                        |         |   985594|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Время выполнения запроса почти не изменилось, но план стал другим. Вместо двух Range Scan и объединения масок через Or, используется новый метода доступа - однократное сканирование индекса по списку (в плане обозначается как List Scan).

1.6. Стратегия оптимизатора ALL ROWS vs FIRST ROWS

Существует две стратегии оптимизации запросов:

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

До Firebird 5.0 эти стратегии тоже существовали, но ими нельзя было управлять. По умолчанию использовалась стратегия `ALL ROWS`, однако если в запросе присутствовало ограничение количества записей выходного потока с помощью предложений `FIRST …`, `ROWS …` или `FETCH FIRST n ROWS`, то стратегия оптимизатора менялась на `FIRST ROWS`. Кроме того, для подзапросов в IN и EXISTS тоже используется стратегия `FIRST ROWS`.

Начиная с Firebird 5.0 по умолчанию используется стратегия оптимизации указанная в параметре `OptimizeForFirstRows` конфигурационного файла `firebird.conf` или `database.conf`.

`OptimizeForFirstRows = false` соответствует стратегии `ALL ROWS`, `OptimizeForFirstRows = true` соответствует стратегии `FIRST ROWS`.

Вы можете изменить стратегию оптимизатора на уровне текущей сессии с помощью оператора:

```
SET OPTIMIZE FOR {FIRST | ALL} ROWS
```

Кроме того, стратегия оптимизации может быть переопределена на уровне SQL оператора с помощью предложения `OPTIMIZE FOR`. SELECT запрос с предложением `OPTIMIZE FOR` имеет следующий синтаксис:

```
SELECT ...
FROM [...]
[WHERE ...]
[...]
[OPTIMIZE FOR {FIRST | ALL} ROWS]
```

Предложение `OPTIMIZE FOR` всегда указывает самым последним в SELECT запросе. В PSQL его необходимо указывать перед предложением `INTO`.

Источники данных могут быть конвейерными и буферизированными. Конвейерный источник данных выдает записи в процессе чтения своих входных потоков, в то время как буферизированный источник сначала должен прочитать все записи из своих входных потоков и только потом сможет выдать первую запись на свой выход. Если используется стратегия оптимизатора `FIRST ROWS`, то при построении плана запроса, оптимизатор старается избежать использования буферизирующих методов доступа, таких как внешняя сортировка SORT или соединение методом HASH JOIN.

Далее я покажу как стратегия доступа влияет на построение плана запроса. Для этого я укажу стратегию оптимизатора прямо в запросе с помощью предложения `OPTIMIZE FOR`.

Пример запроса и его плана со стратегией оптимизатора `ALL ROWS`:

```
SELECT
  HORSE.NAME AS HORSENAME,
  SEX.NAME AS SEXNAME,
  COLOR.NAME AS COLORNAME
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
ORDER BY HORSE.NAME
OPTIMIZE FOR ALL ROWS
```

```
Select Expression
    -> Sort (record length: 876, key length: 304)
        -> Filter
            -> Hash Join (inner)
                -> Nested Loop Join (inner)
                    -> Table "COLOR" Full Scan
                    -> Filter
                        -> Table "HORSE" Access By ID
                            -> Bitmap
                                -> Index "FK_HORSE_COLOR" Range Scan (full match)
                -> Record Buffer (record length: 113)
                    -> Table "SEX" Full Scan
```

Пример запроса и его плана со стратегией оптимизатора `FIRST ROWS`:

```
SELECT
  HORSE.NAME AS HORSENAME,
  SEX.NAME AS SEXNAME,
  COLOR.NAME AS COLORNAME
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
ORDER BY HORSE.NAME
OPTIMIZE FOR FIRST ROWS
```

```
Select Expression
    -> Nested Loop Join (inner)
        -> Table "HORSE" Access By ID
            -> Index "HORSE_IDX_NAME" Full Scan
        -> Filter
            -> Table "SEX" Access By ID
                -> Bitmap
                    -> Index "PK_SEX" Unique Scan
        -> Filter
            -> Table "COLOR" Access By ID
                -> Bitmap
                    -> Index "PK_COLOR" Unique Scan
```

В первом случае планировщик запросов выбрал внешнюю сортировку и HASH соединения для наиболее быстрого возврата всех записей. Во-втором случае выбрана навигация по индексу (ORDER index) и соединение с помощью NESTED LOOP, поскольку это позволяет как можно быстрее вернуть первые записи.

1.7. Улучшенный вывод планов

В выводе подробного плана теперь различаются определяемые пользователем операторы SELECT (сообщаемые как select expression), объявленные PSQL курсоры и подзапросы (sub-query). Как legacy, так и explain планы теперь также включают информацию о положении курсора (строка/столбец) внутри модуля PSQL.

Сравним вывод планов для некоторых запросов для Firebird 4.0 и Firebird 5.0.

Начнём с запроса в котором находится подзапрос:

```
SELECT *
FROM HORSE
WHERE EXISTS(SELECT * FROM COVER
             WHERE COVER.CODE_FATHER = HORSE.CODE_HORSE)
```

Подробный план в Firebird 4.0 будет выглядеть следующим образом:

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

А в Firebird 5.0 план выглядит так:

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Filter
        -> Table "HORSE" Full Scan
```

Теперь в плане чётко видно где основной запрос, а где подзапрос.

Теперь сравним как план выводится для PSQL, например для оператора `EXECUTE BLOCK`:

```
EXECUTE BLOCK
RETURNS (
   CODE_COLOR INT,
   CODE_BREED INT
)
AS
BEGIN
  FOR
    SELECT CODE_COLOR
    FROM COLOR
    INTO CODE_COLOR
  DO
    SUSPEND;

  FOR
    SELECT CODE_BREED
    FROM BREED
    INTO CODE_BREED
  DO
    SUSPEND;
END
```

В Firebird 4.0 legacy и explain план будут выведены для каждого курсора внутри блока, без дополнительных подробностей, просто один за другим.

```
PLAN (COLOR NATURAL)
PLAN (BREED NATURAL)
```

```
Select Expression
    -> Table "COLOR" Full Scan
Select Expression
    -> Table "BREED" Full Scan
```

В Firebird 5.0 перед каждым планом курсора будет выведен номер столбца и строки, где этот курсор объявлен.

```
-- line 8, column 3
PLAN (COLOR NATURAL)
-- line 15, column 3
PLAN (BREED NATURAL)
```

```
Select Expression (line 8, column 3)
    -> Table "COLOR" Full Scan
Select Expression (line 15, column 3)
    -> Table "BREED" Full Scan
```

Теперь сравним вывод explain планов, если курсор объявлен явно.

```
EXECUTE BLOCK
RETURNS (
   CODE_COLOR INT
)
AS
  DECLARE C1 CURSOR FOR (
    SELECT CODE_COLOR
    FROM COLOR
  );

  DECLARE C2 SCROLL CURSOR FOR (
    SELECT CODE_COLOR
    FROM COLOR
  );
BEGIN
  SUSPEND;
END
```

Для Firebird 4.0 план будет таким:

```
Select Expression
    -> Table "COLOR" as "C1 COLOR" Full Scan
Select Expression
    -> Record Buffer (record length: 25)
        -> Table "COLOR" as "C2 COLOR" Full Scan
```

Из плана складывается впечатление, что у таблицы `COLOR` псевдоним `C1`, хотя это не так.

В Firebird 5.0 план, намного понятнее:

```
Cursor "C1" (line 6, column 3)
    -> Table "COLOR" as "C1 COLOR" Full Scan
Cursor "C2" (scrollable) (line 11, column 3)
    -> Record Buffer (record length: 25)
        -> Table "COLOR" as "C2 COLOR" Full Scan
```

Во-первых, сразу ясно что у нас в блоке объявлены курсоры C1 и C2. Для двунаправленного курсора выведен дополнительный атрибут "scrollable".

1.8. Как получать планы хранимых процедур

Можно ли получать планы хранимых процедур по аналогии с тем как мы получаем планы для `EXECUTE BLOCK`?

Ответ и да и нет.

Если мы пойдёт простым путём, то есть попытаемся посмотреть план процедуры для следующего запроса, то ответ будет "Нет".

```
SELECT *
FROM SP_PEDIGREE(?, 5, 1)
```

```
Select Expression
    -> Procedure "SP_PEDIGREE" Scan
```

Как и ожидалось отображён план запроса верхнего уровня без деталей планов курсоров внутри хранимой процедуры. До Firebird 3.0 такие детали отображались в плане, но они были перемешаны в кучу и разобрать там что либо было очень затруднительно.

Но не расстраивайтесь. В Firebird 5.0 появился кеш подготовленных запросов, и таблица мониторинга `MON$COMPILED_STATEMENTS` отображает его содержимое. Как только мы подготовили запрос содержащий нашу хранимую процедуру, то эта процедура также попадает в кеш компилированных запросов и для неё можно посмотреть план с помощью следующего запроса:

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
WHERE CS.MON$OBJECT_NAME = 'SP_PEDIGREE'
  AND CS.MON$OBJECT_TYPE = 5
ORDER BY CS.MON$COMPILED_STATEMENT_ID DESC
FETCH FIRST ROW ONLY
```

```
Cursor "V" (scrollable) (line 19, column 3)
    -> Record Buffer (record length: 132)
        -> Nested Loop Join (inner)
            -> Window
                -> Window Partition
                    -> Record Buffer (record length: 82)
                        -> Sort (record length: 84, key length: 12)
                            -> Window Partition
                                -> Window Buffer
                                    -> Record Buffer (record length: 41)
                                        -> Procedure "SP_HORSE_INBRIDS" as "V H_INB SP_HORSE_INBRIDS" Scan
            -> Filter
                -> Table "HUE" as "V HUE" Access By ID
                    -> Bitmap
                        -> Index "HUE_IDX_ORDER" Range Scan (full match)
Select Expression (line 44, column 3)
    -> Recursion
        -> Filter
            -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
        -> Union
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
```

Кроме того, планы хранимых процедур будут отображаться в трассировке, если в конфигурации трассировки задано `log_procedure_compile = true`.

1.9. Заключение

Как видно из приведённых примеров оптимизатор в Firebird 5.0 стал значительно лучше. Разработчики Firebird проделали огромную работу, за что им огромная благодарность.

В следующий раз я расскажу о других нововведения Firebird 5.0.