Что нового в Firebird 5.0

Симонов Денис

Version 1.0 от 07.12.2023

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.

Кроме того, появились и новые возможности в языке SQL и PSQL, но в этой версии их не так много.

Одной из долгожданных возможностей стало появление встроенного инструмента для профилирования SQL и PSQL, что даёт возможность администраторам баз данных и разработчикам прикладных программ искать узкие места.

Базы данных созданные в Firebird 5.0 имеют версию ODS (On-Disk Structure) 13.1. Firebird 5.0 позволяет работать и с базами данных с ODS 13.0 (созданные в Firebird 4.0), но при этом некоторые возможности будут недоступны.

Для того чтобы переход на Firebird 5.0 был проще в утилиту командной строки `gfix` был добавлен новый переключатель `-upgrade`, который позволяет обновлять минорную версию ODS, без длительных операций backup и restore.

Далее я перечислю ключевые улучшения, сделанные в Firebird 5.0, и их краткое описание. Подробное описание всех изменений можно прочитать в "Firebird 5.0 Release Notes".

Chapter 1. Обновление ODS

Традиционным способом обновления ODS (On-Disk Structure) является выполнение backup на старой версии Firebird и restore на новой. Это довольно длительный процесс, особенно на больших базах данных.

Однако в случае обновления минорной версии ODS (номер после точки) backup/restore является избыточным (необходимо лишь добавить недостающие системные таблицы и поля, а также некоторые пакеты). Примером такого обновления является обновление ODS 13.0 (Firebird 4.0) до ODS 13.1 (Firebird 5.0), поскольку мажорная версия ODS 13 осталось той же.

Начиная с Firebird 5.0 появилась возможность обновления минорной версии ODS без длительный операция backup и restore. Для этого используется утилита `gfix` с переключателем `-upgrade`.

Ключевые моменты:

- Обновление необходимо производить вручную с помощью команды `gfix -upgrade`
- Требуется монопольный доступ к базе данных, в противном случае выдается ошибка.
- Требуется системная привилегия `USE_GFIX_UTILITY`.
- Обновление является транзакционным, все изменения отменяются в случае возникновения ошибки.
- После обновления Firebird 4.0 больше не может открывать базу данных.

Использование:

```
gfix -upgrade <dbname> -user <username> -pass <password>
```

NOTE

- Это односторонняя модификация, возврат назад невозможен. Поэтому перед обновлением сделайте копию базы данных (с помощью `nbackup b -0`), чтобы иметь точку восстановления, если что-то пойдет не так во время процесса.
- Обновление ODS с помощью `gfix -upgrade` не изменяет страницы данных пользовательских таблиц, таким образом записи не будут перепакованы с помощью нового алгоритма сжатия RLE. Но вновь вставляемые записи будут сжаты с помощью усовершенствованного RLE.

Chapter 2. Улучшение алгоритма сжатия данных

Как известно в Firebird записи таблиц располагаются на страницах данных (DP) в сжатом виде. Это сделано для того, чтобы на одной странице поместилось как можно больше записей, а это в свою очередь экономит дисковый ввод-вывод. До Firebird 5.0 для сжатия записей использовался классический алгоритм Run Length Encoding (RLE).

Классический алгоритм RLE работает следующим образом. Последовательность повторяющихся символов сокращается до управляющего байта, который определяет количество повторений, за которым следует фактический повторяемый байт. Если данные не могут быть сжаты, управляющий байт указывает, что "следующие *n* байт должны выводиться без изменений".

Управляющий байт используется следующим образом:

- `n > 0` [1 .. 127] - следующие *n* байт сохраняются как есть;
- `n < 0` [-3 .. -128] - следующий байт повторяется *n* раз, но сохраняется только один раз;
- `n = 0` - конец данных. Обычно заполняющий байт.

В основном RLE эффективен для сжатия хвостовых нулей в полях типа `VARCHAR(N)`, которые заполнены не целиком или равны NULL. Он достаточно быстрый и не сильно нагружает процессор в отличие от алгоритмов на основе словарей, таких как LHZ, ZIP, GZ.

Но у классического алгоритма RLE есть недостатки:

- максимальная степень сжатия составляет 64 раза: управляющий байт может закодировать 128 повторяющихся байтов превращая их в 2 байта. Таким образом 32000 одинаковых байт будут занимать 500 байт. Эта проблема особенно усугубилась в последнее время с приходом кодировки UTF8, где на каждый символ отводится 4 байта.
- в некоторых случаях сжатая последовательность байт может стать длиннее, чем не сжатая, если данные не сжимаемые.
- частое чередование коротких сжимаемых и не сжимаемых последовательностей дополнительно нагружает процессор, нивелирую тем самым выигрыш от экономии дискового ввода-вывода.

Поэтому в Firebird 5.0 был разработан усовершенствованный алгоритм сжатия RLE (со счётчиком переменной длины). Этот алгоритм доступен только в базах данных с ODS 13.1 и выше.

NOTE	Обновление ODS с помощью `gfix -upgrade` не изменяет страницы данных пользовательских таблиц, таким образом записи не будут перепакованы с помощью нового алгоритма сжатия RLE. Но вновь вставляемые записи будут сжаты с помощью усовершенствованного RLE.

Усовершенствованный алгоритм RLE работает следующим образом. Две ранее не

используемые длины -1 и -2 используется как специальные маркеры для более длинных сжимаемых последовательностей:

- `{-1, двух-байтный счётчик, значение байта}` - повторяющиеся последовательности длинной от 128 байт до 64 КБайт;
- `{-2, четырех-байтный счётчик, значение байта}` - повторяющиеся последовательности длинной более 64 КБайт.

Сжимаемые последовательности длинной 3 байта не имеют смысла если расположены между двумя несжимаемыми прогонами. Сжимаемые последовательности длинной от 4 до 8 байт являются пограничным случаем, поскольку они не сильно сжимаются, но увеличивают общее количество прогонов, что негативно влияет на скорость распаковки. Начиная с Firebird 5.0 фрагменты короче 8 байт не сжимаются.

Кроме того в Firebird 5.0 (ODS 13.1) есть ещё одно улучшение: если в результате применения алгоритма сжатия RLE к записи, последовательность байт оказалось длиннее (несжимаемые данные), то запись будет записана на страницу как есть и помечена специальным флагом как несжатая.

Теперь покажу на примерах как новый алгоритм RLE увеличивает производительность запросов.

Для начала отмечу, что сжатие записей совсем не бесплатная операция. В этом легко убедится выполнив два запроса:

```
SELECT COUNT(*) FROM BIG_TABLE;

SELECT COUNT(SOME_FIELD) FROM BIG_TABLE;
```

Первый запрос не использует распаковку записей, поскольку нас не интересует их содержимое (достаточно просто посчитать количество). Второй запрос вынужден делать распаковку каждой записи, чтобы убедится что в поле `SOME_FIELD` не NULL. Сначала посмотрим как это выполняется в Firebird 4.0.

```
SELECT COUNT(*)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2610594912
Delta memory = 0
Max memory = 2610680272
Elapsed time = 0.966 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4318077
```

```
SELECT COUNT(CODE_DICTIONARY)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2610596096
Delta memory = 1184
Max memory = 2610685616
Elapsed time = 1.770 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4318083
```

`1.770 - 0.966 = 0.804` - большая часть этого времени это как раз затраты на распаковку записей.

Теперь посмотрим тоже самое на Firebird 5.0.

```
SELECT COUNT(*)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2577478608
Delta memory = 176
Max memory = 2577562528
Elapsed time = 0.877 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4342385
```

```
SELECT COUNT(CODE_DICTIONARY)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2577491280
Delta memory = 12672
Max memory = 2577577520
Elapsed time = 1.267 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4342393
```

`1.267 - 0.877 = 0.390` - это в двое меньше чем в Firebird 4.0. Давайте глянем на статистику этой таблицы в Firebird 4.0 и Firebird 5.0.

Статистика в Firebird 4.0

```
WORD_DICTIONARY (265)
    Primary pointer page: 855, Index root page: 856
    Total formats: 1, used formats: 1
    Average record length: 191.83, total records: 4079052
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 670.00, compression ratio: 3.49
    Pointer pages: 19, data page slots: 59752
    Data pages: 59752, average fill: 87%
    Primary pages: 59752, secondary pages: 0, swept pages: 0
    Empty pages: 1, full pages: 59750
    Fill distribution:
         0 - 19% = 1
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 59750
```

Статистика в Firebird 5.0

```
WORD_DICTIONARY (265)
    Primary pointer page: 849, Index root page: 850
    Total formats: 1, used formats: 1
    Average record length: 215.83, total records: 4079052
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 670.00, compression ratio: 3.10
    Pointer pages: 21, data page slots: 65832
    Data pages: 65832, average fill: 88%
    Primary pages: 65832, secondary pages: 0, swept pages: 0
    Empty pages: 4, full pages: 65824
    Fill distribution:
         0 - 19% = 5
        20 - 39% = 2
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 65824
```

Из приведённой статистики видно что коэффициент сжатия даже меньше чем в Firebird 4.0. Так за счёт чего же такой колоссальный выигрыш? Для этого надо посмотреть на структуру этой таблицы:

```
CREATE TABLE WORD_DICTIONARY (
    CODE_DICTIONARY      BIGINT NOT NULL,
    CODE_PART_OF_SPEECH  INTEGER NOT NULL,
    CODE_WORD_GENDER     INTEGER NOT NULL,
    CODE_WORD_AFFIXE     INTEGER NOT NULL,
    CODE_WORD_TENSE      INTEGER DEFAULT -1 NOT NULL,
    NAME                 VARCHAR(50) NOT NULL COLLATE UNICODE_CI,
    PARAMS               VARCHAR(80),
    ANIMATE              D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    PLURAL               D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    INVARIABLE           D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    TRANSITIVE           D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    IMPERATIVE           D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    PERFECT              D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    CONJUGATION          SMALLINT,
    REFLEXIVE            D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */,
    PROHIBITION          D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK
(VALUE IN('Да', 'Нет')) */
);
```

В этой таблице хорошо могут быть сжаты только поля NAME и PARAMS. Поскольку у полей типа INTEGER есть модификатор NOT NULL, и поле занимает 4 байта, то в Firebird 5.0 такие поля не сжиматься. Поля с доменом D_BOOL в кодировке UTF8 могут быть сжаты для значения 'Да' (12 - 4 = 8 байт) и не будут для значения 'Нет' (12 - 6 = 6 байт).

Поскольку в таблице много коротких последовательностей, которые могли быть сжаты Firebird 4.0 и не сжимаются в Firebird 5.0, то в Firebird 5.0 количество обрабатываемых прогонов для распаковки меньше, за счёт чего и получаем выигрыш в производительности.

Теперь я покажу пример, где новый алгоритм RLE сильно выигрывает в сжатии. Для этого выполним следующий скрипт:

```
CREATE TABLE GOOD_ZIP
(
  ID BIGINT NOT NULL,
  NAME VARCHAR(100),
  DESCRIPTION VARCHAR(1000),
  CONSTRAINT PK_GOOD_ZIP PRIMARY KEY(ID)
);

SET TERM ^;

EXECUTE BLOCK
AS
DECLARE I BIGINT = 0;
BEGIN
  WHILE (I < 100000) DO
  BEGIN
    I = I + 1;
    INSERT INTO GOOD_ZIP (
      ID,
      NAME,
      DESCRIPTION
    )
    VALUES (
      :I,
      'OBJECT_' || :I,
      'OBJECT_' || :I
    );
  END
END^


SET TERM ;^

COMMIT;
```

А теперь посмотрим статистику таблицы GOOD_ZIP в Firebird 4.0 и Firebird 5.0.

Статистика в Firebird 4.0

```
GOOD_ZIP (128)
    Primary pointer page: 222, Index root page: 223
    Total formats: 1, used formats: 1
    Average record length: 111.09, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 4420.00, compression ratio: 39.79
    Pointer pages: 2, data page slots: 1936
    Data pages: 1936, average fill: 81%
    Primary pages: 1936, secondary pages: 0, swept pages: 0
    Empty pages: 0, full pages: 1935
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 1
        60 - 79% = 5
        80 - 99% = 1930
```

Статистика в Firebird 5.0

```
GOOD_ZIP (128)
    Primary pointer page: 225, Index root page: 226
    Total formats: 1, used formats: 1
    Average record length: 53.76, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 4420.00, compression ratio: 82.22
    Pointer pages: 1, data page slots: 1232
    Data pages: 1232, average fill: 70%
    Primary pages: 1232, secondary pages: 0, swept pages: 0
    Empty pages: 2, full pages: 1229
    Fill distribution:
         0 - 19% = 3
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1229
        80 - 99% = 0
```

Как видите в данном случае коэффициент сжатия в Firebird 5.0 в два раза выше.

И наконец рассмотрим пример с несжимаемыми данными. Для этого выполним скрипт:

```
CREATE TABLE NON_ZIP
(
  UID BINARY(16) NOT NULL,
  REF_UID_1 BINARY(16) NOT NULL,
  REF_UID_2 BINARY(16) NOT NULL
);

SET TERM ^;

EXECUTE BLOCK
AS
DECLARE I BIGINT = 0;
BEGIN
  WHILE (I < 100000) DO
  BEGIN
    I = I + 1;
    INSERT INTO NON_ZIP (
      UID,
      REF_UID_1,
      REF_UID_2
    )
    VALUES (
      GEN_UUID(),
      GEN_UUID(),
      GEN_UUID()
    );
  END
END^


SET TERM ;^

COMMIT;
```

А теперь посмотрим статистику таблицы NON_ZIP.

Статистика в Firebird 4.0

```
NON_ZIP (129)
    Primary pointer page: 2231, Index root page: 2312
    Total formats: 1, used formats: 1
    Average record length: 53.00, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 52.00, compression ratio: 0.98
    Pointer pages: 1, data page slots: 1240
    Data pages: 1240, average fill: 69%
    Primary pages: 1240, secondary pages: 0, swept pages: 0
    Empty pages: 5, full pages: 1234
    Fill distribution:
         0 - 19% = 5
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 1234
        80 - 99% = 0
```

Статистика в Firebird 5.0

```
NON_ZIP (129)
    Primary pointer page: 1587, Index root page: 1588
    Total formats: 1, used formats: 1
    Average record length: 52.00, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 52.00, compression ratio: 1.00
    Pointer pages: 1, data page slots: 1240
    Data pages: 1240, average fill: 68%
    Primary pages: 1240, secondary pages: 0, swept pages: 0
    Empty pages: 5, full pages: 1234
    Fill distribution:
         0 - 19% = 5
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 1234
        80 - 99% = 0
```

В Firebird 4.0 в результате компрессии длина записи увеличилась, Firebird 5.0 увидел, что в результате сжатия записи получаются длиннее и сохранил запись как есть.

Chapter 3. Параллельное выполнение задач

Начиная с версии 5.0 Firebird может выполнять некоторые задачи, используя несколько потоков параллельно. Часть этих задач использует параллелизм на уровне ядра Firebird, другие реализованы непосредственно в утилитах. В настоящее время на уровне ядра реализовано параллельное выполнение задач очистки (sweep) и создания индекса. Параллельное выполнение поддерживается как для автоматической, так и для ручной очистки (sweep).

В следующих версиях планируется добавить параллелизм при выполнении SQL запроса.

3.1. Параллельное выполнение задач в ядре Firebird

Для обработки задачи с несколькими потоками движок Firebird запускает дополнительные рабочие потоки и создает внутренние рабочие соединения. По умолчанию параллельное выполнение отключено. Существует два способа включить параллелизм в пользовательском соединении:

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

Некоторые утилиты (`gfix`, `gbak`) поставляемые с Firebird имеют ключ командной строки `-parallel` для установки количества параллельных рабочих процессов. Зачастую этот переключатель просто передаёт количество рабочих процессов через тег `isc_dpb_parallel_workers` при соединении с базой данных.

Новый параметр `ParallelWorkers` в `firebird.conf` устанавливает количество параллельных рабочих процессов по умолчанию, которые могут использоваться любым пользовательским соединением, выполняющим параллелизуемую задачу. Значение по умолчанию равно 1 и означает отсутствие использования дополнительных параллельных рабочих процессов. Значение в DPB имеет более высокий приоритет, чем значение в `firebird.conf`.

Для контроля количества дополнительных рабочих процессов (workers), которые может создать движок, в `firebird.conf` есть две новые настройки:

ParallelWorkers

Устанавливает количество параллельных рабочих процессов по умолчанию, используемых пользовательскими соединениями. Может быть переопределено для соединения путем использованием тега `isc_dpb_parallel_workers` в DPB.

MaxParallelWorkers

Ограничивает максимальное количество одновременно используемых рабочих процессов для данной базы данных и процесса Firebird.

Внутренние рабочие соединения создаются и управляются самим движком Firebird. Движок поддерживает пулы рабочих соединений для каждой базы данных. Количество потоков в каждом пуле ограничено значением параметра `MaxParallelWorkers`. Пулы создаются каждым процессом Firebird независимо.

В архитектуре SuperServer рабочие соединения реализованы как облегченные системные соединения, а в Classic и SuperClassic они выглядят как обычные пользовательские соединения. Все рабочие соединения встроены в процесс создания сервера. Таким образом, в классических архитектурах нет дополнительных серверных процессов. Соединения рабочих присутствуют в таблицах мониторинга. Неработающие рабочие соединения уничтожаются через 60 секунд бездействия. Кроме того, в классических архитектурах рабочие соединения уничтожаются сразу после того, как последнее пользовательское соединение отключается от базы данных.

Для архитектуры SuperServer я рекомендую ставить значения параметра `MaxParallelWorkers` равным количеству физических ядер вашего процессора (или всех процессоров). Для классической архитектуры `MaxParallelWorkers` надо ставить меньшим равным количеству физических ядер вашего процессора. В Classic ставить `MaxParallelWorkers` меньше чем количеству физических ядер имеет смысл поскольку, ограничение `MaxParallelWorkers` устанавливает на каждый процесс.

NOTE

Значение параметра `ParallelWorkers` должно быть меньшим или равным значению `MaxParallelWorkers`. У вас может возникнуть соблазн установить `ParallelWorkers = MaxParallelWorkers`, но в этом случае надо учитывать, что вы работаете с базой данных не один. Кроме того, если у вас включен автоматических sweep, то при старте он заберёт почти все ресурсы у других работающих клиентский соединений. Лучше установить `ParallelWorkers` равным половине или трети от `MaxParallelWorkers`, а при необходимости переопределять число рабочих потоков через `isc_dpb_parallel_workers` или переключатель `-parallel` в утилитах.

Далее я покажу как параллелизм влияет на время выполнения построения или перестроения индекса. Влияние параллелизма на автоматический sweep показано не будет, поскольку она стартует автоматически без нашего участия. Влияние параллелизма на ручной sweep будет продемонстрировано при рассмотрении выполнения задач утилитами Firebird.

3.1.1. Параллелизм при создании или перестройке индекса

Сравним скорость при создании индекса для таблицы `WORD_DICTIONARY`, содержащей 4079052 записей. Для чистоты эксперимента перед новым тестом перезагружаем службу Firebird. Кроме того, для того чтобы таблица была в страничном кеше выполняем

```
SELECT COUNT(*) FROM WORD_DICTIONARY;
```

Запрос для создания индекса выглядит следующим образом:

```
CREATE INDEX IDX_WORD_DICTIONARY_NAME ON WORD_DICTIONARY (NAME);
```

Статистика выполнения этого запроса с `ParallelWorkers = 1` выглядит следующим образом:

```
Current memory = 2577810256
Delta memory = 310720
Max memory = 3930465024
Elapsed time = 6.798 sec
Buffers = 153600
Reads = 11
Writes = 2273
Fetches = 4347093
```

Теперь удалим этот индекс, установим в конфиге `ParallelWorkers = 4` и `MaxParallelWorkers = 4` и перезапустим сервер. Статистика для выполнения того же запроса выглядит так:

```
Current memory = 2580355968
Delta memory = 2856432
Max memory = 4157427072
Elapsed time = 3.175 sec
Buffers = 153600
Reads = 11
Writes = 2277
Fetches = 4142838
```

Как видите время создания индекса уменьшилось в 2 с небольшим раза.

Тоже самое происходит при перестроении индекса запросом:

```
ALTER INDEX IDX_WORD_DICTIONARY_NAME ACTIVE;
```

3.2. Параллельное выполнение задач утилитами Firebird

Некоторые утилиты (`gfix`, `gbak`) поставляемые с Firebird тоже поддерживает параллельное выполнение задачи. Они используют количество параллельных рабочих процессов установленное в параметре `ParallelWorkers` в `firebird.conf`. Количество параллельных рабочих процессов можно переопределить используя ключ командной строки `-parallel`.

Я рекомендую всегда устанавливать количество параллельных процессов явно через переключатель `-parallel` или `-par`.

Параллелизм в утилитах Firebird поддерживается для следующих задач:

- Создание резервной копии с помощью утилиты `gbak`

- Восстановление из резервной копии с помощью утилиты `gbak`
- Ручной sweep с помощью утилиты `gfix`
- Обновление icu с помощью утилиты `gfix`

3.2.1. Параллелизм при выполнении резервного копирования с помощью утилиты `gbak`

Давайте посмотрим как параллелизм влияет на резервное копирование утилитой `gbak`. Естественно я буду использовать самый быстрый вариант резервного копирования через менеджер сервисов и с отключенной сборкой мусора. Для того чтобы можно было отследить время каждой операции во время резервного копирования добавим переключатель `-stat td`.

Сначала запустим резервное копирования без параллелизма:

```
gbak -b -g -par 1 "c:\fbdata\db.fdb" "d:\fbdata\db.fbk" -se localhost/3055:service_mgr
-user SYSDBA
  -pas masterkey -stat td -v -Y "d:\fbdata\5.0\backup.log"
```

Резервное копирование завершилось за 35.810 секунд.

А теперь попробуем запустить резервное копирование с использованием 4 потоков.

```
gbak -b -g -par 4 "c:\fbdata\db.fdb" "d:\fbdata\db.fbk" -se localhost/3055:service_mgr
-user SYSDBA
  -pas masterkey -stat td -v -Y "d:\fbdata\5.0\backup-4.log"
```

Резервное копирование завершилось за 18.267 секунд.

Как видите при увеличении количества параллельных обработчиков скорость резервного копирования растёт, хотя и не линейно.

NOTE

На самом деле влияние параллельных потоков на скорость резервного копирования зависит от вашего железа. Оптимальное число параллельных потоков следует подбирать экспериментально.

Любые дополнительные переключатели тоже могут изменить картину. Так например переключатель `-ZIP`, сжимающий резервную копию может свести параллелизм на нет, а может всё ещё давать ускорение копирования. Это зависит от скорости дискового накопителя, производится ли копия на тот же диск где лежит база данных и других факторов. Поэтому необходимо проводить эксперименты именно на вашем железе.

3.2.2. Параллелизм при выполнении восстановления из резервной копировании с помощью утилиты `gbak`

Теперь давайте посмотрим как параллелизм влияет на скорость восстановления из резервной копии. Восстановление из резервной копии состоит из следующих этапов:

- создании базы данных с соответствующей ODS;
- восстановление метаданных из резервной копии;
- восстановлении данных пользовательских таблиц;
- построение индексов.

Параллелизм будет задействован только на двух последних этапах.

Для того чтобы можно было отследить время каждой операции во время восстановления из резервной копии добавим переключатель `-stat td`.

Сначала запустим восстановление из резервной копии без параллелизма:

```
gbak -c -par 1 "d:\fbdata\db.fbk" "c:\fbdata\db.fdb" -se localhost/3055:service_mgr
-user SYSDBA
  -pas masterkey -v -stat td -Y "d:\fbdata\restore.log"
```

Восстановление из резервной копии завершилось за 201.590 секунд. Из них 70.73 секунды ушло на восстановление данных таблиц и 121.142 секунды на построение индексов.

А теперь попробуем запустить восстановление из резервной копии с использованием 4 потоков.

```
gbak -c -par 4 "d:\fbdata\db.fbk" "c:\fbdata\db.fdb" -se localhost/3055:service_mgr
-user SYSDBA
  -pas masterkey -v -stat td -Y "d:\fbdata\restore-4.log"
```

Восстановление из резервной копии завершилось за 116.718 секунд. Из них 26.748 секунды ушло на восстановление данных таблиц и 86.075 секунды на построение индексов.

С помощью 4 параллельных рабочих нам удалось увеличить скорость восстановления почти в два раза. При этом скорость восстановления данных выросла почти в 3 раза, а построение индексов ускорилось в 1.5 раза.

Это объясняется довольно просто, индексы не восстанавливаются параллельно, параллелизм используется только при построении больших индексов. Справочные таблицы обычно небольшие, индексы на них маленькие, а количество таких таблиц может быть большим. Поэтому на вашей базе данных цифры могут быть другими.

3.2.3. Параллельный ручной sweep с помощью утилиты `gfix`

sweep (чистка) - это сканирование всех страниц данных (DP) всех таблиц, и если на этих страницах присутствуют "мусорные" записи убрать их. Основная цель запуска sweep - это подвинуть "вверх" номер Oldest Interesting Transaction (Oldest transaction в gstat -h).

NOTE

До Firebird 3.0 sweep всегда сканировал все страницы данных. Однако начиная с Firebird 3.0 (ODS 12.0) на страницах данных (DP) и на страницах указателей на страницы данных (PP) есть специальный swept flag, который устанавливается в 1, если sweep уже просмотрел страницу данных и вычистил с неё мусор. При первой модификации записей на этот таблице флаг снова сбрасывается в 0. Начиная с Firebird 3.0 автоматический и ручной sweep пропускает страницы у которых swept флаг равен 1. Поэтому повторный sweep будет проходить намного быстрее, если конечно с момента предыдущего sweep вы не успели поменять записи на всех страницах данных базы данных.

Новые страницы данных всегда создаются с swept flag = 0. При восстановлении базы данных и резервной копии все страницы DP и PP будут с swept flag = 0.

Как правильно тестировать? Холостой sweep после восстановления из базы данных не дал разницы в однопоточном и многопоточном режиме. Поэтому я сначала провёл на восстановленной БД sweep для того чтобы следующий sweep не проверял незамусоренные страницы, а потом сделал запрос вроде такого:

```
update bigtable set field=field;
rollback;
exit;
```

Целью этого запроса было создания мусора в базе данных. Теперь можно запускать sweep для тестирования скорости его выполнения.

Сначала запустим sweep без параллелизма:

```
gfix -user SYSDBA -password masterkey -sweep -par 1 inet://localhost:3055/mydb
```

```
DESKTOP-E3INAFT Sun Oct 22 16:24:21 2023
    Sweep is started by SYSDBA
    Database "mydb"
    OIT 694, OAT 695, OST 695, Next 696


DESKTOP-E3INAFT Sun Oct 22 16:24:42 2023
    Sweep is finished
    Database "mydb"
    1 workers, time 20.642 sec
    OIT 696, OAT 695, OST 695, Next 697
```

Теперь снова делаем обновление большой таблицы и rollback, и запустим sweep с 4 параллельным рабочими.

```
gfix -user SYSDBA -password masterkey -sweep -par 4 inet://localhost:3055/mydb
```

```
DESKTOP-E3INAFT Sun Oct 22 16:26:56 2023
    Sweep is started by SYSDBA
    Database "mydb"
    OIT 697, OAT 698, OST 698, Next 699


DESKTOP-E3INAFT Sun Oct 22 16:27:06 2023
    Sweep is finished
    Database "mydb"
    4 workers, time 9.406 sec
    OIT 699, OAT 702, OST 701, Next 703
```

Как видите скорость выполнения sweep выросла в 2 с лишним раза.

3.2.4. Параллельное обновление icu с помощью утилиты `gfix`

Переключатель `-icu` позволяет перестроить индексы в базе данных с использованием нового ICU.

Дело в том, что библиотека ICU используется Firebird для поддержки COLLATION для многобайтных кодировок вроде UTF8. В Windows ICU всегда поставляется в комплекте с Firebird. В Linux же ICU обычно является системной библиотекой. При переносе файла базы данных с одного дистрибутива Linux на другой, ICU установленная в системе может иметь разную версию. Это может привести к тому, что база данных в ОС, где установлена другая версия ICU, окажется бинарно несовместимой для индексов символьных типов данных.

Поскольку перестройка индексов может быть выполнена с использованием параллелизма, то и для `gfix -icu` это тоже поддерживается.

Chapter 4. Кэш подготовленных запросов

Любой SQL запрос проходит две обязательные стадии: подготовку (компиляцию) и собственно выполнение.

Во время подготовки запроса происходит его синтаксический разбор, выделение буферов под входные и выходные сообщения, построение плана запроса и дерева его выполнения.

Если в приложении требуется многократное выполнение одного и того же запроса с разным набором входных параметров, то обычно отдельно вызывается prepare, хендл подготовленного запроса сохраняется в приложении, а затем для этого хендла вызывается execute. Это позволяется сократить затраты на переподготовку одного и того же запроса при каждом выполнении.

Начиная с Firebird 5.0 поддерживается кэш компилированных (подготовленных) запросов для каждого соединения. Это позволяет сократить затраты для повторной подготовки одних и тех же запросов, если в вашем приложении не используется явное кэширование хендлов подготовленных запросов (на глобальном уровне это не всегда просто). По умолчанию кэширование включено, порог кэширования определяется параметром `MaxStatementCacheSize` в `firebird.conf`. Его можно отключить, установив для `MaxStatementCacheSize` значение ноль. Кэш поддерживается автоматически: кэшированные операторы становятся недействительными, когда это необходимо (обычно при выполнении какого-либо оператора DDL).

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

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

Table 1. Описание столбцов таблицы `MON$COMPILED_STATEMENTS`

Наименование столбца	Тип данных	Описание
`MON$COMPILED_STATEMENT_ID`	`BIGINT`	Идентификатор скомпилированного запроса.
`MON$SQL_TEXT`	`BLOB TEXT`	Текст оператора на языке SQL. Внутри PSQL объектов текст SQL операторов не отображается.
`MON$EXPLAINED_PLAN`	`BLOB TEXT`	План оператора в explain форме.
`MON$OBJECT_NAME`	`CHAR(63)`	Имя PSQL объекта, в котором был компилирован SQL оператор.
`MON$OBJECT_TYPE`	`SMALLINT`	Тип объекта. `2` — триггер; `5` — хранимая процедура; `15` — хранимая функция.

Наименование столбца	Тип данных	Описание
MON$PACKAGE_NAME	CHAR(63)	Имя PSQL пакета.
MON$STAT_ID	INTEGER	Идентификатор статистики.

В таблицах MON$STATEMENTS и MON$CALL_STACK появился новый столбец MON$COMPILED_STATEMENT_ID, который ссылается на соответствующий подготовленный оператор в MON$COMPILED_STATEMENTS.

Таблица мониторинга MON$COMPILED_STATEMENTS позволяет легко получить планы внутренних запросов в хранимой процедуре, например вот так:

```
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
```

NOTE Обратите внимание, что одна и та же хранимая процедура может встречаться в MON$COMPILED_STATEMENTS многократно. Это связанно с тем, что в настоящее время кэш подготовленных запросов сделан для каждого соединения. В следующих версиях планируется сделать кэш скомпилированных запросов и кэш метаданных общим для всех соединений в архитектуре Super Server.

Chapter 5. Поддержка двунаправленных курсоров в сетевом протоколе

Курсор в SQL - это объект, который позволяет перемещаться по записям любого результирующего набора. С его помощью можно обработать отдельную запись базы данных, возвращаемую запросом. Различают однонаправленные и двунаправленные (прокручиваемые) курсоры.

Однонаправленный курсор не поддерживает прокрутку, то есть получение записей из такого курсора возможно только последовательно, от начала до конца курсора. Этот вид курсоров доступен в Firebird с самых ранних версий, как в PSQL (явно объявленные и неявные курсоры), так и через API.

Прокручиваемый или двунаправленные курсор, позволяет перемещаться по курсору в любом направлении, двигаться скачками и даже перемещаться на заданную позицию. Поддержка двунаправленных (прокручиваемых) курсоров впервые появилась в Firebird 3.0. Они так же доступны в PSQL и через API интерфейс.

Однако до Firebird 5.0 прокручиваемые курсоры не поддерживались на уровне сетевого протокола. Это обозначает, что вы могли использовать использовать API двунаправленных курсоров в своём приложении, только если ваше подключение происходит в embedded режиме. Начиная с Firebird 5.0 вы можете использовать API прокручиваемых курсоров даже если соединяетесь с базой данных по сетевому протоколу, при этом клиентская библиотека fbclient должна быть не ниже версии 5.0.

Если ваше приложение не использует fbclient, например написано на Java или .NET, то соответствующий драйвер должен поддерживать сетевой протокол Firebird 5.0. Например, Jaybird 5 поддерживает двунаправленные курсоры в сетевом протоколе.

Chapter 6. Трассировка события COMPILE

В Firebird 5.0 появилась возможно отслеживать новое событие трассировки: парсинг хранимых модулей. Оно позволяет отслеживать моменты парсинга хранимых модулей, соответствующее затраченное время и самое главное — планы запросов внутри этих модулей PSQL. Отслеживание плана также возможно, если модуль PSQL уже был загружен до начала сеанса трассировки; в этом случае о плане будет сообщено во время первого выполнения, замеченного сеансом трассировки.

Для отслеживания события парсинга модуля в конфигурации трассировки появились следующие параметры:

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

Допустим у нас есть следующий запрос:

```
SELECT * FROM SP_PEDIGREE(7435, 8, 1);
```

Для того чтобы в сеансе трассировки отслеживать план хранимой процедуры, необходимо установить параметр `log_procedure_compile = true`. В этом случае при подготовке этого запроса или его выполнении в логе трассировки появится событие парсинга процедуры, которое выглядит так:

```
2023-10-18T20:40:51.7620 (3920:00000000073A17C0) COMPILE_PROCEDURE
    horses (ATT_30, SYSDBA:NONE, UTF8, TCPv6:::1/54464)
    C:\Firebird\5.0\isql.exe:10960

Procedure SP_PEDIGREE:
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
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
     28 ms
```

Chapter 7. По-табличная статистика в isql

По-табличная статистика показывается сколько записей для каждой таблицы при выполнении запроса было прочитано полным сканированием, сколько с использованием индекса, сколько вставлено, обновлено или удалено и другие счётчики. Значения этих счётчиков с давних пор доступно через API функцию `isc_database_info`, что использовалось многими графическими инструментами, но не консольным инструментом `isql`. Значения этих же счётчиков можно получить через совместное использование таблиц мониторинга `MON$RECORD_STATS` и `MON$TABLE_STATS`, или в трассировке. Начиная с Firebird 5.0 эта полезная функция появилась и в `isql`.

По умолчанию вывод по-табличной статистики выключен.

Для её включения необходимо набрать команду:

```
SET PER_TAB ON;
```

А для отключения:

```
SET PER_TAB OFF;
```

Команда `SET PER_TAB` без слов `ON` или `OFF` переключает состояние вывода статистики.

Полный синтаксис этой команды можно получить используя команду `HELP SET`.

Пример вывода по-табличной статистики:

```
SQL> SET PER_TAB ON;

SQL> SELECT COUNT(*)
CON> FROM HORSE
CON> JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
CON> JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED;

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

Per table statistics:
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name   | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
BREED         |      282|         |         |         |         |         |         |         |
COLOR         |      239|         |         |         |         |         |         |         |
HORSE         |         |   519623|         |         |         |         |         |         |
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Chapter 8. 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.

8.1. 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!

8.2. 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
```

8.3. 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. Для них остался обходной вариант: если в WHERE есть условие `IS NOT NULL` по полю "правой" таблицы, то внешние соединение не трансформируется во внутреннее.

```
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|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

8.4. 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.

8.5. 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).

8.6. 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.

8.7. 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".

8.8. 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`.

Chapter 9. Новые возможности в языке SQL

9.1. 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.

9.1.1. 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.

9.1.2. 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.

9.1.3. 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*.

9.1.4. 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.

9.2. 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	По использованию `SKIP LOCKED` для организации очередей, есть отдельная статья Использованием конструкции SKIP LOCKED в Firebird 5.0

9.3. 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;
```

9.4. 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.

9.5. 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
```

9.6. 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
);
```

9.7. Улучшение литералов

9.7.1. 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'
```

9.7.2. 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
```

9.8. 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.

9.9. 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.

9.9.1. 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 2. 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
```

9.9.2. 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 3. 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
```

Chapter 10. Практический пример использования конструкции SKIP LOCKED

При разработке бизнес логики часто возникает задача по организации очередей обработки некоторых заданий. В этом случае один или несколько постановщиков ставят задания в очередь, а исполнители берут свободное невыполненное задание из очереди и выполняют его, после чего обновляют статус задания. Если исполнитель всего один, то проблем не возникает. С увеличением количества исполнителей возникает конкуренция за задачу и конфликты между исполнителями.

10.1. Подготовка базы данных

Попробуем реализовать очередь обработки заданий. Для этого создадим тестовую базу данных и создадим в ней таблицу `QUEUE_TASK`. В эту таблицу постановщики будут добавлять задачи, а исполнители брать свободные задачи и выполнять их. Скрипт создания базы данных с комментариями приведён ниже:

```
CREATE DATABASE 'inet://localhost:3055/c:\fbdata\5.0\queue.fdb'
USER SYSDBA password 'masterkey'
DEFAULT CHARACTER SET UTF8;

CREATE DOMAIN D_QUEUE_TASK_STATUS
AS SMALLINT CHECK(VALUE IN (0, 1));

COMMENT ON DOMAIN D_QUEUE_TASK_STATUS IS 'Статус завершения задачи';

CREATE TABLE QUEUE_TASK (
  ID BIGINT GENERATED BY DEFAULT AS IDENTITY NOT NULL,
  NAME VARCHAR(50) NOT NULL,
  STARTED BOOLEAN DEFAULT FALSE NOT NULL,
  WORKER_ID BIGINT,
  START_TIME TIMESTAMP,
  FINISH_TIME TIMESTAMP,
  FINISH_STATUS D_QUEUE_TASK_STATUS,
  STATUS_TEXT VARCHAR(100),
  CONSTRAINT PK_QUEUE_TASK PRIMARY KEY(ID)
);

COMMENT ON TABLE QUEUE_TASK IS 'Очередь задач';
COMMENT ON COLUMN QUEUE_TASK.ID IS 'Идентификатор задачи';
COMMENT ON COLUMN QUEUE_TASK.NAME IS 'Имя задачи';
COMMENT ON COLUMN QUEUE_TASK.STARTED IS 'Признак того что задача взята в обработку';
COMMENT ON COLUMN QUEUE_TASK.WORKER_ID IS 'Идентификатор исполнителя задачи';
COMMENT ON COLUMN QUEUE_TASK.START_TIME IS 'Время начала выполнения задачи';
COMMENT ON COLUMN QUEUE_TASK.FINISH_TIME IS 'Время завершения выполнения задачи';
COMMENT ON COLUMN QUEUE_TASK.FINISH_STATUS IS 'Статус с которым завершилось выполнение задачи 0 - успешно, 1 - с
ошибкой';
COMMENT ON COLUMN QUEUE_TASK.STATUS_TEXT IS 'Текст статуса. Если задача выполнена без ошибок, то "OK", иначе текст
ошибки';
```

Для добавления новой задачи достаточно выполнить оператор

```
INSERT INTO QUEUE_TASK(NAME) VALUES (?)
```

В данном случае мы передаём только имя задачи, на практике параметров может быть больше.

Каждый исполнитель должен выбрать одну свободную задачу и установить её признак "Взята в обработку".

Получить свободную задачу можно с помощью следующего запроса:

```
SELECT ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
```

Далее исполнитель помечает задачу как "Взята в обработку", устанавливает время старта задачи и идентификатор исполнителя. Это делается запросом:

```
UPDATE QUEUE_TASK
SET
    STARTED = TRUE,
    WORKER_ID = ?,
    START_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

После того как задача взята в обработку, начинается собственно само выполнение задачи. Когда задача выполнена необходимо установить время завершения задачи и её статус. Выполнение задачи может завершиться с ошибкой, в этом случае устанавливается соответствующий статус и сохраняется текст ошибки.

```
UPDATE QUEUE_TASK
SET
    FINISH_STATUS = ?,
    STATUS_TEXT = ?,
    FINISH_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

10.2. Скрипт моделирующий очередь заданий

Попробуем проверить нашу идею. Для этого напишем простой скрипт на языке Python.

Для написания скрипта нам потребуется установить две библиотеки:

```
pip install firebird-driver
pip install prettytable
```

Теперь можно приступить к написанию скрипта. Скрипт написан для запуска под Windows, однако его можно запускать и под Linux изменив некоторые константы и путь к библиотеке `fbclient`. Сохраним написанный скрипт его в файл `queue_exec.py`:

```
#!/usr/bin/python3

import concurrent.futures as pool
import logging
import random
import time

from firebird.driver import connect, DatabaseError
from firebird.driver import driver_config
from firebird.driver import tpb, Isolation, TraAccessMode
from firebird.driver.core import TransactionManager
from prettytable import PrettyTable

driver_config.fb_client_library.value = "c:\\firebird\\5.0\\fbclient.dll"

DB_URI = 'inet://localhost:3055/d:\\fbdata\\5.0\\queue.fdb'
DB_USER = 'SYSDBA'
DB_PASSWORD = 'masterkey'
DB_CHARSET = 'UTF8'

WORKERS_COUNT = 4  # Количество исполнителей
WORKS_COUNT = 40   # Количество задач

# set up logging to console
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)

logging.basicConfig(level=logging.DEBUG,
                    handlers=[stream_handler])

class Worker:
    """Класс Worker представляет собой исполнителя задачи"""

    def __init__(self, worker_id: int):
        self.worker_id = worker_id

    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Извлекает следующую задачу из очереди.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
        """
        cur = tnx.cursor()

        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
        """)

        row = cur.fetchone()
        cur.close()
        return row

    def __on_start_task(self, tnx: TransactionManager, task_id: int) -> None:
```

```
        """Срабатывает при старте выполнения задачи.

        Устанавливает задаче признак того, что она запущена и время старта.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
            task_id: Идентификатор задачи
        """
        cur = tnx.cursor()
        cur.execute(
            """
            UPDATE QUEUE_TASK
            SET
                STARTED = TRUE,
                WORKER_ID = ?,
                START_TIME = CURRENT_TIMESTAMP
            WHERE ID = ?
            """,
            (self.worker_id, task_id,)
        )

    @staticmethod
    def __on_finish_task(tnx: TransactionManager, task_id: int, status: int, status_text: str) -> None:
        """Срабатывает при завершении выполнения задачи.

        Устанавливает задаче время завершения и статус с которым завершилась задача.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
            task_id: Идентификатор задачи
            status: Код статуса завершения. 0 - успешно, 1 - завершено с ошибкой
            status_text: Текст статуса завершения. При успешном завершении записываем "OK",
                в противном случае текст ошибки.
        """
        cur = tnx.cursor()
        cur.execute(
            """
                UPDATE QUEUE_TASK
                SET
                    FINISH_STATUS = ?,
                    STATUS_TEXT = ?,
                    FINISH_TIME = CURRENT_TIMESTAMP
                WHERE ID = ?
            """,
            (status, status_text, task_id,)
        )

    def on_task_execute(self, task_id: int, name: str) -> None:
        """Этот метод приведён как пример функции выполнения некоторой задачи.

        В реальных задачах он будет другим и с другим набором параметров.

        Arguments:
            task_id: Идентификатор задачи
            name: Имя задачи
        """
        # выбор случайной задержки
        t = random.randint(1, 4)
        time.sleep(t * 0.01)
        # для демонстрации того, что задача может выполняться с ошибками,
        # генерируем исключение для двух из случайных чисел.
        if t == 3:
            raise Exception("Some error")
```

```
    def run(self) -> int:
        """Выполнение задачи"""
        conflict_counter = 0
        # Для параллельного выполнения каждый поток должен иметь своё соединение с БД.
        with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
            tnx = con.transaction_manager(tpb(Isolation.SNAPSHOT, lock_timeout=0, access_mode=TraAccessMode.WRITE))
            while True:
                # Извлекаем очередную задачу и ставим ей признак того что она выполняется.
                # Поскольку задача может выполниться с ошибкой, то признак старта задачи
                # выставляем в отдельной транзакции.
                tnx.begin()
                try:
                    task_row = self.__next_task(tnx)
                    # Если задачи закончились завершаем поток
                    if task_row is None:
                        tnx.commit()
                        break
                    (task_id, name,) = task_row
                    self.__on_start_task(tnx, task_id)
                    tnx.commit()
                except DatabaseError as err:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
                    else:
                        logging.exception('')
                    tnx.rollback()
                    continue

                # Выполняем задачу
                status = 0
                status_text = "OK"
                try:
                    self.on_task_execute(task_id, name)
                except Exception as err:
                    # Если при выполнении возникла ошибка,
                    # то ставим соответствующий код статуса и сохраняем текст ошибки.
                    status = 1
                    status_text = f"{err}"
                    # logging.error(status_text)

                # Сохраняем время завершения задачи и записываем статус её завершения.
                tnx.begin()
                try:
                    self.__on_finish_task(tnx, task_id, status, status_text)
                    tnx.commit()
                except DatabaseError:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
                    else:
                        logging.exception('')
                    tnx.rollback()

        return conflict_counter

def main():
    print(f"Start execute script. Works: {WORKS_COUNT}, workers: {WORKERS_COUNT}\n")

    with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
        # Чистим предыдущие задачи
        con.begin()
        with con.cursor() as cur:
            cur.execute("DELETE FROM QUEUE_TASK")
```

```
        con.commit()
        # Постановщик ставит 40 задач
        con.begin()
        with con.cursor() as cur:
            cur.execute(
                """
                EXECUTE BLOCK (CNT INTEGER = ?)
                AS
                DECLARE I INTEGER;
                BEGIN
                  I = 0;
                  WHILE (I < CNT) DO
                  BEGIN
                    I = I + 1;
                    INSERT INTO QUEUE_TASK(NAME)
                    VALUES ('Task ' || :I);
                  END
                END
                """,
                (WORKS_COUNT,)
            )
        con.commit()

    # создаём исполнителей
    workers = map(lambda worker_id: Worker(worker_id), range(WORKERS_COUNT))
    with pool.ProcessPoolExecutor(max_workers=WORKERS_COUNT) as executer:
        features = map(lambda worker: executer.submit(worker.run), workers)
        conflicts = map(lambda feature: feature.result(), pool.as_completed(features))
        conflict_count = sum(conflicts)

    # считаем статистику
    with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
        cur = con.cursor()
        cur.execute("""
            SELECT
              COUNT(*) AS CNT_TASK,
              COUNT(*) FILTER(WHERE STARTED IS TRUE AND FINISH_TIME IS NULL) AS CNT_ACTIVE_TASK,
              COUNT(*) FILTER(WHERE FINISH_TIME IS NOT NULL) AS CNT_FINISHED_TASK,
              COUNT(*) FILTER(WHERE FINISH_STATUS = 0) AS CNT_SUCCESS,
              COUNT(*) FILTER(WHERE FINISH_STATUS = 1) AS CNT_ERROR,
              AVG(DATEDIFF(MILLISECOND FROM START_TIME TO FINISH_TIME)) AS AVG_ELAPSED_TIME,
              DATEDIFF(MILLISECOND FROM MIN(START_TIME) TO MAX(FINISH_TIME)) AS SUM_ELAPSED_TIME,
              CAST(? AS BIGINT) AS CONFLICTS
            FROM QUEUE_TASK
        """, (conflict_count,))
        row = cur.fetchone()
        cur.close()

        stat_columns = ["TASKS", "ACTIVE_TASKS", "FINISHED_TASKS", "SUCCESS", "ERROR", "AVG_ELAPSED_TIME",
                        "SUM_ELAPSED_TIME", "CONFLICTS"]

        stat_table = PrettyTable(stat_columns)
        stat_table.add_row(row)
        print("\nStatistics:")
        print(stat_table)

        cur = con.cursor()
        cur.execute("""
            SELECT
              ID,
              NAME,
              STARTED,
              WORKER_ID,
              START_TIME,
```

```
            FINISH_TIME,
            FINISH_STATUS,
            STATUS_TEXT
        FROM QUEUE_TASK
    """)
    rows = cur.fetchall()
    cur.close()

    columns = ["ID", "NAME", "STARTED", "WORKER", "START_TIME", "FINISH_TIME",
               "STATUS", "STATUS_TEXT"]

    table = PrettyTable(columns)
    table.add_rows(rows)
    print("\nTasks:")
    print(table)

if __name__ == "__main__":
    main()
```

В этом скрипте постановщик ставит 40 задач, которые должны выполнить 4 исполнителя. Каждый исполнитель работает в собственном потоке. По результатам работы скрипта выводится статистика выполнения задач, а также количество конфликтов и сами задачи.

Попробуем запустит наш скрипт:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95695
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95697
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95703
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95706
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95713
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 3, Task: 3, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95728
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95734
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95736
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95741
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95744
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95749

Statistics:
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
|   40  |      0       |       40       |    28   |   12  |       43.1       |       1353       |     14    |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |         START_TIME      |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1341 |  Task 1 |   True  |   0    | 2023-07-06 15:35:29.9800 | 2023-07-06 15:35:30.0320 |    1   |  Some error |
| 1342 |  Task 2 |   True  |   0    | 2023-07-06 15:35:30.0420 | 2023-07-06 15:35:30.0800 |    1   |  Some error |
| 1343 |  Task 3 |   True  |   0    | 2023-07-06 15:35:30.0900 | 2023-07-06 15:35:30.1130 |    0   |      OK     |
| 1344 |  Task 4 |   True  |   0    | 2023-07-06 15:35:30.1220 | 2023-07-06 15:35:30.1450 |    0   |      OK     |
...
```

Из результатов выполнения скрипта видно, что 4 исполнителя постоянно конфликтуют за задачу. Чем быстрее выполняется задача и чем больше будет исполнителей, тем выше будет вероятность конфликтов.

10.3. Фраза SKIP LOCKED

Как же изменить наше решение, чтобы оно работало эффективно и без ошибок? Тут нам на помощь приходит новая конструкция `SKIP LOCKED` из Firebird 5.0.

Фраза `SKIP LOCKED` позволяет пропускать уже заблокированные записи, позволяя тем самым работать без конфликтов. Она может применяться в запросах, в которых есть вероятность возникновения конфликта обновления, то есть в запросах `SELECT ⋯ WITH LOCK`, `UPDATE` и `DELETE`. Посмотрим на её синтаксис:

```
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 ...]
```

10.4. Очередь заданий без конфликтов

Попробуем исправить наш скрипт, так чтобы исполнители не конфликтовали за задачи.

Для этого нам необходимо немного переписать запрос в методе `__next_task` класса `Worker`.

```
    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Извлекает следующую задачу из очереди.

        Arguments:
            tnx: Транзакция в которой выполняется запрос
        """
        cur = tnx.cursor()

        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
            FOR UPDATE WITH LOCK SKIP LOCKED
        """)

        row = cur.fetchone()
        cur.close()
        return row
```

Попробуем запустит исправленный скрипт:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

Statistics:
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
|   40  |      0       |       40       |    32   |   8   |       39.1       |       1048       |     0     |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |         START_TIME      |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1381 |  Task 1 |   True  |   0    | 2023-07-06 15:57:22.0360 | 2023-07-06 15:57:22.0740 |   0    |      OK     |
| 1382 |  Task 2 |   True  |   0    | 2023-07-06 15:57:22.0840 | 2023-07-06 15:57:22.1130 |   0    |      OK     |
| 1383 |  Task 3 |   True  |   0    | 2023-07-06 15:57:22.1220 | 2023-07-06 15:57:22.1630 |   0    |      OK     |
| 1384 |  Task 4 |   True  |   0    | 2023-07-06 15:57:22.1720 | 2023-07-06 15:57:22.1910 |   0    |      OK     |
| 1385 |  Task 5 |   True  |   0    | 2023-07-06 15:57:22.2020 | 2023-07-06 15:57:22.2540 |   0    |      OK     |
| 1386 |  Task 6 |   True  |   0    | 2023-07-06 15:57:22.2620 | 2023-07-06 15:57:22.3220 |   0    |      OK     |
| 1387 |  Task 7 |   True  |   0    | 2023-07-06 15:57:22.3300 | 2023-07-06 15:57:22.3790 |   1    |  Some error |
...
```

На этот раз никаких конфликтов нет. Таким образом, в Firebird 5.0 вы можете использовать

фразу `SKIP LOCKED` для того, чтобы избежать ненужных конфликтов обновлений.

Нашу очередь заданий можно ещё немного улучшить. Давайте посмотрим на план выполнения запроса

```
SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
```

Этот план выполнения не очень хороший. Запись из таблицы `QUEUE_TASK` извлекается с помощью навигации по индексу, однако сканирование индекса полное. Если таблицу `QUEUE_TASK` не очищать как мы это делали в нашем скрипте, то со временем выборка необработанных задач будет становиться всё медленней и медленней.

Можно создать индекс для поля `STARTED`. Если постановщик постоянно добавляет новые задачи, а исполнители выполняют их, то количество не начатых задач всегда меньше, количества завершённых, таким образом, этот индекс будет эффективно фильтровать задачи. Проверим это утверждение:

```
CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED;
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
                        -> Bitmap
                            -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (full match)
```

Это действительно так, но теперь используется два индекса, один для фильтрации, а второй для навигации.

Можно пойти дальше и создать композитный индекс:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED, ID);
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (partial match: 1/2)
```

Это будет эффективнее поскольку используется только один индекс для навигации, и он сканируется частично. Однако у такого индекса есть существенный недостаток, он не будет компактным.

Для решения этой проблемы можно задействовать ещё одну новую возможность из Firebird 5.0, так называемые частичные индексы.

> Частичный индекс — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением (оно называется предикатом частичного индекса). Такой индекс содержит записи только для строк, удовлетворяющих предикату.

Давайте попробуем построить такой индекс:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK (STARTED, ID) WHERE (STARTED IS FALSE);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY STARTED, ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Full Scan
```

Запись из таблицы `QUEUE_TASK` извлекается с помощью навигации по индексу `IDX_QUEUE_TASK_INACTIVE`. Несмотря на то, что сканирование индекса полное, сам по себе индекс очень компактный, поскольку содержит только ключи для которых выполняется условие `STARTED IS FALSE`. Таких записей в нормальной очереди задач всегда сильно меньше, чем записей с выполненными задачами.

В этой статье мы показали как применять новую функциональность `SKIP LOCKED`, которая появилась в Firebird 5.0. Кроме того, немного рассказано о "частичных индексах", которые тоже появилась в Firebird 5.0. Частичные индексы могут также использоваться для сложных "ограничений" уникальности. Но об этом в следующий раз.

DDL скрипт для создания базы данных, а также Python скрипт с эмуляцией очереди задач можно скачать по следующим ссылкам:

- ddl.sql
- queue_exec.py

Chapter 11. Профилирование SQL и PSQL

Одной из задач разработчика или администратора базы данных является выяснение причин "тормозов" информационной системы.

Начиная с Firebird 2.5 в их арсенале появился мощный инструмент трассировки. Трассировка является незаменимым средством при поиске узких мест приложения, оценке ресурсов, затрачиваемых при выполнении запросов, выяснении частоты каких-либо действий. Трассировка показывает статистику в максимально детализированном виде (в отличие от статистике доступной например в ISQL). В статистике не учитываются затраты на подготовку запроса и передачу данных по сети, что делает её "чище", чем данные, которые показывает ISQL. При этом трассировка оказывает очень незначительное влияние на производительность. Даже при интенсивной записи в журнал, речь обычно идет не более чем о 2-3% падения скорости выполняемых запросов.

После того, как медленные запросы "отловлены" трассировкой, можно приступать к их оптимизации. Однако такие запросы могут быть довольно сложны, а иногда и вовсе вызывать хранимые процедуры, поэтому необходим инструмент профилирования, который поможет выявить узкие места в самом запросе или вызываемом PSQL модуле. Начиная с Firebird 5.0 такой инструмент появился.

Профилировщик позволяет пользователям измерять затраты на производительность кода SQL и PSQL. Это реализовано с помощью системного пакета в движке, передающего данные в плагин профилировщика.

В этом документе движок и плагин рассматриваются как одно целое. Кроме того, подразумевается, что используется плагин профилировщика по умолчанию (Default_Profiler).

Пакет `RDB$PROFILER` может профилировать выполнение кода PSQL, собирая статистику о том, сколько раз была выполнена каждая строка, а также ее минимальное, максимальное и суммарное время выполнения (с точностью до наносекунд), а также статистику об открытии и извлечении записей из неявных и явных SQL курсоров. Кроме того, можно получать статистику SQL курсоров в разрезе источников данных (методов доступа) расширенного плана запроса.

NOTE	Несмотря на то, что время выполнения измеряется с точностью до наносекунд, не следует доверять этому результату. Процесс измерения времени выполнения имеет определённые накладные расходы, которые профилировщик пытается компенсировать. Соответственно измеренное время не может быть точным, однако это позволяет провести сравнительный анализ затрат на выполнение отдельных участков PSQL кода между собой, и выявить узкие места.

Чтобы собрать данные профилирования, пользователь должен сначала запустить сеанс профилирования с помощью функции `RDB$PROFILER.START_SESSION`. Эта функция возвращает идентификатор сеанса профилирования, который позже сохраняется в таблицах снимков профилировщика. Позже вы можете выполнить запросы к этим таблицам для анализа

пользователем. Сеанс профилировщика может быть локальным (то же соединение) или удаленным (другое соединение).

Удаленное профилирование просто перенаправляет команды управления сеансом на удаленное подключение. Таким образом, возможно, что клиент одновременно профилирует несколько соединений. Также возможно, что локально или удаленно запущенный сеанс профилирования содержит команды, выдаваемые другим соединением.

Удаленно выполняемые команды управления сеансом требуют, чтобы целевое соединение находилось в состоянии ожидания, т. е. не выполняло другие запросы. Когда целевое соединение не находится в режиме ожидания, вызов блокируется в ожидании этого состояния.

Если удаленное соединение исходит от другого пользователя, вызывающий пользователь должен иметь системную привилегию `PROFILE_ANY_ATTACHMENT`.

После запуска сеанса статистика PSQL и SQL операторов собирается в памяти. Сеанс профилирования собирает данные только об операторах, выполненных в соединении, связанном с сеансом. Данные агрегируются и сохраняются для каждого запроса (т. е. выполняемого оператора). При запросе к таблицам моментальных снимков пользователь может выполнять дополнительную агрегацию для каждого оператора или использовать вспомогательные представления, которые делают это автоматически.

Сеанс можно приостановить, чтобы временно отключить сбор статистики. Позже его можно возобновить, чтобы вернуть сбор статистики в том же сеансе.

Чтобы проанализировать собранные данные, пользователь должен сбросить данные в таблицы моментальных снимков, что можно сделать, завершив или приостановив сеанс (с параметром `FLUSH`, установленным в `TRUE`), или вызвав `RDB$PROFILER.FLUSH`. Данные сбрасываются с помощью автономной транзакции (транзакция начинается и завершается с конкретной целью обновления данных профилировщика).

Все процедуры и функции пакета `RDB$PROFILER` содержат параметр `ATTACHMENT_ID`, который следует указывать если вы управляете хотите управлять удалённым сеансом профилирования, если этот параметр равен `NULL` или не указан, то процедуры и функции управляют сеансом локального профилирования.

11.1. Старт сеанса профилирования

Для начала сеанса профилирования вам необходимо вызывать функцию `RDB$PROFILER.START_SESSION`, которая возвращает идентификатор сеанса профилирования.

Эта функция имеет следующие параметры:

- `DESCRIPTION` типа `VARCHAR(255) CHARACTER SET UTF8` по умолчанию равен `NULL`;
- `FLUSH_INTERVAL` типа `INTEGER` по умолчанию равен `NULL`;
- `ATTACHMENT_ID` типа `BIGINT` по умолчанию равен `NULL` (что обозначает `CURRENT_CONNECTION`);
- `PLUGIN_NAME` типа `VARCHAR(255) CHARACTER SET UTF8` по умолчанию равен `NULL`;

- PLUGIN_OPTIONS типа VARCHAR(255) CHARACTER SET UTF8 по умолчанию равен NULL.

В параметр DESCRIPTION вы можете передать произвольное текстовое описание сеанса.

Если параметр FLUSH_INTERVAL отличен от NULL, то производится установка интервала автоматического сброса статистики в таблицы снимков, как при вызове ручном процедуры RDB$PROFILER.SET_FLUSH_INTERVAL. Если FLUSH_INTERVAL больше нуля, то автоматический сброс статистики включён, в противном случае - выключен. Параметр FLUSH_INTERVAL измеряется в секундах.

Если ATTACHMENT_ID отличен от NULL, то сеанс профилирования запускается для удалённого соединения., в противном случае сеанс стартует для текущего соединения.

Параметр PLUGIN_NAME предназначен для указания какой плагин профилирования используется для сеанса профилирования. Если он равен NULL, то используется плагин профилирования, указанный в параметре конфигурации DefaultProfilerPlugin.

У каждого плагина профилирования, могут быть свои опции, которые можно передать в параметр PLUGIN_OPTIONS. Для плагина профилирования Default_Profiler, входящего в стандартную поставку Firebird 5.0, допустимы следующие значения NULL или 'DETAILED_REQUESTS'.

Когда используется DETAILED_REQUESTS, то в таблице PLG$PROF_REQUESTS будет храниться подробные данные запросов, то есть одна запись для каждого вызова SQL оператора. Это может привести к созданию большого количества записей, что приведет к медленной работе RDB$PROFILER.FLUSH.

Когда DETAILED_REQUESTS не используется (по умолчанию), то в таблице PLG$PROF_REQUESTS сохраняется агрегированную запись для каждого SQL оператора, используя REQUEST_ID = 0.

NOTE	Здесь под SQL оператором подразумевается подготовленный SQL запрос, который хранится в кеше подготовленных запросов. Запросы считаются одинаковыми, если они совпадают с точностью до символа, то есть если у вас семантические одинаковые запросы, но они отличаются комментарием, то для кэша подготовленных запросов это разные запросы. Подготовленные запросы, могут выполнятся многократно с различными наборами входных параметров.

11.2. Приостановка сеанса профилирования

Процедура RDB$PROFILER.PAUSE_SESSION приостанавливает текущий сеанс профилировщика (с заданным ATTACHMENT_ID). Для приостановленного сеанса статистика выполнения последующих SQL операторов не собирается.

Если параметр FLUSH имеет значение TRUE, nj таблицы моментальных снимков обновляются данными профилирования до текущего момента, в противном случае данные остаются только в памяти для последующего обновления.

Вызов RDB$PROFILER.PAUSE_SESSION(TRUE) имеет тот же эффект, что и вызов

RDB$PROFILER.PAUSE_SESSION(FALSE), за которым следует вызов RDB$PROFILER.FLUSH (с использованием того же ATTACHMENT_ID).

Входные параметры:

- FLUSH типа BOOLEAN NOT NULL по умолчанию равен FALSE;
- ATTACHMENT_ID типа BIGINT по умолчанию равен NULL (что означает CURRENT_CONNECTION).

11.3. Возобновление сеанса профилирования

Процедура RDB$PROFILER.RESUME_SESSION возобновляет текущий сеанс профилировщика (с заданным ATTACHMENT_ID), если он был приостановлен. После возобновления сеанса статистика выполнения последующих SQL операторов собирается снова.

Входные параметры:

- ATTACHMENT_ID типа BIGINT по умолчанию NULL (что означает CURRENT_CONNECTION).

11.4. Завершение сеанса профилирования

Процедура RDB$PROFILER.FINISH_SESSION завершает текущий сеанс профилировщика (с заданным ATTACHMENT_ID).

Если параметр FLUSH имеет значение TRUE, то таблицы моментальных снимков обновляются данными завершившегося сеанса (и старых завершенных сеансов, еще не присутствующих в снимке), в противном случае данные остаются только в памяти для последующего обновления.

Вызов RDB$PROFILER.FINISH_SESSION(TRUE) имеет тот же эффект, что и вызов RDB$PROFILER.FINISH_SESSION(FALSE), за которым следует вызов RDB$PROFILER.FLUSH (с использованием того же ATTACHMENT_ID).

Входные параметры:

- FLUSH типа BOOLEAN NOT NULL по умолчанию равен TRUE;
- ATTACHMENT_ID типа BIGINT по умолчанию равен NULL (что означает CURRENT_CONNECTION).

11.5. Отмена сеанса профилирования

Процедура RDB$PROFILER.CANCEL_SESSION производит отмену текущего сеанса профилирования (с заданным ATTACHMENT_ID).

Все данные сеанса, присутствующие в памяти плагина профилировщика, уничтожаются и не сбрасываются в таблицы снимков.

Уже сброшенные данные не удаляются автоматически.

Входные параметры:

- ATTACHMENT_ID типа BIGINT по умолчанию NULL (что означает CURRENT_CONNECTION).

11.6. Удаление сеансов профилирования

Процедура RDB$PROFILER.DISCARD удаляет все сеансы (с заданным ATTACHMENT_ID) из памяти, не сбрасывая их в таблицы снимков.

Если есть активная сессия профилирования, она отменяется.

Входные параметры:

- ATTACHMENT_ID типа BIGINT по умолчанию NULL (что означает CURRENT_CONNECTION).

11.7. Сброс статистики сеанса профилирования в таблицы снимков

Процедура RDB$PROFILER.FLUSH обновляет таблицы моментальных снимков данными из сеансов профилирования (с заданным ATTACHMENT_ID).

После сброса статистики, данные сохраняются в таблицах PLG$PROF_SESSIONS, PLG$PROF_STATEMENTS, PLG$PROF_RECORD_SOURCES, PLG$PROF_REQUESTS, PLG$PROF_PSQL_STATS и PLG$PROF_RECORD_SOURCE_STATS и могут быть прочитаны и проанализированы пользователем.

Данные обновляются с использованием автономной транзакции, поэтому, если процедура вызывается в shapshot транзакции, то данные не будут доступны для непосредственного чтения в той же транзакции.

После сброса статистики завершенные сеансы профилирования удаляются из памяти.

Входные параметры:

- ATTACHMENT_ID типа BIGINT по умолчанию NULL (что означает CURRENT_CONNECTION).

11.8. Установка интервала сброса статистики

Процедура RDB$PROFILER.SET_FLUSH_INTERVAL включает периодический автоматический сброс статистики (когда FLUSH_INTERVAL больше 0) или выключает его (когда FLUSH_INTERVAL равен 0).

Параметр FLUSH_INTERVAL интерпретируется как количество секунд.

Входные параметры:

- FLUSH_INTERVAL типа INTEGER NOT NULL;
- ATTACHMENT_ID типа BIGINT по умолчанию NULL (что означает CURRENT_CONNECTION).

11.9. Таблицы снимков

Таблицы снимков (а также представления и последовательности) создаются автоматически

при первом использовании профилировщика. Они принадлежат владельцу базы данных с разрешениями на чтение/запись для PUBLIC.

Когда сеанс профилирования удаляется, связанные данные в других таблицах снимков профилировщика автоматически удаляются, используя внешние ключи с опцией DELETE CASCADE.

Ниже приведен список таблиц, в которых хранятся данные профилирования.

11.9.1. Таблица PLG$PROF_SESSIONS

Таблица PLG$PROF_SESSIONS содержит информацию о сессиях профилирования.

Table 4. Описание столбцов таблицы PLG$PROF_SESSIONS

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
ATTACHMENT_ID	BIGINT	Идентификатор соединения, для которого был запущен сеанс профилирования.
USER_NAME	CHAR(63)	Имя пользователя, который запустил сеанс профилирования.
DESCRIPTION	VARCHAR(255)	Описание переданное в параметре DESCRIPTION при вызове RDB$PROFILER.START_SESSION.
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент начала сессии профилирования.
FINISH_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент окончания сессии профилирования (NULL если сессия не завершена).

Первичный ключ: PROFILE_ID.

11.9.2. Таблица PLG$PROF_STATEMENTS

Таблица PLG$PROF_STATEMENTS содержит информацию об операторах.

Table 5. Описание столбцов таблицы PLG$PROF_STATEMENTS

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.

Наименование столбца	Тип данных	Описание
PARENT_STATEMENT_ID	BIGINT	Идентификатор родительского оператора. Относится к подпрограммам.
STATEMENT_TYPE	VARCHAR(20)	Типа оператора: BLOCK, FUNCTION, PROCEDURE или TRIGGER.
PACKAGE_NAME	CHAR(63)	Имя пакета.
ROUTINE_NAME	CHAR(63)	Имя функции, процедуры или триггера.
SQL_TEXT	BLOB SUB_TYPE TEXT	SQL текст для операторов типа BLOCK.

Первичный ключ: PROFILE_ID, STATEMENT_ID.

11.9.3. Таблица PLG$PROF_REQUESTS

Таблица PLG$PROF_REQUESTS содержит статистику выполнения SQL запросов.

Если профилировщик запущен с опцией DETAILED_REQUESTS, то таблица PLG$PROF_REQUESTS будет хранить подробные данные запросов, то есть одну запись для каждого вызова оператора. Это может привести к созданию большого количества записей, что приведет к медленной работе RDB$PROFILER.FLUSH.

Когда DETAILED_REQUESTS не используется (по умолчанию), таблица PLG$PROF_REQUESTS сохраняет агрегированную запись для каждого оператора, используя REQUEST_ID = 0.

Table 6. Описание столбцов таблицы PLG$PROF_REQUESTS

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.
REQUEST_ID	BIGINT	Идентификатор запроса.
CALLER_STATEMENT_ID	BIGINT	Идентификатор вызывающего оператора.
CALLER_REQUEST_ID	BIGINT	Идентификатор вызывающего запроса.
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент старта запроса.
FINISH_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Момент завершения запроса.
TOTAL_ELAPSED_TIME	BIGINT	Накопленное время выполнения запроса (в наносекундах).

Первичный ключ: PROFILE_ID, STATEMENT_ID, REQUEST_ID.

11.9.4. Таблица PLG$PROF_CURSORS

Таблица PLG$PROF_CURSORS содержит информацию о курсорах.

Table 7. Описание столбцов таблицы PLG$PROF_CURSORS

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.
CURSOR_ID	BIGINT	Идентификатор курсора.
NAME	CHAR(63)	Имя явно объявленного курсора.
LINE_NUM	INTEGER	Номер строки PSQL, в которой определён курсор.
COLUMN_NUM	INTEGER	Номер столбца PSQL, в котором определён курсор.

Первичный ключ PROFILE_ID, STATEMENT_ID, CURSOR_ID.

11.9.5. Таблица PLG$PROF_RECORD_SOURCES

Таблица PLG$PROF_RECORD_SOURCES содержит информацию об источниках данных.

Table 8. Описание столбцов таблицы PLG$PROF_RECORD_SOURCES

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.
CURSOR_ID	BIGINT	Идентификатор курсора.
RECORD_SOURCE_ID	BIGINT	Идентификатор источника данных.
PARENT_RECORD_SOURCE_ID	BIGINT	Идентификатор родительского источника данных.
LEVEL	INTEGER	Уровень отступа для источника данных. Необходим при конструировании подробного плана.
ACCESS_PATH	BLOB SUB_TYPE TEXT	Описание метода доступа для источника данных.

Первичный ключ: PROFILE_ID, STATEMENT_ID, CURSOR_ID, RECORD_SOURCE_ID.

11.9.6. Таблица PLG$PROF_RECORD_SOURCE_STATS

Таблица PLG$PROF_RECORD_SOURCES содержит статистику по источникам данных.

Table 9. Описание столбцов таблицы `PLG$PROF_RECORD_SOURCE_STATS`

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.
REQUEST_ID	BIGINT	Идентификатор запроса.
CURSOR_ID	BIGINT	Идентификатор курсора.
RECORD_SOURCE_ID	BIGINT	Идентификатор источника данных.
OPEN_COUNTER	BIGINT	Количество открытий источника данных.
OPEN_MIN_ELAPSED_TIME	BIGINT	Минимальное время открытия источника данных (в наносекундах).
OPEN_MAX_ELAPSED_TIME	BIGINT	Максимальное время открытия источника данных (в наносекундах).
OPEN_TOTAL_ELAPSED_TIME	BIGINT	Накопленное время открытия источника данных (в наносекундах).
FETCH_COUNTER	BIGINT	Количество извлечений из источника данных.
FETCH_MIN_ELAPSED_TIME	BIGINT	Минимальное время извлечения записи из источника данных (в наносекундах).
FETCH_MAX_ELAPSED_TIME	BIGINT	Максимальное время извлечения записи из источника данных (в наносекундах).
FETCH_TOTAL_ELAPSED_TIME	BIGINT	Накопленное время извлечения записей из источника данных (в наносекундах).

Первичный ключ: `PROFILE_ID, STATEMENT_ID, REQUEST_ID, CURSOR_ID, RECORD_SOURCE_ID`.

11.9.7. Таблица `PLG$PROF_PSQL_STATS`

Таблица `PLG$PROF_PSQL_STATS` содержит PSQL статистику.

Table 10. Описание столбцов таблицы `PLG$PROF_PSQL_STATS`

Наименование столбца	Тип данных	Описание
PROFILE_ID	BIGINT	Идентификатор сессии профилирования.
STATEMENT_ID	BIGINT	Идентификатор оператора.
REQUEST_ID	BIGINT	Идентификатор запроса.
LINE_NUM	INTEGER	Номер строки в PSQL для оператора.

Наименование столбца	Тип данных	Описание
COLUMN_NUM	INTEGER	Номер столбца в PSQL для оператора.
COUNTER	BIGINT	Количество выполнений для номера строки/столбца.
MIN_ELAPSED_TIME	BIGINT	Минимальное время выполнения (в наносекундах) для строки/столбца
MAX_ELAPSED_TIME	BIGINT	Максимальное время выполнения (в наносекундах) для строки/столбца
TOTAL_ELAPSED_TIME	BIGINT	Накопленное время выполнения (в наносекундах) для строки/столбца

Первичный ключ: `PROFILE_ID, STATEMENT_ID, REQUEST_ID, LINE_NUM, COLUMN_NUM`.

11.10. Вспомогательные представления

Помимо таблиц снимков, плагин профилирования создаёт вспомогательные представления. Эти представления помогают извлекать данные профилирования, агрегированные на уровне операторов.

Вспомогательные представления являются предпочтительным способом анализа данных профилирования для быстрого нахождения "горячих точек". Их также можно использовать совместно с таблицами моментальных снимков. После того, как "горячие точки" найдены, можно детализировать данные на уровне запросов к таблицам.

Ниже приведен список представлений профилировщика `Default_Profiler`.

`PLG$PROF_PSQL_STATS_VIEW`

агрегированная PSQL статистика в сеансе профилирования.

`PLG$PROF_RECORD_SOURCE_STATS_VIEW`

агрегированная статистика по источникам данных в сеансе профилирования.

`PLG$PROF_STATEMENT_STATS_VIEW`

агрегированная статистика SQL операторов в сеансе профилирования.

В данном документе я не буду приводит сходный код этих представлений и описание их столбцов, вы всегда сможете посмотреть текст этих представлений самостоятельно. Описание столбцов есть в "Руководстве по языку SQL Firebird 5.0".

11.11. Режимы запуска профилировщика

Перед тем как мы перейдём к реальным примерам использования профилировщика, я продемонстрирую разницу между различными режимами запуска плагина профилирования.

11.11.1. Опция DETAILED_REQUESTS

NOTE В статистику профилирования попадает в том числе и запрос `SELECT RDB$PROFILER.START_SESSION()` … и статистика запуска функции `RDB$PROFILER.START_SESSION`. Для того чтобы ограничить вывод статистики я добавляю фильтр по тексту запроса, в данном случае вывожу статистику только для тех запросов в которых встречается 'FROM HORSE'.

```
SELECT RDB$PROFILER.START_SESSION('Profile without "DETAILED_REQUESTS"')
FROM RDB$DATABASE;

SELECT COUNT(*) FROM HORSE;

SELECT COUNT(*) FROM HORSE;

SELECT RDB$PROFILER.START_SESSION('Profile with "DETAILED_REQUESTS"',
  NULL, NULL, NULL, 'DETAILED_REQUESTS')
FROM RDB$DATABASE;

SELECT COUNT(*) FROM HORSE;

SELECT COUNT(*) FROM HORSE;

EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION;

COMMIT;

SELECT
  S.DESCRIPTION,
  V.*
FROM PLG$PROF_STATEMENT_STATS_VIEW V
JOIN PLG$PROF_SESSIONS S ON S.PROFILE_ID = V.PROFILE_ID
WHERE V.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
DESCRIPTION                     Profile without "DETAILED_REQUESTS"
PROFILE_ID                      12
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
PARENT_STATEMENT_ID             <null>
PARENT_STATEMENT_TYPE           <null>
PARENT_ROUTINE_NAME             <null>
SQL_TEXT                        13e:9
SELECT COUNT(*) FROM HORSE
COUNTER                         1
MIN_ELAPSED_TIME                <null>
MAX_ELAPSED_TIME                <null>
TOTAL_ELAPSED_TIME              <null>
AVG_ELAPSED_TIME                <null>

DESCRIPTION                     Profile with "DETAILED_REQUESTS"
PROFILE_ID                      13
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
PARENT_STATEMENT_ID             <null>
PARENT_STATEMENT_TYPE           <null>
PARENT_ROUTINE_NAME             <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
COUNTER                         2
MIN_ELAPSED_TIME                165498198
MAX_ELAPSED_TIME                235246029
TOTAL_ELAPSED_TIME              400744227
AVG_ELAPSED_TIME                200372113
```

Как видите если, запускать сеанс профилировщика без опции 'DETAILED_REQUESTS', то представление даёт нам меньше подробностей. Как минимум отсутствует статистика выполнения запроса, и показано будто запрос был выполнен один раз. Попробуем детализировать эти данные с помощью запроса к таблицам моментальных снимков.

Сначала посмотрим детали сессии без опции 'DETAILED_REQUESTS'.

```
SELECT
  S.PROFILE_ID,
  S.DESCRIPTION,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID =
STMT.STATEMENT_ID
WHERE S.PROFILE_ID = 12
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      12
DESCRIPTION                     Profile without "DETAILED_REQUESTS"
REQUEST_ID                      0
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:9
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:48:59.2250
FINISH_TIMESTAMP                <null>
TOTAL_ELAPSED_TIME              <null>
```

А теперь сравним с сессией с опцией 'DETAILED_REQUESTS'.

```
SELECT
  S.PROFILE_ID,
  S.DESCRIPTION,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID =
STMT.STATEMENT_ID
WHERE S.PROFILE_ID = 13
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      13
DESCRIPTION                     Profile with "DETAILED_REQUESTS"
REQUEST_ID                      2474
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:49:01.6540
FINISH_TIMESTAMP                2023-11-09 15:49:02.8360
TOTAL_ELAPSED_TIME              165498198

PROFILE_ID                      13
DESCRIPTION                     Profile with "DETAILED_REQUESTS"
REQUEST_ID                      2475
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:49:02.8470
FINISH_TIMESTAMP                2023-11-09 15:49:04.0980
TOTAL_ELAPSED_TIME              235246029
```

Таким образом, если профилировщик запускается без опции 'DETAILED_REQUESTS' все запуски одного и того же SQL оператора, будут выглядеть как один запуск, в котором накапливается статистика. Непосредственно в таблице `PLG$PROF_REQUESTS`, статистика вообще не учитывается без опции 'DETAILED_REQUESTS', но она агрегируется в других таблицах.

```sql
SELECT
  R.PROFILE_ID,
  R.STATEMENT_ID,
  RS.CURSOR_ID,
  RS.RECORD_SOURCE_ID,
  RS.PARENT_RECORD_SOURCE_ID,
  RS."LEVEL",
  RS.ACCESS_PATH,
  RSS.OPEN_COUNTER,
  RSS.OPEN_MIN_ELAPSED_TIME,
  RSS.OPEN_MAX_ELAPSED_TIME,
  RSS.OPEN_TOTAL_ELAPSED_TIME,
  RSS.FETCH_COUNTER,
  RSS.FETCH_MIN_ELAPSED_TIME,
  RSS.FETCH_MAX_ELAPSED_TIME,
  RSS.FETCH_TOTAL_ELAPSED_TIME
FROM
  PLG$PROF_REQUESTS R
  JOIN PLG$PROF_RECORD_SOURCES RS
    ON RS.PROFILE_ID = R.PROFILE_ID AND
       RS.STATEMENT_ID = R.STATEMENT_ID
  JOIN PLG$PROF_RECORD_SOURCE_STATS RSS
    ON RSS.PROFILE_ID = R.PROFILE_ID AND
       RSS.STATEMENT_ID = R.STATEMENT_ID AND
       RSS.REQUEST_ID = R.REQUEST_ID AND
       RSS.CURSOR_ID = RS.CURSOR_ID AND
       RSS.RECORD_SOURCE_ID = RS.RECORD_SOURCE_ID
WHERE R.PROFILE_ID = 12
  AND R.STATEMENT_ID = 2149
ORDER BY RSS.REQUEST_ID, RSS.RECORD_SOURCE_ID
```

```
PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                1
PARENT_RECORD_SOURCE_ID         <null>
LEVEL                           0
ACCESS_PATH                     140:f
Select Expression
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           10266
OPEN_MAX_ELAPSED_TIME           10755
OPEN_TOTAL_ELAPSED_TIME         21022
FETCH_COUNTER                   4
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          191538868
FETCH_TOTAL_ELAPSED_TIME        356557956

PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                2
PARENT_RECORD_SOURCE_ID         1
LEVEL                           1
ACCESS_PATH                     140:10
-> Aggregate
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           9777
OPEN_MAX_ELAPSED_TIME           9777
OPEN_TOTAL_ELAPSED_TIME         19555
FETCH_COUNTER                   4
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          191538379
FETCH_TOTAL_ELAPSED_TIME        356556489

PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:11
-> Table "HORSE" Full Scan
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           3911
OPEN_TOTAL_ELAPSED_TIME         6355
FETCH_COUNTER                   1039248
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          905422
FETCH_TOTAL_ELAPSED_TIME        330562264
```

Тут видно, что вся статистика "удвоена", поскольку запрос запускался два раза. А теперь посмотрим на статистику с 'DETAILED_REQUESTS'.

```
SELECT
  R.PROFILE_ID,
  R.STATEMENT_ID,
  RS.CURSOR_ID,
  RS.RECORD_SOURCE_ID,
  RS.PARENT_RECORD_SOURCE_ID,
  RS."LEVEL",
  RS.ACCESS_PATH,
  RSS.OPEN_COUNTER,
  RSS.OPEN_MIN_ELAPSED_TIME,
  RSS.OPEN_MAX_ELAPSED_TIME,
  RSS.OPEN_TOTAL_ELAPSED_TIME,
  RSS.FETCH_COUNTER,
  RSS.FETCH_MIN_ELAPSED_TIME,
  RSS.FETCH_MAX_ELAPSED_TIME,
  RSS.FETCH_TOTAL_ELAPSED_TIME
FROM
  PLG$PROF_REQUESTS R
  JOIN PLG$PROF_RECORD_SOURCES RS
    ON RS.PROFILE_ID = R.PROFILE_ID AND
       RS.STATEMENT_ID = R.STATEMENT_ID
  JOIN PLG$PROF_RECORD_SOURCE_STATS RSS
    ON RSS.PROFILE_ID = R.PROFILE_ID AND
       RSS.STATEMENT_ID = R.STATEMENT_ID AND
       RSS.REQUEST_ID = R.REQUEST_ID AND
       RSS.CURSOR_ID = RS.CURSOR_ID AND
       RSS.RECORD_SOURCE_ID = RS.RECORD_SOURCE_ID
WHERE R.PROFILE_ID = 13
  AND R.STATEMENT_ID = 2149
ORDER BY RSS.REQUEST_ID, RSS.RECORD_SOURCE_ID
```

```
PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                1
PARENT_RECORD_SOURCE_ID         <null>
LEVEL                           0
ACCESS_PATH                     140:14
Select Expression
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           20044
OPEN_MAX_ELAPSED_TIME           20044
OPEN_TOTAL_ELAPSED_TIME         20044
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          165438065
```

```
FETCH_TOTAL_ELAPSED_TIME        165438065

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                2
PARENT_RECORD_SOURCE_ID         1
LEVEL                           1
ACCESS_PATH                     140:15
-> Aggregate
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           19066
OPEN_MAX_ELAPSED_TIME           19066
OPEN_TOTAL_ELAPSED_TIME         19066
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          165437576
FETCH_TOTAL_ELAPSED_TIME        165437576

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:16
-> Table "HORSE" Full Scan
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           2444
OPEN_TOTAL_ELAPSED_TIME         2444
FETCH_COUNTER                   519624
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          892222
FETCH_TOTAL_ELAPSED_TIME        161990420

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                1
PARENT_RECORD_SOURCE_ID         <null>
LEVEL                           0
ACCESS_PATH                     140:14
Select Expression
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           12711
OPEN_MAX_ELAPSED_TIME           12711
OPEN_TOTAL_ELAPSED_TIME         12711
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          488
FETCH_MAX_ELAPSED_TIME          235217674
```

```
FETCH_TOTAL_ELAPSED_TIME        235218163

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                2
PARENT_RECORD_SOURCE_ID         1
LEVEL                           1
ACCESS_PATH                     140:15
-> Aggregate
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           11244
OPEN_MAX_ELAPSED_TIME           11244
OPEN_TOTAL_ELAPSED_TIME         11244
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          235217674
FETCH_TOTAL_ELAPSED_TIME        235217674

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:16
-> Table "HORSE" Full Scan
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           2444
OPEN_TOTAL_ELAPSED_TIME         2444
FETCH_COUNTER                   519624
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          675155
FETCH_TOTAL_ELAPSED_TIME        196082602
```

Для сессии с опцией 'DETAILED_REQUESTS' мы видим что статистика собрана отдельно для каждого запуска SQL оператора.

NOTE

Опция 'DETAILED_REQUESTS' создаёт по одной записи в таблице `PLG$PROF_REQUESTS` не только для каждого SQL запроса верхнего уровня, но и для каждого вызова хранимой процедуры или функции. Таким образом, если у вас вызывается PSQL функция для какого-то поля в предложении `SELECT`, то в `PLG$PROF_REQUESTS` будет столько записей с вызовом это функции, сколько записей было отфетчено. Это может сильно замедлить сброс статистики профилирования. Поэтому не стоит использовать опцию 'DETAILED_REQUESTS' для любого сеанса профилирования. Для поиска "узких мест" в некотором запросе, достаточно будет оставлять значение параметра `PLUGIN_OPTIONS` умолчательным значением.

11.11.2. Запуск профилировщика на удалённом соединении

Для запуска сеанса на профилирования на удалённом соединении вам необходимо узнать идентификатор этого соединения. Этом можно сделать с помощью таблицы мониторинга `MON$ATTACHMENTS` или выполнить запрос с контекстной переменной `CURRENT_CONNECTION` в удалённом сеансе, и установить этот идентификатор в качестве значения параметра `ATTACHMENT_ID` функции `RDB$PROFILER.START_SESSION`.

Запрос идентификатора соединения в сессии 1

```
select current_connection from rdb$database;
```

```
   CURRENT_CONNECTION
=====================
                   29
```

Запуск сеанса профилирования на удалённом компьютере в сессии 2

```
SELECT RDB$PROFILER.START_SESSION('Profile with "DETAILED_REQUESTS"',
  NULL, 29, NULL, 'DETAILED_REQUESTS')
FROM RDB$DATABASE;
```

Запрос или запросы которые профилируем в сессии 1

```
select current_connection from rdb$database;
```

Останавливаем удалённое профилирование в сессии 2

```
EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION(TRUE, 29);

COMMIT;
```

Теперь мы можем посмотреть результат профилирования в любой соединении.

```
SELECT
  S.PROFILE_ID,
  S.ATTACHMENT_ID,
  S.START_TIMESTAMP AS SESSION_START,
  S.FINISH_TIMESTAMP AS SESSION_FINISH,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID =
STMT.STATEMENT_ID
WHERE S.ATTACHMENT_ID = 29
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      14
ATTACHMENT_ID                   29
SESSION_START                   2023-11-09 16:56:39.1640
SESSION_FINISH                  2023-11-09 16:57:41.0010
REQUEST_ID                      3506
STATEMENT_ID                    3506
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:1
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 16:57:29.1010
FINISH_TIMESTAMP                2023-11-09 16:57:30.4800
TOTAL_ELAPSED_TIME              82622
```

11.12. Примеры использования профилировщика для поиска "узких мест"

Теперь когда вы познакомились с различными режимами запуска сеанса профилирования, настало время показать как профилировщик поможет найти вам "узкие места" в ваших SQL запросах и PSQL модулях.

Допустим с помощью трассировки вы нашли такой медленный запрос:

```
SELECT
 CODE_HORSE,
 BYDATE,
 HORSENAME,
 FRISK,
 OWNERNAME
FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
```

Нам необходимо разобраться в причине "тормозов" и исправить их. Первое что стоит сделать - исключить время фетча записей на клиента. Для этого можно просто обернуть этот запрос в другой запрос, который просто будет вычислять количество записей. Таким образом, мы гарантировано прочитаем все записи, но при этом нам необходимо переслать на клиента всего 1 запись.

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

Статистика выполнения данного запроса выглядит так:

```
                COUNT
=====================
                  240

Current memory = 554444768
Delta memory = 17584
Max memory = 554469104
Elapsed time = 2.424 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124985
```

NOTE В данном случае список полей можно было бы просто заменить на `COUNT(*)` без заворачивания запроса в производную таблицу, однако в общем случае в предложении `SELECT` могут быть различные выражения в том числе и подзапросы, поэтому лучше делать так как я показал.

Теперь можно запускать запрос в профилировщике:

```
SELECT RDB$PROFILER.START_SESSION('Profile procedure SP_SOME_STAT')
FROM RDB$DATABASE;

SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);

EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION;

COMMIT;
```

Для начала посмотрим статистику PSQL:

```
SELECT
  ROUTINE_NAME,
  LINE_NUM,
  COLUMN_NUM,
  COUNTER,
  TOTAL_ELAPSED_TIME,
  AVG_ELAPSED_TIME
FROM PLG$PROF_PSQL_STATS_VIEW STAT
WHERE STAT.PROFILE_ID = 5;
```

```
ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
==================== ============ ============ =============== ===================== =====================
SF_RACETIME_TO_SEC             22            5           67801            1582095600                 23334
SF_RACETIME_TO_SEC             18            3           67801              90068700                  1328
SF_RACETIME_TO_SEC             27            5           67801              53903300                   795
SF_RACETIME_TO_SEC             31            5           67801              49835400                   735
SP_SOME_STAT                   16            3               1              43414600              43414600
SF_RACETIME_TO_SEC             25            5           67801              42623200                   628
SF_RACETIME_TO_SEC             34            5           67801              37339200                   550
SF_RACETIME_TO_SEC             14            3           67801              35822000                   528
SF_RACETIME_TO_SEC             29            5           67801              34874400                   514
SF_RACETIME_TO_SEC             32            5           67801              24093200                   355
SF_RACETIME_TO_SEC             15            3           67801              23832900                   351
SF_RACETIME_TO_SEC              6            1           67801              15985600                   235
SF_RACETIME_TO_SEC             26            5           67801              15625500                   230
SP_SOME_STAT                   38            5             240               3454800                 14395
SF_SEC_TO_RACETIME             20            3             240                549900                  2291
SF_SEC_TO_RACETIME             31            3             240                304100                  1267
SF_SEC_TO_RACETIME             21            3             240                294200                  1225
SF_SEC_TO_RACETIME             16            3             240                293900                  1224
SF_RACETIME_TO_SEC              7            1           67801                202400                     2
SF_RACETIME_TO_SEC              8            1           67801                186100                     2

ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
==================== ============ ============ =============== ===================== =====================
SF_RACETIME_TO_SEC             20            3           67801                168400                     2
SF_RACETIME_TO_SEC              9            1           67801                156700                     2
SF_RACETIME_TO_SEC             12            1           67801                153900                     2
SF_RACETIME_TO_SEC             10            1           67801                153300                     2
SF_SEC_TO_RACETIME             18            3             240                148600                   619
SF_RACETIME_TO_SEC             16            3           67801                127100                     1
SF_SEC_TO_RACETIME             17            3             240                 92100                   383
SF_SEC_TO_RACETIME              8            1             240                 89200                   371
SF_RACETIME_TO_SEC             11            1           67801                 69500                     1
SF_SEC_TO_RACETIME             28            3             240                 16600                    69
SF_RACETIME_TO_SEC              5            1           67801                  7800                     0
SF_SEC_TO_RACETIME             11            1             240                  2000                     8
SF_SEC_TO_RACETIME             10            1             240                  1800                     7
SF_SEC_TO_RACETIME              9            1             240                  1200                     5
SP_SOME_STAT                   37            5             240                   500                     2
SF_SEC_TO_RACETIME             13            3             240                   500                     2
SF_SEC_TO_RACETIME              7            1             240                   400                     1
```

Из этой статистики видно, что по суммарному времени выполнения лидирует оператор, который находится в строке 22 функции SF_RACETIME_TO_SEC. Среднее время выполнения этого оператора невелико, но он вызывается 67801 раз. Тут два варианта для оптимизации: либо оптимизировать саму функцию SF_RACETIME_TO_SEC (оператор в строке 22), либо сократить количество вызовов этой функции.

Давайте посмотрим содержимое нашей процедуры SP_SOME_STAT.

```
1 CREATE OR ALTER PROCEDURE SP_SOME_STAT (
2     A_CODE_BREED INTEGER,
3     A_MIN_FRISK  VARCHAR(9),
4     A_YEAR_BEGIN SMALLINT,
```

```
5     A_YEAR_END   SMALLINT
6 )
7 RETURNS (
8     CODE_HORSE BIGINT,
9     BYDATE     DATE,
10    HORSENAME  VARCHAR(50),
11    FRISK      VARCHAR(9),
12    OWNERNAME  VARCHAR(120)
13 )
14 AS
15 BEGIN
16   FOR
17     SELECT
18       TL.CODE_HORSE,
19       TRIAL.BYDATE,
20       H.NAME,
21       SF_SEC_TO_RACETIME(TL.TIME_PASSED_SEC) AS FRISK
22     FROM
23       TRIAL_LINE TL
24       JOIN TRIAL ON TRIAL.CODE_TRIAL = TL.CODE_TRIAL
25       JOIN HORSE H ON H.CODE_HORSE = TL.CODE_HORSE
26     WHERE TL.TIME_PASSED_SEC <= SF_RACETIME_TO_SEC(:A_MIN_FRISK)
27       AND TRIAL.CODE_TRIALTYPE = 2
28       AND H.CODE_BREED = :A_CODE_BREED
29       AND EXTRACT(YEAR FROM TRIAL.BYDATE) BETWEEN :A_YEAR_BEGIN AND :A_YEAR_END
30     INTO
31       CODE_HORSE,
32       BYDATE,
33       HORSENAME,
34       FRISK
35   DO
36   BEGIN
37     OWNERNAME = NULL;
38     SELECT
39       FARM.NAME
40     FROM
41       (
42         SELECT
43           R.CODE_FARM
44         FROM REGISTRATION R
45         WHERE R.CODE_HORSE = :CODE_HORSE
46           AND R.CODE_REGTYPE = 6
47           AND R.BYDATE <= :BYDATE
48         ORDER BY R.BYDATE DESC
49         FETCH FIRST ROW ONLY
50       ) OWN
51       JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
52     INTO OWNERNAME;
53
54     SUSPEND;
55   END
```

```
56 END
```

Функция SF_RACETIME_TO_SEC вызывает в строке с номером 21:

```
    WHERE TL.TIME_PASSED_SEC <= SF_RACETIME_TO_SEC(:A_MIN_FRISK)
```

Очевидно это условие проверятся многократно для каждой записи. Многократное выполнении этой функции вносит существенный вклад в общее время выполнения. Если посмотреть, на сам вызов функции, то можно заметить, что аргументы функции не зависят от источника данных, то есть значение функции инвариантно. Это обозначает, что мы можем вынести её вычисление за пределы запроса. Таким образом, можно переписать нашу процедуру так:

```
1 CREATE OR ALTER PROCEDURE SP_SOME_STAT (
2     A_CODE_BREED INTEGER,
3     A_MIN_FRISK  VARCHAR(9),
4     A_YEAR_BEGIN SMALLINT,
5     A_YEAR_END   SMALLINT
6 )
7 RETURNS (
8     CODE_HORSE BIGINT,
9     BYDATE     DATE,
10    HORSENAME  VARCHAR(50),
11    FRISK      VARCHAR(9),
12    OWNERNAME  VARCHAR(120)
13 )
14 AS
15   DECLARE TIME_PASSED NUMERIC(18, 3);
16 BEGIN
17   TIME_PASSED = SF_RACETIME_TO_SEC(:A_MIN_FRISK);
18   FOR
19     SELECT
20       TL.CODE_HORSE,
21       TRIAL.BYDATE,
22       H.NAME,
23       SF_SEC_TO_RACETIME(TL.TIME_PASSED_SEC) AS FRISK
24     FROM
25       TRIAL_LINE TL
26       JOIN TRIAL ON TRIAL.CODE_TRIAL = TL.CODE_TRIAL
27       JOIN HORSE H ON H.CODE_HORSE = TL.CODE_HORSE
28     WHERE TL.TIME_PASSED_SEC <= :TIME_PASSED
29       AND TRIAL.CODE_TRIALTYPE = 2
30       AND H.CODE_BREED = :A_CODE_BREED
31       AND EXTRACT(YEAR FROM TRIAL.BYDATE) BETWEEN :A_YEAR_BEGIN AND :A_YEAR_END
32     INTO
33       CODE_HORSE,
34       BYDATE,
35       HORSENAME,
```

```
36        FRISK
37    DO
38    BEGIN
39      OWNERNAME = NULL;
40      SELECT
41        FARM.NAME
42      FROM
43        (
44          SELECT
45            R.CODE_FARM
46          FROM REGISTRATION R
47          WHERE R.CODE_HORSE = :CODE_HORSE
48            AND R.CODE_REGTYPE = 6
49            AND R.BYDATE <= :BYDATE
50          ORDER BY R.BYDATE DESC
51          FETCH FIRST ROW ONLY
52        ) OWN
53        JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
54      INTO OWNERNAME;
55
56      SUSPEND;
57    END
58  END
```

Попробуем выполнить наш запрос с учётом изменений процедуры SP_SOME_STAT:

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

```
                 COUNT
=====================
                   240

Current memory = 555293472
Delta memory = 288
Max memory = 555359872
Elapsed time = 0.134 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124992
```

2.424 sec vs 0.134 sec - ускорение существенно. Можно ли сделать лучше. Давайте ещё раз запустим сеанс профилирования. Новый идентификатор сеанса равен 6.

Посмотрим статистику PSQL:

```
SELECT
  ROUTINE_NAME,
  LINE_NUM,
  COLUMN_NUM,
  COUNTER,
  TOTAL_ELAPSED_TIME,
  AVG_ELAPSED_TIME
FROM PLG$PROF_PSQL_STATS_VIEW STAT
WHERE STAT.PROFILE_ID = 6;
```

```
ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
==================== ============ ============ =============== ===================== =====================
SP_SOME_STAT                   18            3               1              10955200              10955200
SP_SOME_STAT                   40            5             240               3985800                 16607
SF_SEC_TO_RACETIME             20            3             240                508100                  2117
SF_SEC_TO_RACETIME             31            3             240                352700                  1469
SF_SEC_TO_RACETIME             21            3             240                262200                  1092
SF_SEC_TO_RACETIME             16            3             240                257300                  1072
SF_SEC_TO_RACETIME             18            3             240                156400                   651
SP_SOME_STAT                   17            3               1                141500                141500
SF_SEC_TO_RACETIME              8            1             240                125700                   523
SF_SEC_TO_RACETIME             17            3             240                 94100                   392
SF_RACETIME_TO_SEC             22            5               1                 83400                 83400
SF_SEC_TO_RACETIME             28            3             240                 38700                   161
SF_SEC_TO_RACETIME             10            1             240                 20800                    86
SF_SEC_TO_RACETIME             11            1             240                 20200                    84
SF_SEC_TO_RACETIME              9            1             240                 16200                    67
SF_RACETIME_TO_SEC              6            1               1                  7100                  7100
SF_SEC_TO_RACETIME              7            1             240                  6600                    27
SF_RACETIME_TO_SEC             27            5               1                  5800                  5800
SF_RACETIME_TO_SEC             18            3               1                  5700                  5700
SF_SEC_TO_RACETIME             13            3             240                  5700                    23

ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
==================== ============ ============ =============== ===================== =====================
SF_RACETIME_TO_SEC             32            5               1                  4600                  4600
SP_SOME_STAT                   39            5             240                  4400                    18
SF_RACETIME_TO_SEC             14            3               1                  4300                  4300
SF_RACETIME_TO_SEC             34            5               1                  3500                  3500
SF_RACETIME_TO_SEC             25            5               1                  3300                  3300
SF_RACETIME_TO_SEC             31            5               1                  2900                  2900
SF_RACETIME_TO_SEC             29            5               1                  2800                  2800
SF_RACETIME_TO_SEC             15            3               1                  1600                  1600
SF_RACETIME_TO_SEC             26            5               1                  1000                  1000
SF_RACETIME_TO_SEC              7            1               1                   800                   800
SF_RACETIME_TO_SEC             20            3               1                   800                   800
SF_RACETIME_TO_SEC              5            1               1                   400                   400
SF_RACETIME_TO_SEC              8            1               1                   400                   400
SF_RACETIME_TO_SEC             10            1               1                   400                   400
SF_RACETIME_TO_SEC             11            1               1                   400                   400
SF_RACETIME_TO_SEC             12            1               1                   400                   400
SF_RACETIME_TO_SEC             16            3               1                   400                   400
SP_SOME_STAT                   15            3               1                   300                   300
SF_RACETIME_TO_SEC              9            1               1                   300                   300
```

Наибольшее время вносит строка 18 в процедуре SP_SOME_STAT - это собственно курсор верхнего уровня, но этот курсор открывается один раз. Здесь важно отметить, что на общее время извлечения всех записей из курсора влияют операторы выполняемые внутри операторного блока по обработки каждой записи курсора, то есть

```
  FOR
    SELECT
...
  DO
  BEGIN
    -- всё что выполняется здесь влияет на время извлечения всех записей из курсора
    ...
  END
```

Давайте посмотрим, что внутри этого блока вносит наиболее существенный вклад. Это строчка номер 40 процедуры SP_SOME_STAT, которая вызывается 240 раз. Вот содержимое оператора который вызывается:

```
    SELECT
      FARM.NAME
    FROM
      (
        SELECT
          R.CODE_FARM
        FROM REGISTRATION R
        WHERE R.CODE_HORSE = :CODE_HORSE
          AND R.CODE_REGTYPE = 6
          AND R.BYDATE <= :BYDATE
        ORDER BY R.BYDATE DESC
        FETCH FIRST ROW ONLY
      ) OWN
      JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
    INTO OWNERNAME;
```

А теперь давайте глянем на план процедуры SP_SOME_STAT:

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
JOIN PLG$PROF_STATEMENTS S ON S.STATEMENT_ID = CS.MON$COMPILED_STATEMENT_ID
WHERE CS.MON$OBJECT_NAME = 'SP_SOME_STAT'
  AND S.PROFILE_ID = 6;
```

```
================================================================================
MON$EXPLAINED_PLAN:

Select Expression (line 40, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Refetch
                    -> Sort (record length: 28, key length: 8)
                        -> Filter
                            -> Table "REGISTRATION" as "OWN R" Access By ID
                                -> Bitmap
                                    -> Index "REGISTRATION_IDX_HORSE_REGTYPE" Range Scan (full match)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
Select Expression (line 18, column 3)
    -> Nested Loop Join (inner)
        -> Filter
            -> Table "TRIAL" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "IDX_TRIAL_BYYEAR" Range Scan (lower bound: 1/1, upper bound: 1/1)
                    -> Bitmap
                        -> Index "FK_TRIAL_TRIALTYPE" Range Scan (full match)
        -> Filter
            -> Table "TRIAL_LINE" as "TL" Access By ID
                -> Bitmap
                    -> Index "FK_TRIAL_LINE_TRIAL" Range Scan (full match)
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
================================================================================
```

Таким образом план нашего запроса:

```
Select Expression (line 40, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Refetch
                    -> Sort (record length: 28, key length: 8)
                        -> Filter
                            -> Table "REGISTRATION" as "OWN R" Access By ID
                                -> Bitmap
                                    -> Index "REGISTRATION_IDX_HORSE_REGTYPE" Range Scan (full match)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
```

Как видим план не самый эффективный. Используется индекс для фильтрации данных, а

потом сортировка полученных ключей и Refetch. Посмотрим на индекс REGISTRATION_IDX_HORSE_REGTYPE:

```
SQL>SHOW INDEX REGISTRATION_IDX_HORSE_REGTYPE;
REGISTRATION_IDX_HORSE_REGTYPE INDEX ON REGISTRATION(CODE_HORSE, CODE_REGTYPE)
```

В индекс включены только поля CODE_HORSE и CODE_REGTYPE, поэтому не может быть использована навигация по индексу для определения последней записи на дату. Попробуем создать другой композитный индекс:

```
CREATE DESCENDING INDEX IDX_REG_HORSE_OWNER ON REGISTRATION(CODE_HORSE, CODE_REGTYPE,
BYDATE);
```

Снова выполним наш запрос:

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

```
                 COUNT
=====================
                   240

Current memory = 554429808
Delta memory = 288
Max memory = 554462400
Elapsed time = 0.125 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124165
```

На всякий случай проверим, что план процедуры изменился.

```sql
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
WHERE CS.MON$OBJECT_NAME = 'SP_SOME_STAT'
ORDER BY CS.MON$COMPILED_STATEMENT_ID DESC
FETCH FIRST ROW ONLY;
```

```
================================================================================
MON$EXPLAINED_PLAN:

Select Expression (line 38, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Filter
                    -> Table "REGISTRATION" as "OWN R" Access By ID
                        -> Index "IDX_REG_HORSE_OWNER" Range Scan (lower bound: 3/3, upper bound: 2/3)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
Select Expression (line 16, column 3)
    -> Nested Loop Join (inner)
        -> Filter
            -> Table "TRIAL" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "IDX_TRIAL_BYYEAR" Range Scan (lower bound: 1/1, upper bound: 1/1)
                    -> Bitmap
                        -> Index "FK_TRIAL_TRIALTYPE" Range Scan (full match)
        -> Filter
            -> Table "TRIAL_LINE" as "TL" Access By ID
                -> Bitmap
                    -> Index "FK_TRIAL_LINE_TRIAL" Range Scan (full match)
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
================================================================================
```

Да, план стал лучше. Теперь используется навигация по индексу `IDX_REG_HORSE_OWNER` с Range Scan. Если сравнивать время выполнения, то получим 0.134 секунд vs 0.125 секунд и 124992 vs 124165 фетчей. Улучшения очень незначительное. В принципе относительно изначального варианта наша процедура уже стала быстрее в 19 раз, поэтому оптимизацию можно закончить.

Chapter 12. Заключение

На этом обзор новинок Firebird 5.0 окончен. Разработчики Firebird проделали огромную работу, за что им огромная благодарность.

Тестируйте Release Candidate и готовьтесь к миграции на Firebird 5.0.