Методы доступа используемые в Firebird

Симонов Денис, Еманов Дмитрий

Версия 2.0 от 17.03.2025

Содержание

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

Данная статья описывает методы доступа существующие в **Firebird 5.0**. Она основана на оригинальной статье Firebird: методы доступа к данным Дмитрия Еманова. Предыдущая статья была написана очень давно и актуальна для Firebird 2.0 и частично для более новых версий Firebird.

По сравнению с оригинальной статьёй изменено следующее:

- Добавлены описания методов доступа, которые появились после Firebird 2.0 — в 2.1, 2.5, 3.0, 4.0 и Firebird 5.0.
- Схематичное дерево выполнения запросов заменено на реальные Explain планы (появились в Firebird 3.0).
- Поскольку в оригинальной статье приводились примеры расчёта кардинальности и стоимости, то ими дополнены Explain планы. Значения кардинальностей получены с помощью процедуры `RDB$SQL.EXPLAIN` из Firebird 6.0. Стоимости посчитаны по формулам, которые либо были в оригинальной статье, либо получены мной при анализе исходных текстов Firebird.
- Для Hash Join добавлена формула расчёта его стоимости.
- Добавлено описание как фильтры режут кардинальность (Filter, group by, distinct).

Глава 1. Терминология

Путь доступа — это набор операций над данными, выполняемых сервером для получения результата заданной выборки. Explain план представляет собой дерево с корнем, представляющим собой конечный результат. Каждый узел этого дерева называется **методом доступа** или **источником данных**. Объектами операций в методах доступа являются **потоки данных**. Каждый метод доступа либо формирует поток данных, либо трансформирует его по определенным правилам. Листовые узлы дерева называются **первичными методами доступа**. Их единственная задача — формирование потоков данных.

С точки зрения видов выполняемых операций существует три класса источников данных:

- первичный метод доступа — выполняет чтение из таблицы или хранимой процедуры, формирует исходный поток данных;
- фильтр — трансформирует один входной поток данных в один выходной поток;
- слияние — преобразует два или более входных потоков данных в один выходной поток.

Источники данных могут быть конвейерными и буферизованными. Конвейерный источник данных выдает записи в процессе чтения своих входных потоков, в то время как буферизованный источник сначала должен прочитать все записи из своих входных потоков и только потом сможет выдать первую запись на свой выход.

С точки зрения оценки производительности, каждый метод доступа имеет два обязательных атрибута — кардинальность (cardinality) и стоимость (cost). Первый отражает, сколько записей будет выбрано из источника данных. Второй оценивает стоимость выполнения метода доступа. Величина стоимости напрямую зависит от кардинальности и механизма выборки или трансформации потока данных. В текущих версиях сервера стоимость определяется количеством логических чтений (страничных фетчей, page fetches), необходимых для возврата всех записей методом доступа. Таким образом, более "высокие" методы всегда имеют большую стоимость, чем низкоуровневые. CPU-нагрузка в вычислительных методах доступа приводится к примерно эквивалентному (по заданным коэффициентам) количеству логических чтений.

Глава 2. Первичные методы доступа

Группа этих методов доступа выполняет создание потока данных на основе низкоуровневых источников, таких как таблицы (внутренние и внешние) и процедуры. Далее мы рассмотрим каждый из первичных источников данных отдельно.

2.1. Чтение таблицы

Является самым распространенным первичным методом доступа. Стоит отметить, что здесь мы ведем речь только о “внутренних” таблицах, то есть тех, данные которых расположены в файлах базы данных. Доступ к внешним таблицам (external tables), а также выборка из хранимых процедур (stored procedures) осуществляются другими способами и будут описаны отдельно.

2.1.1. Полное сканирование (Full table scan)

Этот метод также известен как естественный или последовательный перебор (Full Scan, Natural Scan, Sequential Scan).

В данном методе доступа сервер осуществляет последовательное чтение всех страниц, выделенных для данной таблицы, в порядке их физического расположения на диске. Очевидно, что этот метод обеспечивает наиболее высокую производительность с точки зрения пропускной способности, то есть количества отфетченных записей в единицу времени. Также достаточно очевидно, что прочитаны будут все записи данной таблицы независимо от того, нужны они нам или нет.

Так как ожидаемый объем данных довольно велик, то в процессе чтения страниц таблицы с диска существует проблема вытеснения читаемыми страницами других, потенциально нужных конкурирующим сессиям. Для этого логика работы страничного кеша меняется — текущая страница скана расположена в позиции MRU (most recently used) в течение чтения всех записей с данной страницы. Как только на странице нет больше данных и надо фетчить следующую, то текущая страница освобождается с признаком LRU (least recently used), уходя в “хвост” очереди и будучи таким образом первым кандидатом на удаление из кеша.

Записи читаются из таблицы по одной, сразу выдаваясь на выход. Следует отметить, что пре-фетча записей (хотя бы в пределах страницы) с их буферизацией в сервере нет. То есть полная выборка из таблицы со 100К записями, занимающими 1000 страниц, приведет к 100К страничным фетчам (page fetch), а не к 1000, как можно было бы предположить. Также нет и пакетного чтения (multi-block reads), при котором соседние страницы можно было бы выделить в группы и читать с диска “пачками” по несколько штук, уменьшая этим количество операций физического ввода/вывода.

Обе этих возможности планируются к реализации в следующих версиях сервера.

Начиная с Firebird 3.0 страницы данных (Data Pages или DP) для таблиц (содержащих более 8 DP) выделяются группами, называемыми **экстентами**. Один экстент состоит из 8 страниц. Это позволяет движку эффективнее использовать префетч ОС, поскольку страницы относящиеся к одной таблице расположены рядом, а следовательно полное сканирование таблиц с большей вероятностью будет читать нужные страницы из кеша ОС.

Оптимизатор при выборе данного метода доступа использует стратегию на основе правил (rule based) — полное сканирование осуществляется только в случае отсутствия индексов, применимых к предикату запроса. Стоимость этого метода доступа равна количеству записей в таблице (оценивается приближенно по количеству страниц таблицы, размеру записи и среднему коэффициенту сжатия записей на страницах данных). Теоретически, это неправильно и при выборе метода доступа надо всегда основываться на стоимости, но на практике это оказывается ненужным в подавляющем большинстве случаев. Причины этого будут разъяснены ниже при описании индексного доступа.

Здесь и далее при упоминании поведения оптимизатора будут приводиться: пример SELECT-запроса, план его выполнения в Legacy и Explain форме. В Legacy плане выполнения полное сканирование таблицы обозначается словом “NATURAL”. В Explain форме — `Table "<имя таблицы>" Full Scan`. В настоящее время указанное в квадратных скобках (кардинальность и стоимость) не выводятся в explain плане, оно приведено в качестве расчётного примера.

Пример 1. Полное сканирование таблицы RDB$RELATIONS

```
SELECT *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=120.0, cost=120.0]
Select Expression
    [cardinality=120.0, cost=120.0]
    -> Table "RDB$RELATIONS" Full Scan
```

В статистике выполнения запроса записи, прочитанные полным сканированием, отражаются как неиндексированные чтения (non indexed reads).

При использовании псевдонимов таблиц план также будет отображать псевдонимы.

Пример 2. Полное сканирование таблицы RDB$RELATIONS с псевдонимом

```
SELECT *
FROM RDB$RELATIONS R
```

```
PLAN (R NATURAL)
```

```
[cardinality=120.0, cost=120.0]
Select Expression
    [cardinality=120.0, cost=120.0]
    -> Table "RDB$RELATIONS" as "R" Full Scan
```

2.1.2. Доступ через идентификатор записи

В этом случае подсистема выполнения получает идентификатор (физический номер) записи, которую надо прочитать. Этот номер записи может быть получен из разных источников, каждый из которых будет описан далее.

Несколько упрощенно, физический номер записи содержит информацию о странице, на которой расположена данная запись, и о смещении внутри этой страницы. Таким образом этой информации достаточно, чтобы выполнить фетч необходимой страницы и найти на ней искомую запись.

Этот метод доступа является низкоуровневым и используется только как реализация выборки на основе битовой карты (как индексного скана, так и доступа через `RDB$DB_KEY`) и индексной навигации. Более подробно об этих методах доступа будет рассказано далее.

Стоимость данного вида доступа всегда равна единице. Чтения записей отражаются в статистике как индексированные.

2.1.3. Позиционированный доступ

Здесь идет речь о позиционированных командах `UPDATE` и `DELETE` (синтаксис `WHERE CURRENT OF`). Существует ошибочное мнение, что данный метод доступа является синтаксическим аналогом выборки с помощью `RDB$DB_KEY`, однако это не так. Позиционированный доступ работает только для активного курсора, т.е. для уже отфетченной записи (с помощью команд `FOR SELECT` или `FETCH`). В противном случае будет выдана ошибка `isc_no_cur_rec` (“no current record for fetch operation”). Таким образом, это просто способ ссылки на активную запись курсора, не требующий операций чтения вообще. В то время как выборка через `RDB$DB_KEY` задействует доступ через идентификатор записи и, следовательно, всегда приводит к фетчу одной страницы.

2.2. Индексный доступ

Идея индексного доступа проста — помимо таблицы с данными у нас есть еще структура,

содержащая пары "ключ - номер записи" в виде, позволяющем выполнять быстрый поиск по значению ключа. В Firebird индекс представляет собой страничное B+ дерево с префиксной компрессией ключей.

Индексы могут быть простыми (односегментными) и составными (многосегментными или композитными). Следует отметить, что совокупность полей композитного индекса представляет собой единый ключ. Поиск в индексе может осуществляться как по ключу целиком, так и по его подстроке (подключу). Очевидно, что поиск по подключу допустим только для начальной части ключа (например, `STARTING WITH` или использование не всех сегментов композита). Если поиск осуществляется по всем сегментам индекса, то это называется полным совпадением (full match) ключа, иначе это частичное совпадение (partial match) ключа. Отсюда для композитного индекса по полям (A, B, C) следует, что:

- он может быть использовать для предикатов (`A = 0`) или (`A = 0 and B = 0`) или (`A = 0 and B = 0 and C = 0`), но не может быть использован для предикатов (`B = 0`) или (`C = 0`) или (`B = 0 and C = 0`);
- предикат (`A = 0 and B > 0 and C = 0`) приведет к частичному совпадению по двум сегментам, а предикат (`A > 0 and B = 0`) – к частичному совпадению всего по одному сегменту.

Очевидно, что индексный доступ требует от нас чтения как страниц индекса для поиска, так и страниц данных для чтения записей. Другие СУБД в некоторых случаях способны ограничиваться только страницами индекса – например, если все поля выборки входят в индекс. Но такая схема невозможна в Firebird в связи с его архитектурой — ключ индекса содержит только номер записи без информации о ее версии — следовательно, сервер в любом случае должен прочитать саму запись для определения, видна ли хоть одна из версий с данным ключом для текущей транзакции. Часто возникает вопрос: а если включить информацию о версиях (то есть номера транзакций) в ключ индекса? Ведь тогда можно будет реализовать чистое индексное покрытие (index only scan). Но тут есть два проблематичных момента. Во-первых, увеличится длина ключа, следовательно индекс будет занимать больше страниц, что приведет к большему объему ввода/вывода на ту же самую операцию сканирования. Во-вторых, каждое изменение записи приведет к модификации ключа индекса, даже если изменялись неключевые поля. В то время как сейчас индекс модифицируется только в случае изменения ключевых полей записи. Первая из проблем приведёт к существенному ухудшению производительности сканирования индексов с короткими ключами (с типами INT, BIGINT и других). Вторая — как минимум утраивает количество модифицированных страниц на каждую команду изменения данных по сравнению с текущей ситуацией. До сих пор разработчики сервера считают это слишком большой ценой за реализацию чистого индексного покрытия.

Существует альтернативный вариант чисто индексного сканирования реализованный в Postgres. Он эффективно работает для таблиц в которых изменяется только небольшая часть записей. Для этого помимо самого индекса существует дополнительная дисковая структура называемая карта видимости. Процедура сканирования индекса, найдя потенциально подходящую запись в индексе, проверяет бит в карте видимости для соответствующей страницы данных. Если он установлен, значит эта строка видна, и данные могут быть возвращены сразу. В противном случае придётся посетить запись строки и проверить, видима ли она, так что никакого выигрыша по сравнению с обычным сканированием индекса не будет.

Этот вариант возможно реализовать и в Firebird. Начиная с Firebird 3.0 для страниц DP (Data Page) существует так называемый swept flag. Он устанавливается в 1, если sweep или сборщик мусора посетил страницу и не обнаружил или вычистил все не первичные версии записей. Этот же флаг присутствует и на страницах указателей PP (Pointer Page), которые можно использовать как аналог вышеописанной карты видимости.

По сравнению с другими СУБД у индексов в Firebird есть еще одна особенность — сканирование индекса всегда однонаправленное, от меньших ключей к большим. Часто из-за этого индекс называют однонаправленным и говорят, что в его узле есть указатели только на следующий узел и нет указателя на предыдущий. На самом деле, проблема не в этом. Все дисковые структуры сервера Firebird спроектированы как свободные от взаимных блокировок (deadlock free), причем гарантируется минимальная возможная гранулярность блокировок. Помимо этого, действует также правило "аккуратной" записи (careful write) страниц, служащее мгновенному восстановлению после сбоя. Проблема в двунаправленных индексах заключается в том, что они нарушают это правило при расщеплении страницы. На текущий момент неизвестен способ безблокировочной работы с двунаправленными указателями в случае, если одна из страниц должна быть записана строго перед другой.

На самом деле, варианты обхода этой проблемы существуют и они сейчас обсуждаются разработчиками.

Данная особенность приводит к невозможности использования ASC-индекса для DESC-сортировки или вычисления `MAX` и наоборот, невозможности использования DESC-индекса для ASC-сортировки или вычисления `MIN`. Разумеется, сканированию индекса с целью поиска однонаправленность никак не мешает.

2.2.1. Селективность индекса

Основным параметром, влияющим на оптимизацию индексного доступа, является **селективность** индекса. Это величина, обратная количеству уникальных значений ключа. В расчетах кардинальности и стоимости предполагается равномерное распределение значений ключа в индексе.

Селективность индекса можно узнать используя следующий запрос

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = '<index_name>'
```

Пример 3. Получение сохранённой статистики для индекса CUSTNAMEX

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = 'CUSTNAMEX'
```

```
        RDB$STATISTICS
======================
    0.06666667014360428
```

Для композитных индексов, помимо селективности индекса Firebird также хранит селективность совокупности его сегментов, начиная от первого до данного. Селективность совокупности сегментов можно узнать используя запрос к таблице RDB$INDEX_SEGMENTS.

Пример 4. Получение сохранённой статистики для каждого сегмента индекса NAMEX

```
SELECT RDB$FIELD_NAME, RDB$FIELD_POSITION, RDB$STATISTICS
FROM RDB$INDEX_SEGMENTS
WHERE RDB$INDEX_NAME = 'NAMEX';
```

```
RDB$FIELD_NAME     RDB$FIELD_POSITION          RDB$STATISTICS
================== ================== ======================
LAST_NAME                           0     0.02500000037252903
FIRST_NAME                          1     0.02380952425301075
```

Пример 5. Получение сохранённой статистики для индекса NAMEX

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = 'NAMEX'
```

```
        RDB$STATISTICS
======================
    0.02380952425301075
```

Селективность индекса рассчитывается при его построении (CREATE INDEX, ALTER INDEX … ACTIVE) и в последствии может устареть. Для обновления статистики по индексу

используется следующий SQL запрос

```
SET STATISTICS INDEX <index_name>;
```

Пример 6. Сбор статистики для индекса NAMEX

```
SET STATISTICS INDEX NAMEX;
```

В настоящее время Firebird не обновляет статистику индексов автоматически. Кроме того, хранимая статистика содержит очень мало сведений для оптимизации доступа с использованием индекса. Фактически хранится только селективность индекса и его сегментов. Но не хранятся такие важные характеристики индекса как, избирательность значения NULL, глубина индекса, средняя длинна ключа индекса, гистограммы распределения значений, степень кластеризации ключей индекса.

Хранимая статистика будет расширена в следующих версиях Firebird. Кроме того, планируются некоторые возможности по её автоматическому обновлению.

Помимо обычных и композитных индексов в Firebird можно создавать индексы по выражению. В этом случае вместо значений полей таблицы в качестве ключей индекса используется значений выражения, возвращающего скалярную величину. Для использования таких индексов оптимизатором в условии фильтрации запроса должно быть использовано точно такое же выражение какое было указано при создании индекса. Индексы по выражению не могут быть композитными, но выражение может содержать несколько полей таблицы. Селективность такого индекса считается точно также как и для обычных индексов.

2.2.2. Частичные индексы

Начиная с 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.

Оптимизатор может использовать частичный индекс 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`.

Последний пункт продемонстрирую на примере. Предположим вы создали следующий частичный индекс:

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

Теперь оптимизатор может использовать данный индекс не только для предиката `IS NOT NULL`, но и для предикатов `=`, `<`, `>`, `BETWEEN` и других, если они не возвращают `TRUE` при сравнении с `NULL`. Для предикатов `IS NULL` и `IS NOT DISTINCT FROM` индекс не может быть задействован.

```
-- частичный индекс будет использован
SELECT *
FROM HORSE
WHERE DEATHDATE IS NOT NULL

-- частичный индекс будет использован
SELECT *
FROM HORSE
WHERE DEATHDATE = ?

-- частичный индекс будет использован
SELECT *
FROM HORSE
WHERE DEATHDATE BETWEEN ? AND ?

-- частичный индекс не может быть использован
SELECT *
FROM HORSE
WHERE DEATHDATE IS NOT DISTINCT FROM ?
```

Если для одного и того же набора полей существует обычный индекс и частичный индекс, то оптимизатор в большинстве случаев выберет обычный индекс, даже если условие `WHERE` включает тоже самое выражение, что определено в частичном индексе. Причина такого поведения состоит в том, что селективность обычного индекса известна “точно” (посчитана как величина, обратная количеству уникальных ключей). Но в частичный индекс попадает меньше ключей из-за наличия дополнительного условия фильтрации, поэтому хранимая селективность индекса будет хуже. Расчётная селективность частичного индекса оценивается как хранимая селективность умноженная долю записей попавших в индекс. Эта доля в настоящий момент не хранится в статистике (“точно” не известна), а оценивается исходя из селективности выражения фильтрации, указанном при создании индекса (см. Проверка предикатов). При этом реальная селективность может быть меньше и поэтому обычный индекс (с точно вычисленным значением селективности) может выиграть.

Это может быть изменено в следующих версиях сервера.

2.2.3. Битовые карты

Основная стандартная проблема индексного доступа это случайный ввод/вывод по отношению к страницам данных. Ведь действительно порядок ключей в индексе весьма нечасто совпадает с порядком соответствующих записей в таблице. Таким образом, выборка значительного числа записей через индекс наверняка приведет к многократному фетчу каждой соответствующей страницы данных. Эта проблема решается в других СУБД с помощью кластерных индексов (термин MSSQL) или таблиц, упорядоченных по индексу (термин Oracle), в которых данные расположены прямо в индексе в порядке возрастания кластерного ключа.

В Firebird эта проблема решена другим путем. Сканирование индекса не является конвейерной операцией, а выполняется для всего диапазона поиска, включая полученные номера записей в специальную битовую карту. Эта карта представляет собой разреженный битовый массив, где каждый бит соответствует конкретной записи и наличие единицы в нем является указанием для выборки данной записи. Особенность данного решения состоит в том, что битовая карта по определению отсортирована по номерам записей. После окончания скана данный массив служит основой для последовательного доступа через идентификатор записи. Достоинство очевидно — чтение из таблицы идет в физическим порядке расположения страниц, как и при полном сканировании, то есть каждая страница будет прочитана не более одного раза. Таким образом простота реализации индекса тут сочетается с максимально эффективным доступом к записям. Цена — некоторое увеличение времени отклика на запросах со стратегией FIRST ROWS, когда нужно очень быстро получить первые записи.

Над битовыми картами допустимы операции пересечения (bit and) и объединения (bit or), таким образом сервер может использовать несколько индексов для одной таблицы.

2.2.4. Сканирование диапазона

Поиск под индексу осуществляется с использованием верхней и нижней границы. То есть, если указана нижняя граница сканирования, то сначала находится соответствующий ей ключ и только потом начинается последовательный перебор ключей с занесением номеров записей в битовую карту. Если указана верхняя граница, то каждый ключ сравнивается с ней и при ее достижении сканирование прекращается. Такой механизм называется сканированием диапазона (range scan). В случае, когда ключ нижней границы равен ключу верхней, говорят о сканировании на равенство (equality scan). Если сканирование на равенство выполняется для уникального индекса, то это называется уникальным сканированием (unique scan). Данный вид сканирования имеет особое значение, так как может вернуть не более одной записи и, следовательно, по определению является самым дешевым. Если у нас не задана ни одна из границ, то мы имеем дело с полным сканированием (full scan). Этот метод применяется исключительно для индексной навигации, описанной ниже.

При возможности сервер пропускает NULL-значения при сканировании диапазона, если это допустимо. В большинстве случаев это приводит к увеличению производительности за счет меньшего размера битовой карты и, соответственно, меньшего количества индексированных чтений. Данная опция не используется только для индексированных предикатов вида `IS NULL` и `IS NOT DISTINCT FROM`, которые учитывают NULL-значения.

При выборе индексов для сканирования оптимизатор использует стратегию на основе стоимости (cost based). Стоимость сканирования диапазона оценивается на основании селективности индекса, количества записей в таблице, среднего количества ключей на индексной странице и высоты B+ дерева.

Высота B+ дерева в настоящее время не хранится в статистике, а потому задана в коде в виде константы равной 3.

В Legacy плане выполнения сканирование диапазона индекса обозначается словом “INDEX”, за которым в скобках следуют наименования всех индексов, образующих итоговую битовую карту.

В Explain плане отображение сканирования диапазона несколько более сложное. В общем виде оно выглядит следующим образом

```
Table "<table_name>" Access By ID
    -> Bitmap
        -> Index "<index_name>" <scan_type>
```

Где `index_name` — имя используемого индекса, `scan_type` — тип сканирования, `table_name` — имя таблицы.

Unique scan

Уникальное сканирование может быть использовано только для уникальных индексов при сканировании по всем сегментам (полное совпадение) и с использованием только предикатов равенства "=". Уникальные индексы автоматически создаются для ограничений первичного ключа или ограничения уникальности, но могут быть созданы и самостоятельно. Данный вид сканирования имеет особое значение, так как может вернуть не более одной записи, а следовательно его кардинальность всегда равна 1. Стоимость данного рассчитывается по формуле

```
cost = indexDepth + 1
```

Пример 7. Index Unique Scan

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME = ?
```

```
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

В Explain плане уникальное сканирование отображается следующим образом:

```
[cardinality=1.0, cost=4.000]
Select Expression
    [cardinality=1.0, cost=4.000]
    -> Filter
        [cardinality=1.0, cost=4.000]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" Unique Scan
```

Сканирование на равенство

Сканирование на равенство используется для предикатов `IS NULL`, `IS NOT DISTINCT FROM` и равенства (`=`). Если при сканировании на равенство задействованы все сегменты индекса, то в Explain плане для такого сканирования будет указано (`full match`), в противном случае будет указано (`partial match`) и количество задействованных сегментов.

В этом случае кардинальность высчитывается по формуле:

```
cardinality = table_cardinality * index_selectivity
```

Здесь `index_selectivity` — это селективность совокупности сегментов индекса задействованных при сканировании на равенство.

Стоимость сканирования индекса на равенство рассчитывается немного сложнее

```
cost = indexDepth + MAX(avgKeyLength * table_cardinality * index_selectivity / (page_size - BTR_SIZE), 1)
```

Где `avgKeyLength` — средняя длина ключа, `page_size` — размер страницы, `BTR_SIZE` — размер заголовка индексной страницы, в настоящее время составляет 39 байт.

В настоящее время средняя длина ключа не хранится в статистике индекса, а рассчитывается по средней степени сжатия и длине ключа по формуле:

```
avgKeyLength = 2 + length * factor
```

Здесь `length` — длина ключа индекса, `factor` — степень сжатия.

Степень сжатия считается равной 0.5 для простых индексов, и 0.7 для композитных.

Пример 8. Index Range Scan (full match)

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID = ?
```

```
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_1))
```

```
[cardinality=1.02, cost=4.020]
Select Expression
    [cardinality=1.02, cost=4.020]
    -> Filter
        [cardinality=1.02, cost=4.020]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_1" Range Scan (full match)
```

Пример полного совпадения для композитного индекса:

Пример 9. Index Range Scan (full match) для композитного индекса

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME = ? AND LAST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.0, cost=4.000]
Select Expression
    [cardinality=1.0, cost=4.000]
    -> Filter
        [cardinality=1.0, cost=4.000]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (full match)
```

Теперь посмотрим на пример, в котором для композитного индекса используется только первый из двух сегментов.

Пример 10. Index Range scan (partial match)

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.05, cost=4.050]
Select Expression
    [cardinality=1.05, cost=4.050]
    -> Filter
        [cardinality=1.05, cost=4.050]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (partial match: 1/2)
```

Range Scan с границами

Если верхняя и нижняя граница сканирования не совпадают или используется только одна из них, то в Explain плане будет отображено какие границы сканирования используются: "lower bound" — нижняя граница, "upper bound" — верхняя граница. Для каждой границы указывается сколько сегментов индекса задействовано. Например (lower bound: 1/2) обозначает, что для нижней границы будет задействован один сегмент из двух.

Стоимость и кардинальность сканирования диапазона рассчитывается точно также как и для сканирования на равенство, но при этом вместо селективности индекса используются константы для селективности предикатов.

Таблица 1. Селективность предикатов индексного сканирования

Предикаты	Селективность
<, <=, >, >=	0.05
BETWEEN	0.0025
STARTING WITH	0.01
IN	indexSelectivity * N

Пример 11. Index Range Scan с нижней границей

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO > 0
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=6.09, cost=9.090]
Select Expression
    [cardinality=6.09, cost=9.090]
    -> Filter
        [cardinality=6.09, cost=9.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (lower bound: 1/1)
```

Пример 12. Index Range Scan с верхней границей

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO < 0
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=6.09, cost=9.090]
Select Expression
    [cardinality=6.09, cost=9.090]
    -> Filter
        [cardinality=6.09, cost=9.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (upper bound: 1/1)
```

Пример 13. Index Range Scan с нижней и верхней границами

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO BETWEEN ? AND ?
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=4.295, cost=7.295]
Select Expression
    [cardinality=4.295, cost=7.295]
    -> Filter
        [cardinality=4.295, cost=7.295]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (lower bound: 1/1, upper bound: 1/1)
```

Теперь посмотрим примеры для композитных индексов, в которых для границ используется только часть сегментов.

Пример 14. Range Scan композитного индекса (нижняя граница использует первый из двух сегментов)

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=3.09, cost=6.090]
Select Expression
    [cardinality=3.09, cost=6.090]
    -> Filter
        [cardinality=3.09, cost=6.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 1/2)
```

Пример 15. Range Scan композитного индекса. Нижняя граница использует оба сегмента, а верхняя только первый

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME = ? AND FIRST_NAME > ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.03, cost=4.030]
Select Expression
    [cardinality=1.03, cost=4.030]
    -> Filter
        [cardinality=1.03, cost=4.030]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 2/2, upper bound: 1/2)
```

Попробуем наоборот:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > ? AND FIRST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.0, cost=6.097]
Select Expression
    [cardinality=1.0, cost=6.097]
    -> Filter
        [cardinality=3.097, cost=6.097]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 1/2)
```

Как видите используется только нижняя граница первого сегмента, а второй сегмент не задействован вовсе.

Теперь попробуем задействовать только второй сегмент.

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME = ?
```

```
PLAN (EMPLOYEE NATURAL)
```

```
[cardinality=4.2, cost=42.000]
Select Expression
    [cardinality=4.2, cost=42.000]
    -> Filter
        [cardinality=42.0, cost=42.000]
        -> Table "EMPLOYEE" Full Scan
```

В данном случае индексное сканирование задействовать не удалось.

В некоторых СУБД (в частности в Oracle) существует так называемый Index Skip Scan, при котором весь ключ композитного индекса не сравнивается, а сравниваются только задействованная часть ключа, в этом случае последний пример мог бы использовать индексное сканирование. В настоящее время данный метод доступа отсутствует в Firebird.

List scan

Сканирование по списку (List scan) доступно начиная с Firebird 5.0. Оно применимо только для предиката `IN` со списком значений. При этом формируется одна общая битовая карта для всего списка значений. Поиск может выполняться вертикальным сканированием (от корня для каждого ключа) или горизонтальным сканированием диапазона между min/max ключами. Как именно будет выполняться сканирование решает оптимизатор в зависимости от того что дешевле с точки зрения стоимости.

До Firebird 5.0 такой предикат преобразовывался в набор условий по равенству объединённых через предикат `OR`.

То есть,

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

преобразовывался в

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

В Firebird 5.0 это работает иначе. 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. Если для предиката применим индекс, то используется сканирование индекса по списку (List scan).

Кардинальность данного метода доступа высчитывается по формуле:

```
cardinality = table_cardinality * MAX(index_selectivity * N, 1)
```

Где `N` - количество элементов в списке `IN`.

Стоимость сканирования индекса по списку рассчитывается так:

```
cost = indexDepth + MAX(avgKeyLength * table_cardinality * index_selectivity * N, 1)
```

Пример 16. List scan

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME IN ('RDB$RELATIONS', 'RDB$PROCEDURES', 'RDB$FUNCTIONS')
```

```
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

```
[cardinality=3.2, cost=6.200]
Select Expression
    [cardinality=3.2, cost=6.200]
    -> Filter
        [cardinality=3.2, cost=6.200]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" List Scan (full match)
```

2.2.5. Пересечение и объединение битовых масок

Как говорилось выше, над битовыми картами допустимы операции пересечения (bit and) и объединения (bit or), таким образом сервер может использовать несколько индексов для одной таблицы. При пересечении битовых карт результирующая кардинальность не увеличивается. Например, для выражения `F = 0 and ? = 0` его вторая часть не индексируема и, следовательно, проверяется уже после индексной выборки, не оказывая этим влияния на конечный результат. Но объединение битовых карт приводит к увеличению результирующей кардинальности, поэтому объединяться могут только части полностью индексированного предиката. То есть при выносе второй части выражения `F = 0 or ? = 0` на уровень выше может оказаться, что надо было сканировать все записи. Поэтому индекс для поля F не будет использован в таком выражении.

Селективность пересечения двух битовых карт вычисляется по формуле:

```
(bestSel + (worstSel - bestSel) / (1 - bestSel) * bestSel) / 2
```

Селективность объединения двух битовых карт вычисляется по формуле:

```
bestSel + worstSel
```

Так как стоимость доступа через идентификатор записи равна единице, то итоговая стоимость доступа через битовую карту будет равна суммарной стоимости индексного поиска (для всех индексов, формирующих битовую карту) плюс результирующей кардинальности битовой карты.

Давайте посмотрим примеры с использованием нескольких индексов.

Пример 17. Пересечение битовых масок

```
SELECT *
FROM RDB$INDICES
WHERE RDB$RELATION_NAME = ? AND RDB$FOREIGN_KEY = ?
```

```
PLAN (RDB$INDICES INDEX (RDB$INDEX_31, RDB$INDEX_41))
```

```
[cardinality=1.0, cost=7.000]
Select Expression
    [cardinality=1.0, cost=7.000]
    -> Filter
        [cardinality=1.0, cost=7.000]
        -> Table "RDB$INDICES" Access By ID
            -> Bitmap And
                -> Bitmap
                    -> Index "RDB$INDEX_31" Range Scan (full match)
                -> Bitmap
                    -> Index "RDB$INDEX_41" Range Scan (full match)
```

Пример 18. Объединение битовых масок

```
SELECT *
FROM RDB$INDICES
WHERE RDB$RELATION_NAME = ? OR RDB$FOREIGN_KEY = ?
```

```
PLAN (RDB$INDICES INDEX (RDB$INDEX_31, RDB$INDEX_41))
```

```
[cardinality=10.85, cost=16.850]
Select Expression
    [cardinality=10.85, cost=16.850]
    -> Filter
        [cardinality=10.85, cost=16.850]
        -> Table "RDB$INDICES" Access By ID
            -> Bitmap Or
                -> Bitmap
                    -> Index "RDB$INDEX_31" Range Scan (full match)
                -> Bitmap
                    -> Index "RDB$INDEX_41" Range Scan (full match)
```

2.2.6. Навигация по индексу

Индексная навигация есть ни что иное, как последовательное сканирование ключей индекса. А так как для каждой записи серверу надо выполнить фетч записи и проверить ее видимость для нашей транзакции, то данная операция выходит достаточно дорогой. Именно поэтому данный метод доступа не применяется для обычных выборок (в отличие от Oracle, например), а используется только в случаях, когда это оправдано.

Сортировка с помощью нескольких индексов не поддерживается.

На сегодняшний день, существует два таких случая. Первый — это вычисление агрегатных функций `MIN`/`MAX`. Очевидно, что для вычисления `MIN` достаточно взять первый ключ в ASC-индексе, а для вычисления `MAX` — первый ключ в DESC-индексе. Если после фетча записи оказывается, что она нам не видна, то мы берем следующий ключ, и так далее. Второй — это сортировка или группировка записей. Об этом говорят предложения `ORDER BY` или `GROUP BY` в запросе пользователя. В данной ситуации мы просто идем по индексу, выбирая записи по мере сканирования. В обоих случаях фетч записей выполняется на основе доступа через ее идентификатор.

Об однонаправленности индексной навигации написано в Индексный доступ.

Есть несколько особенностей, оптимизирующих данный процесс. Во-первых, при наличии ограничивающих предикатов на поле сортировки, они создают верхнюю и/или нижнюю границы сканирования. То есть в случае запроса (`WHERE A > 0 ORDER BY A`) будет выполнено

частичное сканирование индекса вместо полного. Этим мы сокращаем расходы на собственно сканирование. Во-вторых, при наличии прочих ограничивающих предикатов (не на поле сортировки), однако оптимизированных через индекс, включается комплексный режим работы, где сканирование индекса совмещено с использованием битовой карты. Рассмотрим, как это работает. Пусть у нас есть запрос вида (`WHERE A > 0 AND B > 0 ORDER BY A`). В данном случае, сначала выполняется сканирования диапазона для индекса по полю `B` и составляется битовая карта. Далее значение `0` устанавливается нижней границей сканирования индекса по полю `A`. Затем мы сканируем данный индекс начиная от нижней границы и для каждого извлеченного из индекса номера записи проверяем его входимость в битовую карту. И только в случае входимости производим фетч записи. Этим мы сокращаем расходы на фетч страниц данных.

Основное отличие индексной навигации от сканирования (описанного в Сканирование диапазона) заключается в отсутствии битовой карты между сканированием индекса и доступом к записи через ее идентификатор. Причина понятна — сортировка записей по физическим номерам в данном случае противопоказана. Из этого можно сделать вывод, что при индекс сканируется по мере фетча с клиента, а не “за раз” (как в случае использования битовой карты).

Стоимость навигации оценить достаточно сложно. Для вычисления `MIN/MAX` в подавляющем большинстве случаев она будет равна высоте B+ дерева (поиск первого ключа) плюс единица (фетч страницы). Стоимость такого доступа считается пренебрежительно малой величиной, так как на практике описанное вычисление `MIN/MAX` всегда будет быстрее альтернативных вариантов. Для оценки же стоимости индексной навигации надо учесть как количество и среднюю ширину ключей индекса, так и кардинальность битовой карты (если таковая есть), а также иметь представление о факторе кластеризации (clustering factor) индекса — коэффициенте соответствия расположения ключей физическим номерам записей. В настоящий момент в сервере отсутствует вычисление стоимости навигации, то есть снова используется стратегия на основе правил (rule based). В будущем планируется использовать данных подход только для `MIN/MAX` и `FIRST`, а в остальных случаях опираться на стоимость.

Фактор кластеризации (clustering factor) индекса не хранится в статистике индекса, но вы можете его посмотреть используя утилиту `gstat`.

В Firebird 6.0 добавлена стоимостная оценка для выбора между навигацией по индексу и внешней сортировкой.

В Legacy плане навигация по индексу обозначается словом “ORDER”, за которым следует наименование индекса (без скобок, так как такой индекс может быть только один). Сервер сообщает также и индексы, образующие битовую карту, используемую для фильтрации. В этом случае в плане выполнения присутствуют оба слова: сначала “ORDER”, потом “INDEX”.

Пример 19. Навигация по индексу с полным сканированием индекса

```
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
ORDER BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0)
```

```
[cardinality=265.0, cost=302.000]
Select Expression
    [cardinality=265.0, cost=302.000]
    -> Table "RDB$RELATIONS" Access By ID
        -> Index "RDB$INDEX_0" Full Scan
```

Пример 20. Навигация по индексу с предикатом, создающим нижнюю границу сканирования

```
SELECT MIN(RDB$RELATION_NAME)
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0)
```

```
[cardinality=1.0, cost=26.785]
Select Expression
    [cardinality=1.0, cost=26.785]
    -> Aggregate
        [cardinality=18.785, cost=26.785]
        -> Filter
            [cardinality=18.785, cost=26.785]
            -> Table "RDB$RELATIONS" Access By ID
                -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Пример 21. Навигация по индексу вместе с битовой картой

```
SELECT MIN(RDB$RELATION_NAME)
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID > ?
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0 INDEX (RDB$INDEX_1))
```

```
[cardinality=1.0, cost=159.000]
Select Expression
    [cardinality=1.0, cost=159.000]
    -> Aggregate
        [cardinality=125.0, cost=159.000]
        -> Filter
            [cardinality=125.0, cost=159.000]
            -> Table "RDB$RELATIONS" Access By ID
                -> Index "RDB$INDEX_0" Full Scan
                    -> Bitmap
                        -> Index "RDB$INDEX_1" Range Scan (lower bound: 1/1)
```

2.3. Доступ посредством RDB$DB_KEY

Механизм доступа через RDB$DB_KEY в основном используется сервером для внутренних целей. Однако его можно задействовать и в пользовательских запросах.

С точки зрения методов доступа тут все просто — создается битовая карта и в нее помещается единственной номер записи — значение RDB$DB_KEY. После чего выполняется чтение записи посредством описанного выше доступа через идентификатор записи. Очевидно, что стоимость такого доступа равна единице. То есть получаем что-то вроде псевдо-индексного доступа, что и отражается в плане выполнения запроса.

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$DB_KEY = ?
```

```
PLAN (RDB$RELATIONS INDEX ())
```

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Filter
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> DBKEY
```

Чтения через RDB$DB_KEY отражаются в статистике как индексированные. Причина этого понятна из описания выше — все, что читается посредством доступа через идентификатор записи, считается сервером индексированным доступом. Не совсем корректное, но достаточно безобидное допущение.

2.4. Внешняя таблица (External table scan)

Данный вид доступа используется только при работе с внешними таблицами. По сути он представляет собой аналог полного сканирования таблицы. Записи читаются из внешнего файла поштучно, посредством текущего указателя (смещения). Страничный кеш не используется. Индексирование внешних таблиц не поддерживается.

По причине отсутствия альтернативных методов доступа к внешним данным, стоимость не вычисляется и не используется. Кардинальность выборки из внешней таблицы оценивается как `File_size / Record_size`.

В Legacy плане чтение из внешней таблицы отображается словом “NATURAL”.

В Explain плане чтение из внешней таблицы отображается как “Table Full Scan”.

Пример 22. Чтение внешней таблицы

```
SELECT *
FROM EXT_TABLE
```

```
PLAN (EXT_TABLE NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Table "EXT_TABLE" Full Scan
```

2.5. Виртуальная таблица (Virtual table scan)

Данный метод доступа используется для чтения данных из виртуальных таблиц (таблиц мониторинга MON$, виртуальных таблиц безопасности SEC$, а также виртуальной таблиц RDB$CONFIG). Данные таких таблиц формируются налету и кешируются в памяти. Например, все таблицы мониторинга заполняются при первом обращении к любой из них, а их данные сохраняются до конца транзакции. Виртуальная таблица RDB$CONFIG также заполняется при первом обращении, а её данные сохраняются до конца сессии.

По причине отсутствия альтернативных методов доступа к виртуальным таблицам, стоимость не вычисляется и не используется. Кардинальность виртуальных таблиц принимается равной 1000.

В Legacy плане чтение из виртуальной таблицы отображается словом “NATURAL”.

В Explain плане чтение из виртуальной таблицы отображается как “Table Full Scan”.

Пример 23. Чтение виртуальной таблицы MON$ATTACHMENT

```
SELECT *
FROM MON$ATTACHMENTS
```

```
PLAN (MON$ATTACHMENTS NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Table "MON$ATTACHMENTS" Full Scan
```

2.6. Локальная временная таблица (Local table)

Данный метод доступа доступен начиная с Firebird 5.0. Он используется для возврата вставленных, модифицированных или удалённых записей DML операторами содержащими предложение `RETURNING` и возвращающих курсор. К таким операторам относятся `INSERT … SELECT … RETURNING`, `UPDATE… RETURNING`, `DELETE… RETURNING`, `MERGE… RETURNING`. Оператор `INSERT … VALUES .. RETURNING`, который может вставить и вернуть только одну запись к ним не относится.

Локальные временные таблицы создаются "на лету" со столбцами перечисленными в предложении `RETURNING` и хранят свои данные и структуру до окончания выборки строк из DML оператора.

В будущих версиях Firebird планируется добавить возможность объявлять и заполнять локальные временные таблицы в секции объявления локальных переменных PSQL модулей, как `DECLARE LOCAL TABLE`.

По причине отсутствия альтернативных методов доступа к локальным таблицам, стоимость не вычисляется и не используется. Кардинальность локальных временных таблиц принимается равной 1000.

В Legacy и Explain планах чтение из локальной временной таблицы отображается отдельно от основного плана (как для подзапросов), сразу после него. В Legacy плане чтение из локальной таблицы отображается словом “NATURAL”, а таблица имеет имя `Local_Table`. В Explain плане чтение из локальной временной таблицы отображается как “Local Table Full Scan”.

Пример 24. Использование локальной временной таблицы для DML оператора с предложением `RETURNING`.

```
UPDATE COLOR
SET NAME = ?
WHERE NAME = ?
RETURNING CODE_COLOR, NAME, OLD.NAME AS OLD_NAME
```

```
PLAN (COLOR INDEX (UNQ_COLOR_NAME))
PLAN (Local_Table NATURAL)
```

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Filter
        [cardinality=1.0, cost=4.0]
        -> Table "COLOR" Access By ID
            -> Bitmap
                -> Index "UNQ_COLOR_NAME" Unique Scan
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Local Table Full Scan
```

2.7. Процедурный доступ

Данный метод доступа используется при выборке из хранимых процедур, использующих предложение `SUSPEND` для возврата результата. В Firebird хранимая процедура всегда представляет собой черный ящик, о внутренностях и деталях реализации которого сервер не делает никаких предположений. Процедура всегда считается недетерминированным источником данных, то есть может возвращать разные данные при двух последующих вызовах, выполненных в равных условиях. В свете этого очевидно, что любого вида индексация результатов процедуры невозможна. Процедурный метод доступа также представляет собой аналог полного сканирования таблицы. При каждом фетче из процедуры она выполняется с момента предыдущего останова (начала процедуры для первого фетча) до следующего предложения `SUSPEND`, после чего ее выходные параметры формируют строку данных, которая и возвращается из данного метода доступа.

Аналогично доступу для внешних таблиц, стоимость процедурного доступа также не рассчитывается и не учитывается, так как нет альтернативных вариантов. Для хранимых процедур кардинальность принимается равной 1000.

В Legacy плане выполнения процедурный отображается как “NATURAL”.

В старых версиях Firebird (до 3.0) может выводиться детализация (планы) выборок внутри процедуры. Это позволяет оценить план выполнения запроса в целом, с учетом “внутренностей” всех участвующих процедур, но формально это является неверным.

Пример 25. Процедурный доступ

```
SELECT *
FROM PROC_TABLE
```

```
PLAN (PROC_TABLE NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Procedure "PROC_TABLE" Scan
```

Глава 3. Фильтры

Данная группа методов доступа является преобразователем полученных данных. Главное отличие этой группы от прочих заключается в том, что у всех фильтров присутствует только один вход. Они не изменяют ширину входного потока, а лишь уменьшают его кардинальность по правилам, определяемым своей функцией. На вход фильтру может поступать поток данных как из первичного, так и из любого другого источника данных.

С точки зрения реализации, у фильтров есть еще одна общая черта — для них не оценивается стоимость. То есть считается, что все фильтры выполняются за нулевое время.

Далее будут описаны существующие в Firebird реализации методов-фильтров и решаемые ими задачи.

3.1. Проверка предикатов

Наверное, это наиболее часто используемый случай, и заодно самый простой для понимания. Данный фильтр проверяет некоторое логическое условие для каждой записи, переданной ему на вход. Если это условие выполняется, то запись без изменений пропускается на выход, иначе игнорируется. В зависимости от входных данных и логического условия, фетч одной записи из данного фильтра может привести к одному или более фетчей из входного потока. Вырожденными случаями проверочных фильтров являются предопределенные условия вида (1 = 1) или (1 <> 1), которые либо просто пропускают входные данные через себя, либо удаляют их.

Как можно понять из названия, данные фильтры применяются для выполнения предикатов в предложениях WHERE, HAVING, ON и других. С целью уменьшения результирующей кардинальности выборки, проверка предикатов всегда ставится как можно “ниже” (“глубже”) в дерево выполнения запроса. В случае проверки на поле таблицы, проверка предиката будет выполнена сразу после фетча записи из этой таблицы.

Каждый предикат имеет свою селективность и таким образом приводит к уменьшению кардинальности выходного потока. Для неиндексированного доступа предикаты имеют следующую селективность:

Таблица 2. Селективность предикатов

Предикаты	Селективность
=, IS NULL, IS NOT DISTINCT FROM	0.1
<, <=, >, >=, <>, IS NOT NULL, IS DISTINCT FROM	0.5
BETWEEN	0.25
STARTING WITH, CONTAINING	0.5
IN (<list>)	0.1 * N
IN (<select>), EXISTS(<select>)	0.5

Селективность предикатов объединённых через оператор AND умножается, а через OR суммируется.

Таким образом, кардинальность выходного потока вычисляется по формуле

```
cardinality = selectivity * inputCardinality
```

В Legacy плане проверка предикатов не отображается.

В Explain плане проверка предикатов отображается с помощью слова “Filter”, при этом сам предикат не выводится.

Пример 26. Проверка предиката фильтрации

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 0
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Filter
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Проверка предиката также отображается и при использовании индексированного доступа.

Пример 27. Проверка предиката фильтрации при использовании индексированного доступа

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME = ?
```

```
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Filter
        [cardinality=1.0, cost=4.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" Unique Scan
```

По последнему примеру может возникнуть вопрос: зачем использован Filter, если предикат реализуется через `INDEX UNIQUE SCAN`. Дело в том, что Firebird реализует нечеткий поиск в индексе. То есть индексное сканирование является лишь оптимизацией вычисления предиката и в ряде случаев может вернуть больше записей, чем требуется. Именно поэтому сервер не полагается лишь на результат индексного сканирования и проверяет предикат еще раз, после фетча записи. Отмечу, что в подавляющем большинстве случаев это не несет заметных накладных расходов с точки зрения производительности. В этом случае проверка предиката не изменяет оценку кардинальности, поскольку она уже получена при индексном доступе (отфильтровано по индексу).

3.1.1. Проверка инвариантных предикатов

Предикат является инвариантным, если его значение не зависит от полей фильтруемых потоков. Простейшим примером инвариантных предикатов являются условия вида `1=0` и `1=1`. Инвариантные предикаты также могут содержать и маркеры параметров, например `? = 1`.

Начиная с Firebird 5.0 инвариантные и при этом детерминированные предикаты распознаются оптимизатором и вычисляются однократно. В отличие от обычных предикатов фильтрации инвариантные предикаты вычисляются как можно раньше, их проверка всегда ставится как можно “выше” в дерево выполнения запроса. Если инвариантный предикат фильтрации имеет значение `FALSE`, то извлечение записей из нижележащих источников данных немедленно прекращается.

В отличие от обычных предикатов фильтрации, фильтрация инвариантным предикатом не изменяет кардинальность выходного потока.

В Legacy плане проверка инвариантных предикатов не отображается. В Explain плане проверка инвариантных предикатов отображается с помощью слова “Filter (preliminary)”, при этом сам предикат не выводится.

Пример 28. Проверка инвариантных предикатов

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 0
  AND 1 = ?
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Filter (preliminary)
        [cardinality=35.0, cost=350.0]
        -> Filter
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" Full Scan
```

3.2. Сортировка

Также известна как внешняя сортировка (external sort). Данный фильтр применяется оптимизатором при необходимости упорядочить входной поток в случае невозможности применения индексной навигации (см. Навигация по индексу). Примеры его использования: сортировка или группировка (в случае, когда нет подходящих индексов или индексы неприменимы для входного потока), построение B+ дерева индекса, выполнение операции DISTINCT, подготовка данных для однопроходного слияния (см. ниже) и другие.

Так как входные данные по определению неупорядоченны, то очевидно, что фильтр сортировки должен выполнить фетч всех записей из своего входного потока прежде чем он сможет выдать хоть одну из них на выход. Таким образом, данный фильтр можно считать буферизируемым источником данных.

Внешняя сортировка выполняется следующим образом. Набор входных записей помещается во внутренний буфер, после чего он сортируется алгоритмом быстрой сортировки (quick sort) и блок перемещается во внешнюю память. Далее таким же образом заполняется следующий блок и процесс продолжается до окончания записей во входном потоке. После чего заполненные блоки вычитываются и по ним строится бинарное дерево слияния. При чтении из фильтра сортировки происходит разбор дерева и слияние записей в один проход. Внешней памятью может выступать как виртуальная память, так и дисковое пространство, в зависимости от настроек файла конфигурации сервера.

При выполнении внешней сортировки Firebird записывает как ключевые поля (то есть, которые указаны в предложении `ORDER BY` или `GROUP BY`), так и неключевые поля (все остальные поля, на которые имеются ссылки внутри запроса) в блоки сортировки, которые либо сохраняются в памяти, либо во временные файлы. После завершения сортировки эти поля считываются обратно из блоков сортировки.

У сортировки существует два режима работы: “нормальный” и “усекающий”. Первый из них сохраняет записи с дублирующимися ключами сортировки, в то время как второй приводит к тому, что дубликаты удаляются. Именно “усекающий” режим реализует операцию DISTINCT, например.

Стоимость для внешней сортировки не рассчитывается. В нормальном режиме сортировки оценка кардинальности выходного потока не изменяется, в усекающем — кардинальность делится на 10.

В Legacy плане выполнения сортировка обозначается словом “SORT”, за которым в скобках следует описание входного потока.

В Explain плане выполнения сортировка обозначается словом “Sort” для нормального режима и “Unique Sort” для усекающего режима. Далее в скобках указаны длина сортируемых записей (record length) и длина ключа сортировки (key length). Сортируемые записи всегда включают ключ сортировки.

Оценить потребление памяти для сортировки можно умножив длину сортируемых записей (record length) на кардинальность сортируемого потока. По умолчанию Firebird будет пытаться выполнить сортировку в оперативной памяти, как только эта память будет исчерпана, движок начнёт использовать временные файлы. Объём оперативной памяти доступный для сортировок устанавливается параметром `TempCacheLimit`. В архитектуре Classic этот лимит устанавливается для каждого соединения, а в архитектурах SuperServer и SuperClassic для всех соединений базы данных.

Пример 29. Использование внешней сортировки в нормальном режиме

```
SELECT RDB$RELATION_NAME, RDB$SYSTEM_FLAG
FROM RDB$RELATIONS
ORDER BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Sort (record length: 284, key length: 8)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Пример 30. Использование внешней сортировки в усекающем режиме

```
SELECT DISTINCT
    RDB$RELATION_NAME, RDB$SYSTEM_FLAG
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Unique Sort (record length: 284, key length: 264)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Пример 31. Использование внешней сортировки совместно с фильтрацией

```
SELECT RDB$RELATION_NAME, RDB$SYSTEM_FLAG
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?
ORDER BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

```
[cardinality=125.0, cost=159.000]
Select Expression
    [cardinality=125.0, cost=159.000]
    -> Sort (record length: 284, key length: 8)
        [cardinality=125.0, cost=159.000]
        -> Filter
            [cardinality=125.0, cost=159.000]
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Возможные варианты избыточной сортировки (например, `DISTINCT` и `ORDER BY` по одному и тому же полю) устраняются оптимизатором и приводятся лишь к минимально необходимому числу сортировок.

3.2.1. Refetch

Начиная с Firebird 4.0 появилась оптимизация сортировки для широких наборов данных, которая отображается в Explain плане как `Refetch`. Под “широким набором данных”

понимается набор данных в котором суммарная длина полей записи велика.

При выполнении внешней сортировки Firebird записывает как ключевые поля (то есть, которые указаны в предложении ORDER BY или GROUP BY), так и неключевые поля (все остальные поля, на которые имеются ссылки внутри запроса) в блоки сортировки, которые либо сохраняются в памяти, либо во временные файлы. После завершения сортировки эти поля считываются обратно из блоков сортировки. Обычно этот подход считается более быстрым, поскольку записи считываются из временных файлов в порядке соответствующему отсортированным записям, а не выбираются случайным образом со страницы данных. Однако если неключевые поля большие (например, используются длинные VARCHAR), то это увеличивает размер блоков сортировки и, таким образом, приводит к большему количеству операций ввода-вывода для временных файлов.

Альтернативный подход (Refetch) заключается в том, что внутри блоков сортировки хранятся только ключевые поля и DBKEY записей всех участвующих таблиц, а неключевые поля извлекаются из страниц данных после сортировки. Это повышает производительность сортировки в случае длинных неключевых полей. Из описания Refetch ясно, что применяться он может только в случае нормального режима сортировки, для усекающего режима Refetch не применим, поскольку в этом режиме DBKEY не попадают в блоки сортировки.

Для внешней сортировки с использованием Refetch стоимость не вычисляется. Для выбора метода, которым будут сортироваться данные оптимизатор использует параметр InlineSortThreshold. Значение, указанное для InlineSortThreshold, определяет максимальный размер записи сортировки (в байтах), которая может храниться встроено, то есть внутри блока сортировки. Ноль означает, что записи всегда перечитываются (Refetch). Оптимальное значение данного параметра необходимо подбирать экспериментальным путём. Значение по умолчанию равно 1000 байт.

В Legacy плане выполнения Refetch и внешняя сортировка отображается одинаково, как SORT.

В Explain плане выполнения сортировка с использованием Refetch, отображается как дерево, где в качестве корня указано слово “Refetch”, на нижнем уровне указано слово “Sort” . Далее в скобках указаны длина сортируемых ключей + DBKEY используемых таблиц (record length) и длина ключа сортировки (key length).

Пример 32. Оптимизация внешней сортировки с использованием Refetch

```
SELECT *
FROM RDB$RELATIONS
ORDER BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Refetch
        -> Sort (record length: 28, key length: 8)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" Full Scan
```

Из Explain плана видно, что сначала происходит сортировка по ключам-сортировки, в блоки сортировки попадает сами ключи + DBKEY таблицы, а затем полные записи извлекаются из таблицы в отсортированном порядке по DBKEY методом `Refetch`.

Теперь попробуем задействовать `Refetch` для усекающего режима сортировки.

Пример 33. В усекающем режиме сортировки, Refetch не задействован

```
SELECT DISTINCT *
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Unique Sort (record length: 1438, key length: 1412)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Здесь несмотря на то, что необходимо сортировать широкие записи, `Refetch` не может быть задействован, а потому применяется обычная внешняя сортировка.

3.3. Агрегация

Данный фильтр используется исключительно для вычисления агрегирующих функций (`MIN`,

MAX, AVG, SUM, COUNT ...), включая случаи их использования в группировках.

Реализация довольна тривиальна. Все перечисленные функции требуют единственного временного регистра для хранения текущего граничного (MIN/MAX) или аккумулированного (прочие функции) значения. Далее для каждой записи из входного потока мы проверяем или суммируем этот регистр. В случае группировки алгоритм несколько усложняется. Для корректного вычисления функции для каждой группы должны быть четко определены границы между этими группами. Самым простым решением является гарантия упорядоченности входного потока. В данном случае мы выполняем агрегирование для одного и того же ключа группировки, а после его изменения выдаем результат на выход и продолжаем уже со следующим ключом. Данный подход как раз и использован в Firebird — агрегация по группам строго зависит от наличия сортировки на входе. Входной поток может быть упорядочен с помощью внешней сортировки или навигации по индексу (если она возможна). Метод Refetch не может быть использован для упорядочивания входного потока для группировки.

Существует альтернативный способ вычисления агрегатов — хеширование. В этом случае сортировка входного потока не требуется, однако к каждому ключу группировки применяется хеш-функция и регистр хранится для каждого ключа (точнее, для каждой коллизии ключа) в хеш-таблице. Плюс данного алгоритма — нет необходимости в дополнительной сортировке. Минус — больший расход памяти для хранения хеш-таблицы. Данный метод обычно выигрывает у сортировки при низкой селективности ключей группировки (мало уникальных значений, следовательно небольшая хеш-таблица). Агрегация хешированием в Firebird отсутствует.

Агрегация хешированием планируются к реализации в следующих версиях сервера.

Если агрегатные функции используются без группировки, то кардинальность выходного потока всегда равна единице. При использовании группировки входная кардинальность делится на 1000. Стоимость агрегации не вычисляется.

Группировка может использоваться и без агрегатных функций, в этом случае она является аналогом SELECT DISTINCT для устранения дубликатов записей, но в отличие от DISTINCT может использовать как внешнюю сортировку, так и навигацию по индексу.

В Legacy плане выполнения агрегация не показывается.

В Explain плане агрегация отображается с использованием слова “Aggregate”.

Пример 34. Вычисление агрегатной функции MAX

```
SELECT MAX(RDB$RELATION_ID)
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=1.0, cost=350.0]
Select Expression
    [cardinality=1.0, cost=350.0]
    -> Aggregate
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Поскольку для поля RDB$RELATION_ID существует ASCENDING индекс, то вычисление агрегатной функции MIN можно выполнить с помощью навигации по индексу (см. Навигация по индексу).

Пример 35. Вычисление агрегатной функции MIN с использованием навигации по индексу

```
SELECT MIN(RDB$RELATION_ID)
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_1)
```

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Aggregate
        [cardinality=1.0, cost=4.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> Index "RDB$INDEX_1" Full Scan
```

Пример 36. Вычисление агрегатной функции с группировкой. Для разделения групп используется внешняя сортировка.

```
SELECT RDB$SYSTEM_FLAG, COUNT(*)
FROM RDB$FIELDS
GROUP BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$FIELDS NATURAL)
```

```
[cardinality=7.756, cost=7756.0]
Select Expression
    [cardinality=7.756, cost=7756.0]
    -> Aggregate
        [cardinality=7756.0, cost=7756.0]
        -> Sort (record length: 28, key length: 8)
            [cardinality=7756.0, cost=7756.0]
            -> Table "RDB$RELATIONS" Full Scan
```

Пример 37. Вычисление агрегатной функции с группировкой. Для разделения групп используется навигация по индексу.

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=2.4, cost=4381.0]
Select Expression
    [cardinality=2.4, cost=4381.0]
    -> Aggregate
        [cardinality=2400.0, cost=4381.0]
        -> Table "RDB$RELATION_FIELDS" Access By ID
            -> Index "RDB$INDEX_4" Full Scan
```

Пример 38. Вычисление агрегатной функции с группировкой и фильтрацией

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
WHERE RDB$RELATION_NAME > ?
GROUP BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=0.13, cost=237.3]
Select Expression
    [cardinality=0.13, cost=237.3]
    -> Aggregate
        [cardinality=130.1, cost=237.3]
        -> Filter
            [cardinality=130.1, cost=237.3]
            -> Table "RDB$RELATION_FIELDS" Access By ID
                -> Index "RDB$INDEX_4" Range Scan (lower bound: 1/1)
```

В случае вычисления функций SUM/AVG/COUNT в режиме DISTINCT для каждой группы значений перед аккумулированием производится их сортировка в режиме устранения дубликатов. При этом в плане не отображается слово SORT.

Пример 39. Вычисление агрегатной функции с группировкой и фильтрацией

```
SELECT RDB$RELATION_NAME, COUNT(DISTINCT RDB$NULL_FLAG)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=2.4, cost=4381.0]
Select Expression
    [cardinality=2.4, cost=4381.0]
    -> Aggregate
        [cardinality=2400.0, cost=4381.0]
        -> Table "RDB$RELATION_FIELDS" Access By ID
            -> Index "RDB$INDEX_4" Full Scan
```

3.3.1. Фильтрация в предложении HAVING

Предикаты в предложении HAVING могут содержать агрегатные функции или поля и выражения группировки. Если предикат применяется к агрегатной функции, то

фильтрация происходит после группировки и расчёта агрегатов. Если предикат применяется к полям и выражением группировки, то произойдёт проталкивание предиката внутрь, таким образом, он будет вычислен до группировки и агрегации.

Пример 40. Фильтрация по агрегатной функции в предложении HAVING

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
HAVING COUNT(*) > ?
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=1.2, cost=4381.0]
Select Expression
    [cardinality=1.2, cost=4381.0]
    -> Filter
        [cardinality=2.4, cost=4381.0]
        -> Aggregate
            [cardinality=2400.0, cost=4381.0]
            -> Table "RDB$RELATION_FIELDS" Access By ID
                -> Index "RDB$INDEX_4" Full Scan
```

Пример 41. Фильтрация по столбцу группировки

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
HAVING RDB$RELATION_NAME > ?
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=1.0, cost=237.3]
Select Expression
    [cardinality=1.0, cost=237.3]
    -> Filter
        [cardinality=0.13, cost=237.3]
        -> Aggregate
            [cardinality=130.1, cost=237.3]
            -> Filter
                [cardinality=130.1, cost=237.3]
                -> Table "RDB$RELATION_FIELDS" Access By ID
                    -> Index "RDB$INDEX_4" Range Scan (lower bound: 1/1)
```

В последнем примере предикат фильтрации был “протолкнут внутрь”, однако после вычисления агрегатов фильтрация применена ещё раз. Надо отметить, что повторная фильтрация выполнила нормализацию кардинальности (по идее агрегатные функции с группировкой и без никогда не могут возвращать меньше одной записи).

3.4. Счетчики

Данный вид фильтра также очень прост. Его цель — выдать только часть записей входного потока, основываясь на некотором значении N внутреннего счетчика. Есть две разновидности данного фильтра, применяемые для реализаций предложений FIRST/SKIP/ROWS/FETCH FIRST/OFFSET запроса. Первая разновидность (FIRST-счетчик) выдает на выход только первые N записей своего входного потока, после чего выдает признак EOF. Вторая разновидность (SKIP-счетчик) игнорирует первые N записей своего входного потока и начинает выдавать их на выход, начиная с записи N+1. Очевидно, что при наличии в запросе ограничения выборки вида (`FIRST 100 SKIP 100`), сначала должен быть применен SKIP-счетчик, и только после него — FIRST-счетчик. Оптимизатор гарантирует правильное применение счетчиков при выполнении запроса.

Счётчик SKIP не изменяет кардинальность выходного потока. Кардинальность выходного потока после счётчика FIRST равна значению указанному для этого счётчика, если значение не является литералом, то кардинальность принимается равной 1000.

В Legacy плане выполнения счётчики не отображаются

В Explain плане FIRST-счётчик отображается как “First N Records”, а SKIP-счётчик — “Skip N Records”.

Пример 42. Использование FIRST-счётчика

```
SELECT *
FROM RDB$RELATIONS
FETCH FIRST 10 ROWS ONLY
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=10.0, cost=350.5]
Select Expression
    [cardinality=10.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Пример 43. Использование FIRST(?)

```
SELECT FIRST(?) *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=1000.0, cost=350.5]
Select Expression
    [cardinality=1000.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Пример 44. Использование SKIP-счётчика

```
SELECT *
FROM RDB$RELATIONS
OFFSET 10 ROWS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.5, cost=350.5]
Select Expression
    [cardinality=350.5, cost=350.5]
    -> Skip N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Пример 45. Одновременное использование FIRST и SKIP счётчиков

```
SELECT FIRST(10) SKIP(20) *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=10.0, cost=350.5]
Select Expression
    [cardinality=10.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Skip N Records
            [cardinality=350.5, cost=350.5]
            -> Table "RDB$RELATIONS" Full Scan
```

3.5. Проверка сингулярности

Этот фильтр используется для гарантии возврата из источника данных только одной записи. Он применяется в случае использовании в тексте запроса сингулярных подзапросов (singleton subqueries).

Для выполнения своей функции, фильтр проверки сингулярности производит два чтения из входного потока. Если второе чтение вернуло EOF, то выдаем первую запись на выход. Иначе инициируем ошибку `isc_sing_select_err` (“multiple rows in singleton select”).

Кардинальность выходного потока после проверки сингулярности равна 1.

В Legacy плане выполнения проверка сингулярности не отображается.

В Explain плане проверка сингулярности отображается как “Singularity Check”.

Пример 46. Проверка сингулярности коррелированного подзапроса

```
SELECT
  (SELECT RDB$RELATION_NAME FROM RDB$RELATIONS R
   WHERE R.RDB$RELATION_ID = D.RDB$RELATION_ID - 1) AS LAST_RELATION,
  D.RDB$SQL_SECURITY
FROM RDB$DATABASE D
```

```
PLAN (R INDEX (RDB$INDEX_1))
PLAN (D NATURAL)
```

```
[cardinality=1.0, cost=4.62]
Sub-query
    [cardinality=1.0, cost=4.62]
    -> Singularity Check
        [cardinality=1.62, cost=4.62]
        -> Filter
            [cardinality=1.62, cost=4.62]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_1" Range Scan (full match)
[cardinality=1.0, cost=5.62]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Table "RDB$DATABASE" as "D" Full Scan
```

В Legacy форме первый план относится к подзапросу, а второй к основному запросу.

Пример 47. Проверка сингулярности некоррелированного подзапроса

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID = ( SELECT RDB$RELATION_ID - 1 FROM RDB$DATABASE )
```

```
PLAN (RDB$DATABASE NATURAL)
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_1))
```

```
[cardinality=1.0, cost=1.0]
Sub-query (invariant)
    [cardinality=1.0, cost=1.0]
    -> Singularity Check
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" Full Scan
[cardinality=1.62, cost=5.62]
Select Expression
    [cardinality=1.62, cost=4.62]
    -> Filter
        [cardinality=1.62, cost=4.62]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_1" Range Scan (full match)
```

3.6. Блокировка записи

Данный фильтр реализует пессимистическую блокировку записи. Она применяется только в случае присутствия в тексте запроса предложения `WITH LOCK`. Каждая запись, прочитанная из входного потока фильтра, возвращается на выход уже заблокированной текущей транзакцией. Для блокировки используется создание новой версии записи, помеченной идентификатором текущей транзакции. В случае, если текущая транзакция уже изменяла данную запись, то блокировка не выполняется.

В текущих версиях фильтр блокировки работает только для первичных методов доступа. Она не изменяет кардинальность выходного потока данных. В Legacy плане выполнения блокировка не отображается.

Пример 48. Блокировка записи

```
SELECT *
FROM COLOR
WITH LOCK
```

```
PLAN (COLOR NATURAL)
```

```
[cardinality=233.0, cost=233.0]
Select Expression
    [cardinality=233.0, cost=233.0]
    -> Write Lock
        [cardinality=233.0, cost=233.0]
        -> Table "COLOR" Full Scan
```

3.7. Условное ветвление потоков (Conditional Stream)

Условное ветвление потоков доступно начиная с Firebird 3.0. Этот метод доступа применяется в тех случаях, когда в зависимости от входного параметра таблица может быть прочитана либо с помощью полного сканирования, либо с использованием индексного доступа. Обычно такое возможно для условия фильтрации вида `INDEXED_FIELD = ? OR 1=?`.

Кардинальность выходного потока считается как средняя между кардинальностями вариантов извлечения данных. Стоимость так же рассчитывается как средняя стоимость между первым и вторым вариантом. Предполагается, что вероятность выполнения разных вариантов одинакова.

```
cardinality = (cardinality_option_1 + cardinality_option_2) / 2

cost = (cost_option_1 + cost_option_2) / 2
```

В Legacy плане условное ветвление не отображается, но отображаются оба варианта извлечения данных из таблицы, через запятую.

В Explain плане условное ветвление отображается как дерево, корень которого обозначен словом `Condition`, а листьями дерева является варианты извлечения данных из таблицы.

Пример 49. Условное ветвление потоков

```
SELECT *
FROM RDB$RELATIONS R
WHERE (R.RDB$RELATION_NAME = ? OR 1=?)
```

```
PLAN (R NATURAL, R INDEX (RDB$INDEX_0))
```

```
[cardinality=175.65, cost=177.15]
Select Expression
    [cardinality=175.65, cost=177.15]
    -> Filter
        [cardinality=175.65, cost=177.15]
        -> Condition
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=1.3, cost=4.3]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

3.8. Буферизация записей

Данный метод доступа используется как вспомогательный при реализации других более высокоуровневых методов доступа, таких как “Скользящее окно” и “Соединение хешированием”.

Буферизация записей предназначена для сохранения результатов нижележащих методов доступа в оперативную память, как только эта память будет исчерпана, движок начнёт использоваться временные файлы. Объём внутренней памяти доступный для буферизации записей устанавливается параметром `TempCacheLimit`. В архитектуре Classic этот лимит устанавливается для каждого соединения, а в архитектурах Classic и SuperClassic для всех соединений базы данных.

Оценить потребление памяти для буферизации можно умножив длину записей (record length) на кардинальность входного потока.

В Legacy плане буферизация записей не отображается.

В Explain плане буферизация записей отображается как “Record Buffer”, после чего в круглых скобках указана длина записи как (`record length: <length>`).

Примеры в которых используется буферизация записей будут показаны при описании методов доступа “Скользящее окно” и “Соединение хешированием”.

3.9. Скользящее окно (Window)

Данная группа методов доступа используется при вычислении так называемых оконных или аналитических функций.

Оконная функция выполняет вычисления для набора строк (rows), некоторым образом связанных с текущей строкой. Её действие можно сравнить с вычислением, производимым агрегатной функцией. Однако с оконными функциями строки не группируются в одну выходную строку, что имеет место с обычными, не оконными, агрегатными функциями. Вместо этого, эти строки остаются отдельными сущностями. Внутри же, оконная функция, как и агрегатная, может обращаться не только к текущей строке результата запроса. Набор строк над которым выполняет вычисления оконная функция называется **окном**.

Набор строк может быть разбит на секции. По умолчанию все строки окна включаются в одну секцию, это можно изменить, указав в `PARTITION BY` каким образом строки разбиваются на секции. Для каждой строки, оконная функция обсчитывает только строки, которые попадают в ту же самую секцию, что и текущая строка.

Внутри каждой секции можно задать порядок обработки строк оконной функцией. Это можно сделать в выражении окна с помощью `ORDER BY`.

Кроме того, в выражении окна можно указать рамку окна, которая определяет сколько строк в секции будет обработано. По умолчанию рамка окна зависит от типа оконной функции. Чаще всего, если указан порядок обработки строк (`ORDER BY` в предложении окна) — это от начала секции до текущей строки или значения.

Синтаксис оконных функций и выражений окон описан в главе “Оконные (аналитические) функции” руководства по языку SQL Firebird.

Из выше описанного следует, что методы доступа используемые при вычислении оконных функций не изменяют кардинальность выборки. Стоимость методов доступа для вычисления оконных функций не вычисляется, поскольку нет альтернативных вариантов.

Вычисление оконных функций начинается с буферизации результата выборки, который выполняется после всех остальных частей `SELECT` запроса, но до сортировки (`ORDER BY`), ограничения результатов выборки с помощью first/skip счётчиков, и вычисления выражений для столбцов в предложении `SELECT`. Данный буфер называется буфером окна. В Explain плане обозначается он как `Window Buffer` (см. также Буферизация записей).

Далее в оконном буфере определяются границы секций, что в Explain плане обозначается как `Window Partition`. Это делается даже если предложение `PARTITION BY` отсутствует (секция будет одна).

Если в выражении окна будет описано разделение на секции `PARTITION BY`, то исходная (одна секция) будет разделена на секции повторно. Для разделения на секции используется внешняя сортировка по ключам секционирования. Для определения порядка обработки строк (`ORDER BY` внутри окна) также используется внешняя сортировка. Firebird объединяет ключи секционирования и ключи для определения порядка строк в одну внешнюю сортировку. Если набор строк для внешней сортировки слишком широкий, то может быть

использован метод Refetch для оптимизации.

После разбиения на секции и сортировки строк секции над полученным набором вновь производится буферизация с помощью метода доступа Буферизация записей поверх которого будет использован метод разделения на секции `Window Partition`.

Таких `Window Partition + Record Buffer` и возможно внешних сортировок будет столько сколько разных окон используется в запросе.

Как только все секции определены, и строки внутри секций упорядочены, начинается вычисление самих оконных функций, которое отображено в Explain плане как `Window`. В процессе вычисления оконных функций внутри секции, набор строк для вычисления функции может либо расти от начала секции (`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`), либо уменьшать при движении к концу секции (`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`), либо иметь фиксированный размер перемещаться по секции относительно текущей строки (`ROWS BETWEEN BETWEEN 2 PRECEDING AND 2 FOLLOWING`), либо оставаться неизменным, если рамка окна определена как `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`). Именно поэтому данный метод доступа называют “Скользящее окно” (“Sliding window”).

В Legacy плане вычисление оконных функций никак не отображается, но если для разделения секций окна или упорядочивания строк внутри секции используется внешняя сортировка (SORT), то она будет отображена.

Пример 50. Вычисление простейшей оконной функции

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER() AS CNT
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Window Buffer
                [cardinality=350.0, cost=350.0]
                -> Record Buffer (record length: 273)
                    [cardinality=350.0, cost=350.0]
                    -> Table "RDB$RELATIONS" Full Scan
```

Здесь присутствует только одна секций, в которую сходят все записи запроса.

Пример 51. Вычисление секционированной оконной функции

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER(PARTITION BY RDB$SYSTEM_FLAG) AS CNT
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 546)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 556, key length: 8)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Window Buffer
                            [cardinality=350.0, cost=350.0]
                            -> Record Buffer (record length: 281)
                                [cardinality=350.0, cost=350.0]
                                -> Table "RDB$RELATIONS" Full Scan
```

Добавляем порядок обработки строк в секции, что также неявно задаст рамку окна (`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

Пример 52. Вычисление секционированной оконной функции с заданием порядка обработки строк

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER(PARTITION BY RDB$SYSTEM_FLAG ORDER BY RDB$RELATION_ID) AS CNT
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 546)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 564, key length: 16)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Window Buffer
                            [cardinality=350.0, cost=350.0]
                            -> Record Buffer (record length: 281)
                                [cardinality=350.0, cost=350.0]
                                -> Table "RDB$RELATIONS" Full Scan
```

Теперь посмотрим как изменится план, если будет использовано несколько оконных функций с разными окнами.

Пример 53. Вычисление нескольких оконных функций с разными окнами

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER W1 AS CNT,
  LAG(RDB$RELATION_NAME) OVER W2 AS PREV
FROM RDB$RELATIONS
WINDOW
  W1 AS (PARTITION BY RDB$SYSTEM_FLAG ORDER BY RDB$RELATION_ID),
  W2 AS (ORDER BY RDB$RELATION_ID)
```

```
PLAN SORT (SORT (RDB$RELATIONS NATURAL))
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 579)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 588, key length: 8)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Record Buffer (record length: 546)
                            [cardinality=350.0, cost=350.0]
                            -> Sort (record length: 564, key length: 16)
                                [cardinality=350.0, cost=350.0]
                                -> Window Partition
                                    [cardinality=350.0, cost=350.0]
                                    -> Window Buffer
                                        [cardinality=350.0, cost=350.0]
                                        -> Record Buffer (record length: 281)
                                            [cardinality=350.0, cost=350.0]
                                            -> Table "RDB$RELATIONS" Full Scan
```

Глава 4. Методы слияния

Эта категория методов доступа во многом напоминает фильтры. Методы слияния тоже преобразуют входные данные по определенному алгоритму. Формальная разница только в том, что данная группа методов всегда оперирует с несколькими входными потоками. Кроме того, обычным результатом их работы является либо расширение выборки по полям, либо увеличение ее кардинальности.

Существует два класса методов слияния — соединение (join) и объединение (union), выполняющие разные функции. Кроме того, к этой категории также относится и выполнение рекурсивных запросов, которое является специальным случаем объединений (union). Ниже мы познакомимся с их описанием и возможными алгоритмами реализации.

4.1. Соединения (Joins)

Как можно догадаться по названию, данная группа методов реализует SQL-соединения (joins). Стандарт SQL определяет два вида соединений: внутренние (inner) и внешние (outer). Кроме того, внешние соединения делятся на односторонние (left или right) и полные (full). У любого вида соединений есть два входных потока — левый и правый. Для внутреннего и полного внешнего соединений эти потоки семантически равноценны. В случае же одностороннего внешнего соединения один из потоков является ведущим (обязательным), а второй — ведомым (необязательным). Часто также ведущий поток называют внешним, а ведомый – внутренним. Для левого внешнего соединения ведущим (внешним) потоком является левый, а ведомым (внутренним) — правый. Для правого внешнего соединения — наоборот.

Сразу отмечу, что формально левое внешнее соединение эквивалентно инвертированному правому внешнему соединению. Под инверсией в данном контексте понимается замена внешнего потока на внутренний или наоборот. Так что достаточно реализовать только один вид одностороннего внешнего соединения (обычно это левое). Так и сделано в Firebird. Однако, решение о преобразовании правого соединения в левое (базовое) и его последующей оптимизации принимал оптимизатор, что зачастую приводило к некорректной оптимизации правых соединений. Начиная с версии 1.5, сервер превращает правые соединения в левые на уровне разбора BLR, что устраняет возможные конфликты в оптимизаторе.

У каждого соединения помимо входных потоков существует еще один атрибут — условие связи. Именно это условие и определяет результат, то есть как именно будут поставлены в соответствие данные входных потоков. При отсутствии данного условия получаем вырожденный случай — декартово произведение (cross join) входных потоков.

Вернемся к односторонним внешним соединениям. Их потоки не зря называются ведущим и ведомым. В данном случае внешний (ведущий) поток всегда должен быть прочитан перед внутренним (ведомым), иначе невозможно будет выполнить требуемую стандартом подстановку NULL-значений в случае отсутствия соответствий внутреннего потока внешнему. Отсюда можно сделать вывод, что оптимизатор не имеет возможности выбирать порядок выполнения одностороннего внешнего соединения и он всегда будет определяться текстом запроса. В то время как для внутренних и полных внешних соединений входные

потоки независимы и могут читаться в произвольном порядке, следовательно алгоритм выполнения таких соединений определяется исключительно оптимизатором и никак не зависит от текста запроса.

Помимо соединений представленных в SQL стандарте многие СУБД реализуют также дополнительные способы соединений: полу-соединение (semi-join) и анти-соединение (anti-join). Эти виды соединений не представлены в SQL непосредственно, но могут использоваться оптимизатором после трансформации дерева выполнения SQL запроса. Например они могут использоваться для выполнения предикатов фильтрации `EXISTS` и `NOT EXISTS`. Эти виды соединений по своей природе ассиметричны, то есть соединяемые потоки для них не равноценны — выделяют ведущий (обязательный) и ведомый (тот который проверяется в предикате фильтрации).

4.1.1. Соединение вложенными циклами (Nested loop)

Данный метод является наиболее распространенным в Firebird. В других СУБД этот алгоритм также называется соединением посредством вложенных циклов (nested loops join).

Суть его проста. Открывается один из входных потоков (внешний) и из него читается одна запись. После чего открывается второй поток (внутренний) и из него тоже читается одна запись. Затем проверяется условие связи. Если условие выполнено, эти две записи сливаются в одну и выдаются на выход. Далее читается вторая запись из внутреннего потока и процесс повторяется до выдачи из него EOF. Если внутренний поток не содержит ни одной соответствующей внешнему потоку записи, то возможны два варианта развития событий. В случае внутреннего соединения запись не возвращается на выход. В случае же внешнего соединения мы соединяем запись из внешнего потока с необходимым количеством NULL-значений и выдаем ее на выход. Далее, независимо от вида соединения, мы читаем вторую запись из внешнего потока и начинаем процесс итерации по внутреннему потоку заново. Очевиден факт, что данный алгоритм работает по конвейерному принципу и соединение потоков выполняется прямо в процессе клиентского фетча.

Ключевой особенностью этого метода соединения является “вложенная” выборка из внутреннего потока. Очевидно, что читать весь внутренний поток для каждой записи внешнего потока очень накладно. Поэтому соединение вложенными циклами работает эффективно только в случае наличия индекса, применимого к условию связи. В этом случае на каждой итерации из внутреннего потока будет выбрано подмножество записей, удовлетворяющих текущей записи внешнего потока. Стоит обратить внимание, что не каждый индекс подходит для эффективного выполнения данного алгоритма, а только соответствующий условию связи. Например, при связи вида (`ON SLAVE.F = 0`) даже при использовании индекса по полю `F` внутренней таблицы все равно из нее будут читаться одни и те же записи на каждой итерации, что есть напрасная трата ресурсов. В данной ситуации, соединение слиянием было бы более эффективным (см. ниже).

Можно ввести определение “зависимости потоков” как наличие полей обоих потоков в условии связи и сделать вывод, что соединение вложенными циклами эффективно только при зависимости потоков друг от друга.

Если вспомнить описание "Проверка предикатов", где описан принцип максимально “глубокого” помещения предикативных фильтров оптимизатором, то становится понятен один момент: индивидуальные табличные фильтры уйдут “ниже” методов соединения. Соответственно, в случае предиката вида (`WHERE MASTER.F = 0`) и отсутствии в таблице `MASTER` записей с полем `F`, равным нулю, обращений к внутреннему потоку соединения вообще не будет, так как в данном случае нет итерации по внешнему потоку (в нем нет записей).

Так как логические связки `AND` и `OR` оптимизируются через битовые карты, то соединение вложенными циклами может быть использован для всех видов условий связи, причем обычно достаточно эффективно.

Для внешних соединений соединение вложенными циклами выбирается всегда, поскольку в настоящее время нет альтернативных вариантов. Для внутренних соединений оптимизатор выбирает соединение вложенными циклами, если его стоимость дешевле альтернативных вариантов (Hash Join). Если для полей связи внутренних соединений нет подходящих индексов, то почти всегда выбираются альтернативные алгоритмы соединения, если они возможны. Кроме того, внутреннего соединения оптимизатор выбирает наиболее эффективный порядок связи потоков. Главные критерии выбора: кардинальности обоих потоков и селективность условия связи. Используется следующие формулы для оценки стоимости определенного порядка соединения:

```
cost = cardinality(outer) + cardinality(outer) * (indexScanCost + cardinality(inner) *
selectivity(link))

cardinality = cardinality(outer) * (cardinality(inner) * selectivity(link))
```

Последняя часть формулы определяет стоимость выборки из внутреннего потока на каждой итерации. Умножив ее на количество итераций, получаем общую стоимость выборки из внутреннего потока. Общая стоимость получается путем добавления стоимости выборки из внешнего потока. Из всех возможных перестановок выбирается вариант с наименьшей стоимостью. В процессе перебора вариантов отбрасываются заведомо худшие (на основании уже имеющейся стоимостной информации). В перестановках участвуют не только потоки соединяемые вложенными циклами (Nested Loop), но и другими алгоритмами соединений (Hash Join), стоимость которых вычисляется по другим правилам.

В Legacy плане выполнения соединяемые вложенными циклами отображается словом "JOIN", за которым в скобках через запятую описываются входные потоки.

В Explain плане выполнения соединяемые вложенными циклами отображается в виде дерева, корень которого подписан как “Nested Loop Join” после чего в круглых скобках указан тип соединения. Уровнем ниже описываются соединяемые потоки.

Nested Loop Join (inner)

Пример 54. Nested Loop Join (inner)

```
SELECT *
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, RF INDEX (RDB$INDEX_4))
```

```
[cardinality=3212.6, cost=4620.0]
Select Expression
    [cardinality=3212.6, cost=4620.0]
    -> Nested Loop Join (inner)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.2, cost=12.2]
        -> Filter
            [cardinality=9.2, cost=12.2]
            -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
```

Отметим один момент. Так как для внутреннего соединения все потоки равнозначны, оптимизатор в случае более чем двух потоков преобразует бинарное дерево в "плоский" вид. Выходит, что де-факто внутреннее соединение оперирует с более чем двумя входными потоками. На алгоритм это никак не влияет, однако объясняет разницу в синтаксисе планов для внутренних и внешних соединений.

Пример 55. Внутреннее соединение вложенными циклами множества потоков

```
SELECT *
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF ON RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
```

```
PLAN JOIN (RF NATURAL, F INDEX (RDB$INDEX_2), R INDEX (RDB$INDEX_0))
```

```
[cardinality=3212.6, cost=20152.4]
Select Expression
    [cardinality=3212.6, cost=20152.4]
    -> Nested Loop Join (inner)
        [cardinality=2428.0, cost=2428.0]
        -> Table "RDB$RELATION_FIELDS" as "RF" Full Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
        [cardinality=1.3, cost=4.3]
        -> Filter
            [cardinality=1.3, cost=4.3]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

Кросс-соединение (cross-join) также относится к внутренним соединениям и всегда выполняется вложенными циклами.

Пример 56. Nested Loop Join для CROSS JOIN

```
SELECT *
FROM RDB$PAGES
CROSS JOIN RDB$PAGES
```

```
PLAN JOIN (RDB$PAGES NATURAL, RDB$PAGES NATURAL)
```

```
[cardinality=261376.5625, cost=261376.5625]
Select Expression
    [cardinality=261376.5625, cost=261376.5625]
    -> Nested Loop Join (inner)
        [cardinality=511.25, cost=511.25]
        -> Table "RDB$PAGES" Full Scan
        [cardinality=511.25, cost=511.25]
        -> Table "RDB$PAGES" Full Scan
```

Nested Loop Join (outer)

Пример 57. Nested Loop Join для внешнего соединения

```
SELECT *
FROM RDB$RELATIONS R
LEFT JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, RF INDEX (RDB$INDEX_4))
```

```
[cardinality=3200.6, cost=4620.0]
Select Expression
    [cardinality=3200.6, cost=4620.0]
    -> Nested Loop Join (outer)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.2, cost=12.2]
        -> Filter
            [cardinality=9.2, cost=12.2]
            -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
```

Пример 58. Nested Loop Join для внешнего соединения нескольких потоков

```
SELECT *
FROM RDB$RELATIONS R
LEFT JOIN RDB$RELATION_FIELDS RF ON RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
LEFT JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
```

```
PLAN JOIN (JOIN (R NATURAL, RF INDEX (RDB$INDEX_4)), F INDEX (RDB$INDEX_2))
```

```
[cardinality=3200.6, cost=16003.0]
Select Expression
    [cardinality=3200.6, cost=16003.0]
    -> Nested Loop Join (outer)
        [cardinality=3200.6, cost=4620.0]
        -> Nested Loop Join (outer)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=9.2, cost=12.2]
            -> Filter
                [cardinality=9.2, cost=12.2]
                -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
```

Обратите внимание, в отличие от внутреннего соединения здесь Explain план содержит вложенные “Nested Loop Join”.

Nested Loop Join (semi)

В настоящее время данный вид соединения не используется оптимизатором Firebird.

Его суть состоит в следующем. Открывается один из входных потоков (внешний) и из него читается одна запись. После чего открывается второй поток (внутренний) и из него тоже читается одна запись. Затем проверяется условие связи. Если условие выполнено, то запись из внешнего потока выдаются на выход, и внутренний поток немедленно закрывается. Если условие связи не выполнено, то читается вторая запись из внутреннего потока и процесс повторяется до выдачи из него EOF. Затем внешний поток читает следующую запись и всё повторяется до выдачи из него EOF.

Данный метод доступа может быть использован для выполнения подзапросов в предикатах `EXIST` и `IN (<select>)`, а также других предикатов которые преобразуются в `EXIST` (`ANY`, `SOME`).

Кардинальность выходного потока оценить весьма сложно, но что можно сказать точно она никогда не превысит кардинальность внешнего потока. Предполагается, что условию связи соответствует половина записей внешнего потока.

```
cardinality = cardinality(outer) * 0.5

cost = cardinality(outer) + cardinality(outer) * cost(inner first row)
```

Здесь `cost(inner first row)` — стоимость выборки первой записи соответствующей условию связи. Поскольку предикат `EXISTS` может содержать сложный подзапрос, а не чтение из одной таблицы, то посчитать её не так просто. Так же достаточно очевидно, что для эффективной работы данного алгоритма оптимизатор для выполнения внутреннего подзапроса должен переключится на стратегию оптимизации `FIRST ROWS`.

Nested Loop Join (anti)

В настоящее время данный вид соединения не используется оптимизатором Firebird.

По сути данный вид соединения является противоположным “Nested Loop Join (semi)”. Открывается один из входных потоков (внешний) и из него читается одна запись. После чего открывается второй поток (внутренний) и из него тоже читается одна запись. Затем проверяется условие связи. Если условие не выполнено, то запись из внешнего потока выдаются на выход, и внутренний поток немедленно закрывается. Если условие связи выполнено, то читается вторая запись из внутреннего потока и процесс повторяется до выдачи из него EOF. Затем внешний поток читает следующую запись и всё повторяется до выдачи из него EOF.

Данный метод доступа может быть использован для выполнения подзапросов в предикате `NOT EXIST`, а также других предикатов которые преобразуются в `NOT EXIST` (`ALL`). Кроме того, некоторые при некоторых условиях односторонние внешние запросы также могут быть преобразованы в него.

Кардинальность выходного потока оценить весьма сложно, но что можно сказать точно она никогда не превысит кардинальность внешнего потока. Предполагается, что условию связи не соответствует половина записей внешнего потока.

```
cardinality = cardinality(outer) * 0.5

cost = cardinality(outer) + cardinality(outer) * cost(inner first row)
```

Здесь `cost(inner first row)` — стоимость выборки первой записи соответствующей условию связи. Поскольку предикат `NOT EXISTS` может содержать сложный подзапрос, а не чтение из одной таблицы, то посчитать её не так просто. Так же достаточно очевидно, что для эффективной работы данного алгоритма оптимизатор для выполнения внутреннего подзапроса должен переключится на стратегию оптимизации `FIRST ROWS`.

Full Outer Join

Данный вид соединения не выполняется непосредственно, вместо этого он раскладывается в эквивалентную форму — сначала один поток соединяется с другим с помощью левого внешнего соединения (left outer join), затем потоки меняются местами и выполняется анти-соединение (anti-join), результаты обоих соединений объединяются.

Поскольку полного внешнего соединений потоки семантически равноценны, то оптимизатор может поменять их местами в зависимости от того что дешевле.

Пример 59. Nested Loop для полного внешнего соединения

```
SELECT *
FROM RDB$RELATIONS R
FULL JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (JOIN (RF NATURAL, R INDEX (RDB$INDEX_0)), JOIN (R NATURAL, RF INDEX
(RDB$INDEX_4)))
```

```
[cardinality=3375.0, cost=23980.4]
Select Expression
    [cardinality=3375.0, cost=23980.4]
    -> Full Outer Join
        [cardinality=3200.0, cost=22580.4]
        -> Nested Loop Join (outer)
            [cardinality=2428.0, cost=2428.0]
            -> Table "RDB$RELATION_FIELDS" as "RF" Full Scan
            [cardinality=1.3, cost=4.3]
            -> Filter
                [cardinality=1.3, cost=4.3]
                -> Table "RDB$RELATIONS" as "R" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_0" Unique Scan
        [cardinality=175.0, cost=1400.0]
        -> Nested Loop Join (outer)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=1.0, cost=4.0]
            -> Filter
                [cardinality=9.1, cost=12.1]
                -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
```

До Firebird до версии 3.0 не умел использовать индексы для полного внешнего соединения, что делало данный вид соединения крайне не эффективным. Выйти из ситуации можно было переписав запрос следующим образом.

```
SELECT *
FROM RDB$RELATION_FIELDS RF
LEFT JOIN RDB$RELATIONS R ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
UNION ALL
SELECT *
FROM RDB$RELATIONS R
LEFT JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
WHERE RF.RDB$RELATION_NAME IS NULL
```

Вторая часть запрос по сути является анти-соединением (anti-join). Сейчас оптимизатор делает такие преобразование за вас автоматически.

Соединение с хранимой процедурой

При использовании внутренних соединений с хранимой процедурой есть два варианта:

- входные параметры хранимой процедуры не зависят от других потоков;
- входные параметры хранимой процедуры зависят от входных потоков.

Эти случаи обрабатываются по разному. Если входные параметры процедуры не зависят от других потоков, то процедура автоматически становится ведущим потоком. Это позволяет выполнить её один раз вместо того, чтобы выполняться заново на каждой итерации (ведь индекс мы к ней применить не можем).

Пример 60. Соединение с некоррелированной хранимой процедурой

```
SELECT *
FROM SP_PEDIGREE(?) P
JOIN HORSE ON HORSE.CODE_HORSE = P.CODE_HORSE
```

```
PLAN JOIN (P NATURAL, HORSE INDEX (PK_HORSE))
```

```
[cardinality=1000.0, cost=4000.0]
Select Expression
    [cardinality=1000.0, cost=4000.0]
    -> Nested Loop Join (inner)
        [cardinality=1000.0, cost=????]
        -> Procedure "SP_PEDIGREE" as "P" Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
```

Если входные параметры процедуры зависят от других потоков, то оптимизатор определяет наличие этих зависимостей и сортирует потоки таким образом, чтобы процедура стала

ведомой от входных потоков.

Пример 61. Соединение с коррелированной хранимой процедурой

```
SELECT *
FROM
  HORSE
  CROSS JOIN SP_PEDIGREE(HORSE.CODE_HORSE) P
WHERE HORSE.CODE_COLOR = ?
```

```
PLAN JOIN (HORSE INDEX (FK_HORSE_COLOR), P NATURAL)
```

```
[cardinality=2377700.0, cost=2618.7]
Select Expression
    [cardinality=2377700.0, cost=2618.7]
    -> Nested Loop Join (inner)
        [cardinality=2377.7, cost=2618.7]
        -> Filter
            [cardinality=2377.7, cost=2618.7]
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_COLOR" Range Scan (full match)
        [cardinality=1000.0, cost=???]
        -> Procedure "SP_PEDIGREE" as "P" Scan
```

Firebird до версии 3.0 не умел определять зависимость входных параметров процедуры от прочих потоков. В этом случае процедура ставилась вперед, когда из таблицы еще не выбрана ни одна запись. Соответственно, на этапе исполнения возникала ошибка `isc_no_cur_rec` (“no current record for fetch operation”). Для обхода данной проблемы использовалось явное указание порядка соединения синтаксисом:

```
SELECT *
FROM
  HORSE
  LEFT JOIN SP_PEDIGREE(HORSE.CODE_HORSE) P ON 1=1
WHERE HORSE.CODE_COLOR = ?
```

В этом случае таблица всегда будет прочитана перед процедурой и все будет работать корректно.

Соединение с табличными выражениями

Табличными выражениями называются подзапросы, которые используются в качестве источников данных в основном запросе. Существует два типа табличных выражений:

- производные таблицы (derived table);
- общие табличные выражения (common table expression) или сокращённо CTE.

Производная таблица (derived table) — это табличное выражение, входящее в предложение FROM запроса. Производные таблицы представляют собой запрос в круглых скобках. Этом запросу можно дать псевдоним, после чего с табличным выражением можно обращаться как с обычной таблицей (использовать в качестве источника данных).

Пример 62. Запрос с использованием производной таблицы

```
SELECT
  F.RDB$RELATION_NAME
FROM (
    SELECT
      DISTINCT
      RF.RDB$RELATION_NAME,
      RF.RDB$SYSTEM_FLAG
    FROM
      RDB$RELATION_FIELDS RF
    WHERE RF.RDB$FIELD_NAME STARTING WITH 'RDB$'
  ) F
WHERE F.RDB$SYSTEM_FLAG = 1
```

Общие табличное выражение (common table expression) или сокращённо CTE — это именованное табличное выражение, объявленное в предложении WITH. С общим табличным выражением точно также как и с производной таблицей можно обращаться как с обычной таблицей (использовать в качестве источника данных). Общие табличные выражения которые объявлены ниже, могут использовать выше объявленные табличные выражения в своём теле. Общие табличное выражение делятся на рекурсивными и нерекурсивными. О рекурсивных CTE мы поговорим позже при рассмотрении соответствующего метода доступа. Нерекурсивные табличные выражения имеют следующий вид:

```
WITH
  <cte_1>, <cte_2>, ... <cte_N>
<main_select_expr>

<cte> ::= _cte_name_ [(<column_aliases>)] AS <select_expr>
```

Пример выше можно переписать с использованием CTE так:

Пример 63. Запрос с использованием CTE

```
WITH
  F AS (
    SELECT
      DISTINCT
      RF.RDB$RELATION_NAME,
      RF.RDB$SYSTEM_FLAG
    FROM
      RDB$RELATION_FIELDS RF
    WHERE RF.RDB$FIELD_NAME STARTING WITH 'RDB$'
  )
SELECT
  F.RDB$RELATION_NAME
FROM F
WHERE F.RDB$SYSTEM_FLAG = 1
```

Как будут вести себя соединения при использовании табличных выражений? Это будет зависеть от содержимого табличного выражения. В простейших случаях запрос с табличным выражением можно трансформировать в эквивалентный запрос без использования табличного выражения. Другими словами простейшие запросы раскрываются до базовых таблиц. В более сложных случаях табличное выражение всегда выбирается ведущим потоком во внутренних соединениях. Простые табличные выражения содержат только операции соединения потоков и условия их фильтрации. Добавление сортировки, DISTINCT, вычисление агрегатов, группировка, вычисление оконных функций, UNION и FIRST/SKIP счётчиков превращает табличное выражение в сложное.

Пример 64. Раскрытие CTE до базовых таблиц

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, FIELDS RF INDEX (RDB$INDEX_4), FIELDS F INDEX (RDB$INDEX_2))
```

```
[cardinality=2182.9, cost=4858.3]
Select Expression
    [cardinality=2182.9, cost=4858.3]
    -> Nested Loop Join (inner)
        [cardinality=233.7, cost=233.7]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.34, cost=12.34]
        -> Filter
            [cardinality=9.34, cost=12.34]
            -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
```

В данном случае CTE `FIELDS` была раскрыта до базовых таблиц и оптимизация происходила так, как будто мы просто присоединяли таблицы находящиеся внутри CTE к `RDB$RELATIONS R`. А теперь добавим внутрь CTE сортировку (она не изменяет кардинальность).

Пример 65. Сложные CTE, которые не могут быть раскрыты до базовых таблиц

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
    ORDER BY F.RDB$FIELD_TYPE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
```

```
PLAN JOIN (SORT (JOIN (FIELDS RF NATURAL, FIELDS F INDEX (RDB$INDEX_2))), R INDEX
(RDB$INDEX_0))
```

```
Select Expression
    [cardinality=2428.4, cost=21855.6]
    -> Nested Loop Join (inner)
        [cardinality=2428.4, cost=12142.0]
        -> Refetch
            [cardinality=2428.4, cost=12142.0]
            -> Sort (record length: 44, key length: 8)
                [cardinality=2428.4, cost=12142.0]
                -> Nested Loop Join (inner)
                    [cardinality=2428.4, cost=2428.4]
                    -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Full Scan
                    [cardinality=1.0, cost=4.0]
                    -> Filter
                        [cardinality=1.0, cost=4.0]
                        -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                            -> Bitmap
                                -> Index "RDB$INDEX_2" Unique Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

Здесь план изменился кардинально. Сначала был выполнен запрос внутри CTE, а только потом к нему присоединились остальные потоки.

В будущем данное поведение может измениться, поэтому не стоит рассчитывать, что сложные CTE всегда будут ведущим потоком во внутренних соединениях.

Если вас не устраивает данный порядок соединения вы всегда можете использовать указание порядка соединения через `LEFT JOIN` с последующей фильтрацией `IS NOT NULL`.

Пример 66. Сложные CTE, соединённые `LEFT JOIN`

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
    ORDER BY F.RDB$FIELD_TYPE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  LEFT JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
WHERE FIELDS.RDB$RELATION_NAME IS NOT NULL
```

```
PLAN JOIN (R NATURAL, SORT (JOIN (FIELDS RF INDEX (RDB$INDEX_4), FIELDS F INDEX
(RDB$INDEX_2))))
```

```
[cardinality=1091.45, cost=12082.3]
Select Expression
    [cardinality=1091.45, cost=12082.3]
    -> Filter
        [cardinality=2182.9, cost=12082.3]
        -> Nested Loop Join (outer)
            [cardinality=233.7, cost=233.7]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=9.34, cost=50.7]
            -> Refetch
                [cardinality=9.34, cost=50.7]
                -> Sort (record length: 44, key length: 8)
                    [cardinality=9.34, cost=50.7]
                    -> Nested Loop Join (inner)
                        [cardinality=9.34, cost=12.34]
                        -> Filter
                            [cardinality=9.34, cost=12.34]
                            -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                                -> Bitmap
                                    -> Index "RDB$INDEX_4" Range Scan (full match)
                        [cardinality=1.0, cost=4.0]
                        -> Filter
                            [cardinality=1.0, cost=4.0]
                            -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                                -> Bitmap
                                    -> Index "RDB$INDEX_2" Unique Scan
```

Помимо производных таблиц (derived table) не зависимых от внешних потоков, стандартом предусматривают производные таблицы зависимые от внешних потоков. Для возможности использования таких производных таблиц необходимо использовать ключевое слово `LATERAL` перед табличным выражением. Общие табличные выражения зависимые от внешних потоков не возможны.

Для LATERAL производных таблиц имеет смысл следующие типы соединений: `CROSS JOIN` и `LEFT JOIN`. Синтаксис допускает и другие виды соединения. Особенностью соединения с lateral derived table является то, что они могут быть выполнены только алгоритмом рекурсивного перебора. Другая особенность касается выполнения `CROSS JOIN`. Дело в том, что из-за того что LATERAL производная таблица зависит от внешних потоков, то она не может устанавливаться ведущим потоком в соединениях. Оптимизатор распознаёт наличие зависимостей и выбирает правильный порядок соединения.

Пример 67. CROSS JOIN LATERAL

```
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME
FROM
  RDB$RELATIONS R
  CROSS JOIN LATERAL (
    SELECT RF.RDB$FIELD_NAME
    FROM RDB$RELATION_FIELDS RF
    WHERE RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
    ORDER BY RF.RDB$FIELD_POSITION
    FETCH FIRST ROW ONLY
  ) FIELDS
```

```
PLAN JOIN (R NATURAL, SORT (FIELDS RF INDEX (RDB$INDEX_4)))
```

```
[cardinality=233.7, cost=3351.26]
Select Expression
    [cardinality=233.7, cost=3351.26]
    -> Nested Loop Join (inner)
        [cardinality=233.7, cost=233.7]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=1.0, cost=12.34]
        -> First N Records
            [cardinality=9.34, cost=12.34]
            -> Refetch
                [cardinality=9.34, cost=12.34]
                -> Sort (record length: 28, key length: 8)
                    [cardinality=9.34, cost=12.34]
                    -> Filter
                        [cardinality=9.34, cost=12.34]
                        -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                            -> Bitmap
                                -> Index "RDB$INDEX_4" Range Scan (full match)
```

Соединение с представлениями

Представление (view) — виртуальная (логическая) таблица, представляющая собой поименованный запрос (синоним к запросу), который будет подставлен как табличное выражение при использовании представления.

С точки зрения оптимизатора, соединения с представлениями ничем не отличается от соединения с табличными выражениями описанными выше. Единственное существенное различие заключается в том, что в отличие от производных таблиц, представления не могут быть коррелированными.

4.1.2. Хеширование (Hash join)

Соединение хешированием (Hash join) является альтернативным алгоритмом выполнения соединения. Соединение потоков с помощью алгоритма HASH JOIN стало доступно начиная с Firebird 3.0.

При соединение методом хеширования 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 пробуется 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.

Очевидно, что заменить сравнение ключей на сравнение хешей можно только при сравнении на строгое равенство ключей. Таким образом, соединение с условием связи вида (`ON MASTER.F > SLAVE.F`) не может быть выполнено алгоритмом Hash Join.

Тем не менее, у данного метода есть одно достоинство по сравнению с соединением вложенными циклами — допускается соединение хешированием на основе выражений. Например, соединение с условием связи вида (`ON MASTER.F + 1 = SLAVE.F + 2`) легко выполняется методом хеширования. Причина понятна: нет зависимости между потоками и, следовательно, нет требования использования индексов.

В процессе построения хеш таблицы могут возникать коллизии. Коллизией называется такая ситуация, когда для разных значений ключей получается одно и то же значение хеш функции. Поэтому после совпадения значения хеш функции, всегда выполняется повторное вычисление предиката связи. Коллизии сохраняются в виде списка, отсортированному по ключам, благодаря чему возможно производить бинарный поиск по цепочке коллизий.

В текущей реализации хеш таблица имеет фиксированный размер 1009 слотами, он не изменяется в зависимости от оценки кардинальности хешируемого потока. Также не происходит увеличения размера хеш таблицы и перехеширования при превышении длины цепочек коллизий. Это обозначает, что соединение хешированием эффективно только при относительно небольшой кардинальности ведомого потока. Если кардинальность хешируемого потока превышает 1009000 записей, то выбираются другие алгоритмы соединения (при наличии индексов NESTED LOOP JOIN, при отсутствии — MERGE JOIN).

В будущих версиях Firebird это может быть изменено.

Почему 1009000 записей? 1009000 / 1009 слотов = 1000 возможных коллизий, поиск по отсортированным коллизиям занимает log2(1000) = 10 шагов, что уже считается не эффективным.

Алгоритм соединения хешированием способен обработать нескольких условий связи, объединенных через AND. При этом хеш функция вычисляется для нескольких полей, входящих в условие связи (или нескольких связей, если соединяются более одного потока).

Однако, в случае объединения условий связи через OR, метод соединения хешированием не может быть применен. Здесь следует отметить, что хеш функция не всегда вычисляется для всех полей входящих в условие связи, она может быть вычислена только для части из них. Такое происходит если количество ключей связей превышает 8. В этом случае при совпадении значения хеш-функции, из хеш-таблицы будет возвращено больше записей, чем необходимо. Лишние записи будут отфильтрованы после вычисления всех предикатов в условии связи. В этом случае эффективность соединения хешированием падает. В Explain плане не отображается количество ключей, задействованных в хеш-функции (эта информация добавлена в Firebird 6.0).

Ещё одной особенностью соединения хешированием является то, что оно может быть выполнено при сравнение ключей на строгое равенство с учётом `NULL` значений. То есть допускается использование как `=`, так и `IS NOT DISTINCT FROM`. При этом при построении хеш таблицы между этими предикатами не делается никакой разницы, хотя очевидно, что если используется предикат `=`, то записи с ключом `NULL` можно не добавлять в хеш таблицу. Это приводит к увеличенному потреблению памяти, и снижению производительности соединения хешированием, если ключей со значением `NULL` много. Данный недостаток планируются исправить в будущих версиях Firebird (см. CORE-7769).

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 соединения вложенными циклами (NESTED LOOP) 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.

Стоимость соединения методом HASH JOIN состоит из следующих частей:

- стоимость извлечения записей из потока данных для хеширования;
- стоимость копирования записей в хеш таблицу (включая стоимость вычисления хеш функции);
- стоимость пробирования хеш таблицы и стоимость копирования для записей по которым хеши совпали.

Используется следующие формулы для оценки кардинальности и стоимости (см. InnerJoin.cpp):

```
// кардинальность выходного потока
cardinality = outerCardinality * innerCardinality * linkSelectivity

// кардинальность хеш таблицы, если не все поля условия связи являются ключами хеш-таблицы
hashCardinality = innerCardinality * outerCardinality * hashSelectivity

// если все поля условия связи являются ключами хеш-таблицы
hashCardinality = innerCardinality

cost =
    // hashed stream retrieval
    innerCost +
    // hashing cost
    hashCardinality * (COST_FACTOR_MEMCOPY + COST_FACTOR_HASHING) +
    // probing + copying cost
    outerCardinality * (COST_FACTOR_HASHING + innerCardinality * linkSelectivity * COST_FACTOR_MEMCOPY);

COST_FACTOR_MEMCOPY = 0.5

COST_FACTOR_HASHING = 0.5
```

Где

- `hashSelectivity` — селективность ключей связи, по которым считалась хеш функция;
- `linkSelectivity` — селективность условия связи;
- `innerCost` — стоимость извлечения записей для хешируемого потока;
- `innerCardinality` — кардинальность ведомого (хешируемого) потока;
- `outerCardinality` — кардинальность ведущего потока;
- `COST_FACTOR_MEMCOPY` - стоимость копирования записи из/в память;
- `COST_FACTOR_HASHING` - стоимость вычисления хеш функции.

В Legacy плане выполнения соединение хешированием отображается словом "HASH", за которым в скобках через запятую описываются входные потоки.

В Explain плане выполнения соединение хешированием отображается в виде дерева, корень которого подписан как “Hash Join” после чего в круглых скобках указан тип соединения. Уровнем ниже описываются соединяемые потоки. Хеширование ведомой таблицы отображается как “Record Buffer”, после чего в круглых скобках указана длина записи как (`record length: <length>`) (см. Буферизация записей).

Hash Join (inner)

Пример 68. Соединение методом Hash Join (inner)

```
SELECT *
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
```

```
PLAN HASH (P NATURAL, R NATURAL)
```

```
[cardinality=829.7, cost=1369.0]
Select Expression
    [cardinality=829.7, cost=1369.0]
    -> Filter
        [cardinality=829.7, cost=1369.0]
        -> Hash Join (inner)
            [cardinality=511.25, cost=511.25]
            -> Table "RDB$PAGES" as "P" Full Scan
            [cardinality=350.6, cost=701.2]
            -> Record Buffer (record length: 1345)
                [cardinality=350.6, cost=350.6]
                -> Table "RDB$RELATIONS" as "R" Full Scan
```

Hash Join (outer)

В настоящее время одностороннее внешнее соединение методом хеширования не реализовано в Firebird. Алгоритм его выполнения похож на выполнение внутреннего соединения, за одним исключением — хешируется всегда внутренний (ведомый или необязательный поток). Для каждой записи внешнего (обязательного) потока производится пробирование хеш-таблицы, если соответствие найдено, то две записи сливаются в одну и выдаются на выход. Если соответствие не найдено, то запись из внешнего потока с необходимым количеством NULL-значений и выдаем ее на выход.

Hash Join (semi)

Данный метод доступа может быть использован для выполнения подзапросов в предикатах `EXIST` и `IN (<select>)`, а также других предикатов которые преобразуются в `EXIST (ANY, SOME)`. Он доступен начиная с Firebird 5.0.1. По умолчанию эта возможность отключена, для её включения необходимо установить параметр конфигурации `SubQueryConversion` равным значению `true` в файле `firebird.conf` или `database.conf`.

Не любой подзапрос в `EXIST` может быть преобразован в semi-join. Если подзапрос содержит ограничители FETCH/FIRST/SKIP/ROWS, то преобразовать подзапрос в полу-соединение нельзя и он будет выполняться как обычный коррелированный подзапрос. Как и для внутреннего соединения для выполнения полу-соединения методом хеширования необходимо чтобы ключи сравнивались на строгое равенство, допускается использование выражений в условиях связи, условия связи могут быть объединены через `AND`.

Алгоритм полу-соединения методом хеширования следующий. В качестве ведомого потока выбирается поток подзапроса, который 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 опробируется в хеш-таблице. Если соответствие найдено, то запись из внешнего потока выдаётся на выход. Если вхождения ключа в хеш-таблицу нет, переходим к следующей записи ведущего потока и так далее.

Кардинальность выходного потока оценить весьма сложно, но что можно сказать точно — она никогда не превысит кардинальность внешнего потока. Предполагается, что условию связи соответствует половина записей внешнего потока.

В настоящее время стоимость полу-соединения методом хеширования не рассчитывается.

```
cardinality = cardinality(outer) * 0.5
```

Пример 69. Соединение методом Hash Join (semi)

```
SELECT *
FROM RDB$RELATIONS R
WHERE EXISTS (
  SELECT *
  FROM RDB$PAGES P
  WHERE P.RDB$RELATION_ID = R.RDB$RELATION_ID
    AND P.RDB$PAGE_SEQUENCE > 5
)
```

```
PLAN HASH (R NATURAL, P NATURAL)
```

```
[cardinality=175.3, cost=1548.4]
Select Expression
    [cardinality=175.3, cost=1548.4]
    -> Filter
        [cardinality=175.3, cost=1548.4]
        -> Hash Join (semi)
            [cardinality=350.6, cost=350.6]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=255.625, cost=1022.5]
            -> Record Buffer (record length: 33)
                [cardinality=255.625, cost=511.25]
                -> Filter
                    [cardinality=511.25, cost=511.25]
                    -> Table "RDB$PAGES" as "P" Full Scan
```

В данном случае условием связи является `P.RDB$RELATION_ID = R.RDB$RELATION_ID`. Данный запрос **как бы** преобразуется в следующий вид:

```
SELECT *
FROM
  RDB$RELATIONS R
  SEMI JOIN (
    SELECT *
    FROM RDB$PAGES
    WHERE RDB$PAGES.RDB$PAGE_SEQUENCE > 5
  ) P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
```

Конечно же такого синтаксиса в SQL не существует. Он продемонстрирован для лучшего понимания происходящего в терминах соединений.

Hash Join (anti)

Данный метод доступа может быть использован для выполнения подзапросов в предикате `NOT EXIST`, а также других предикатов которые преобразуются в `NOT EXIST (ALL)`. Кроме того, некоторые при некоторых условиях односторонние внешние запросы также могут быть преобразованы в него. В текущих версиях Firebird он не реализован.

Алгоритм анти-соединения методом хеширования следующий. В качестве ведомого потока выбирается поток подзапроса, который 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 опробируется в хеш-таблице. Если соответствие не найдено, то запись из внешнего потока выдаётся на выход. Если соответствие найдено, то переходим к следующей записи ведущего потока и так далее.

4.1.3. Однопроходное слияние (Merge)

Слияние является альтернативным алгоритмом реализации соединения. В этом случае входные потоки полностью независимы. Для корректного выполнения операции потоки должны быть предварительно упорядочены по ключу соединения, после чего строится бинарное дерево слияния. Далее при фетче читаются записи из обоих потоков и их ключи связи сравниваются на равенство. При совпадении ключей запись возвращается на выход. Затем читаются новые записи из входных потоков и процесс повторяется. Слияние всегда выполняется в один проход, то есть каждый входной поток читается только один раз. Это возможно благодаря упорядоченному расположению ключей связи во входных потоках.

Однако, данный алгоритм не может быть использован для всех видов соединения. Как было отмечено выше, требуется сравнение ключей на строгое равенство. Таким образом, соединение с условием связи вида `(ON MASTER.F > SLAVE.F)` не может быть выполнено посредством однопроходного слияния. Справедливости ради стоит отметить, что соединения такого вида не могут быть эффективно выполнены никаким способом, так как даже в случае использования алгоритма соединения вложенными циклами на каждой итерации будут повторяющиеся чтения из внутреннего потока.

В некоторых СУБД, например Oracle, возможно соединение методом слияния не только для строго равенства, то есть возможны соединения с условиями связи вида (`ON MASTER.F > SLAVE.F`).

Тем не менее, у данного метода есть одно достоинство по сравнению алгоритмом соединения вложенными циклами — допускается слияние на основе выражений. Например, соединение с условием связи вида (`ON MASTER.F + 1 = SLAVE.F + 2`) легко выполняется методом слияния. Причина понятна: нет зависимости между потоками и, следовательно, нет требования использования индексов.

Внимательный читатель мог заметить, что в описании алгоритма слияния не оговорен режим работы метода в случае зависимости потоков (одностороннее внешнее соединение). Ответ прост: для таких соединений данный алгоритм не поддерживается. Кроме того, он также не поддерживается и для полных внешних соединений, полу-соединений и анти-соединений.

Поддержка алгоритмом внешних соединений планируется в следующих версиях сервера.

Данный алгоритм способен обработать нескольких условий связи, объединенных через AND. При этом входные потоки просто сортируются по нескольким полям. Однако, в случае объединения условий связи через OR, метод слияния не может быть применен.

Основным недостатком данного алгоритма является требование сортировки обоих потоков по ключам соединения. Если потоки уже отсортированы, то данный алгоритм соединения является самым дешёвым среди прочих (NESTED LOOP, HASH JOIN). Однако обычно соединяемые потоки не отсортированы по ключам соединения, а потому их приходится сортировать, что резко повышает стоимость выполнения соединения данным методом. К сожалению в настоящий момент оптимизатор не умеет определять, что потоки уже отсортированы по ключам соединения, а потому данный алгоритм может быть не задействован в тех случаях. когда он был бы действительно более дешёвым.

Данный недостаток планируется исправить в следующих версиях сервера.

До Firebird 3.0 соединения слиянием только в случае невозможности или неоптимальности использования алгоритма соединения вложенными циклами, то есть в первую очередь при отсутствии индексов по условию связи или их неприменимости, а также при отсутствии зависимости между входными потоками. Начиная с Firebird 3.0 алгоритм соединения слиянием был отключен, а вместо него всегда использовалось соединение хешированием. Начиная с Firebird 5.0 соединение слиянием снова стало доступно. Оно используется, только в тех случаях когда соединяемые потоки слишком велики и соединение хешированием становится неэффективным (кардинальность всех соединяемых потоков больше 1009000 записей см. Хеширование (Hash join)), а также при отсутствии индексов по условию связи, из-за чего неоптимальным будет и алгоритм соединения вложенными циклами.

Используется следующие формулы для оценки стоимости и кардинальности соединения:

```
cost = cost_1 + cost_2

cardinality = cardinality_1 * cardinality_2 * selectivity(link)
```

Где cost_1, cost_2 — это стоимость извлечения данных из отсортированных входных потоков. А поскольку стоимость внешней сортировки не рассчитывается, то и стоимость соединения слиянием невозможно оценить корректно.

В Legacy плане выполнения соединение слиянием отображается словом "MERGE", за которым в скобках через запятую описываются входные потоки.

В Explain плане выполнения соединение слиянием отображается в виде дерева, корень которого подписан как “Merge Join” после чего в круглых скобках указан тип соединения. Уровнем ниже описываются соединяемые потоки, которые сортируются внешней сортировкой.

Пример 70. Соединение методом Merge Join (inner)

```
SELECT *
FROM
  WORD_DICTIONARY WD1
  JOIN WORD_DICTIONARY WD2 ON WD1.PARAMS = WD2.PARAMS
```

```
PLAN MERGE (SORT (WD1 NATURAL), SORT (WD2 NATURAL))
```

```
[cardinality=22307569204, cost=???]
Select Expression
    [cardinality=22307569204, cost=???]
    -> Filter
        [cardinality=22307569204, cost=???]
        -> Merge Join (inner)
            [cardinality=4723000, cost=???]
            -> Sort (record length: 710, key length: 328)
                [cardinality=4723000, cost=???]
                -> Table "WORD_DICTIONARY" as "WD1" Full Scan
            [cardinality=4723000, cost=???]
            -> Sort (record length: 710, key length: 328)
                [cardinality=4723000, cost=4723000]
                -> Table "WORD_DICTIONARY" as "WD2" Full Scan
```

Последний пример искусственный, потому что заставить оптимизатор производить соединение методом слияния довольно тяжело.

4.2. Объединения (Union)

Название метода доступа говорит само за себя. Этот метод доступа выполняет операцию SQL-объединения (union). Существует два режима выполнения этой операции: `ALL` и `DISTINCT`. В первом случае реализация тривиальна: данный метод просто читает первый входной поток и выдает его на выход, по получении из него EOF начинает читать второй входной поток и так далее. В случае же `DISTINCT` требуется устранить полные дубликаты записей, присутствующие в результате объединения. Для этого на выходе метода объединения размещается фильтр сортировки, работающий в “усекающем” режиме по всем полям.

Стоимость выполнения объединения равна суммарной стоимости всех входных потоков, кардинальность также получается суммированием. В режиме `DISTINCT` результирующая кардинальность делится на 10.

В Legacy плане выполнения объединение отображается отдельными планами на каждый из входных потоков.

В Explain плане выполнения объединение отображается в виде дерева, корень которого подписан как “Union”. Уровнем ниже описываются объединяемые потоки.

Пример 71. Объединение потоков в режиме ALL

```
SELECT RDB$RELATION_ID
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 1
UNION ALL
SELECT RDB$PROCEDURE_ID
FROM RDB$PROCEDURES
WHERE RDB$SYSTEM_FLAG = 1
```

```
PLAN (RDB$RELATIONS NATURAL, RDB$PROCEDURES NATURAL)
```

```
[cardinality=90.33, cost=873.3]
Select Expression
    [cardinality=90.33, cost=873.3]
    -> Union
        [cardinality=35.06, cost=350.6]
        -> Filter
            [cardinality=350.6, cost=350.6]
            -> Table "RDB$RELATIONS" Full Scan
        [cardinality=55.27, cost=552.7]
        -> Filter
            [cardinality=552.7, cost=552.7]
            -> Table "RDB$PROCEDURES" Full Scan
```

Пример 72. Объединение потоков в режиме DISTINCT

```
SELECT RDB$RELATION_ID
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 1
UNION
SELECT RDB$PROCEDURE_ID
FROM RDB$PROCEDURES
WHERE RDB$SYSTEM_FLAG = 1
```

```
PLAN SORT (RDB$RELATIONS NATURAL, RDB$PROCEDURES NATURAL)
```

```
[cardinality=9.033, cost=873.3]
Select Expression
    [cardinality=9.033, cost=873.3]
    -> Unique Sort (record length: 44, key length: 8)
        [cardinality=90.33, cost=873.3]
        -> Union
            [cardinality=35.06, cost=350.6]
            -> Filter
                [cardinality=350.6, cost=350.6]
                -> Table "RDB$RELATIONS" Full Scan
            [cardinality=55.27, cost=552.7]
            -> Filter
                [cardinality=552.7, cost=552.7]
                -> Table "RDB$PROCEDURES" Full Scan
```

4.2.1. Материализация недетерминированных выражений

Табличные выражения в Firebird имеют одну неприятную особенность, а именно при обращении к столбцам табличного выражения, которые ссылаются на выражения, эти выражения вычисляются каждый раз при упоминании столбца табличного выражения. Это легко продемонстрировать при использовании недетерминированных функций.

Попробуйте выполнить следующий запрос:

```
WITH
  T AS (
    SELECT GEN_UUID() AS UUID
    FROM RDB$DATABASE
  )
SELECT
  UUID_TO_CHAR(UUID) AS ID1,
  UUID_TO_CHAR(UUID) AS ID2
FROM T
```

Результатом будет что-то вроде

```
ID1                                  ID2
===================================  ===================================
3C8CA94D-9D05-4A49-8788-1D9024193C4E  D5E86F5C-BF48-4AA3-AB6D-3EF9C9B58B52
```

Хотя вы ожидали, что значения в столбцах будет одинаковым.

Это также проявляется и при сортировке по ссылке на столбец. Например:

```
SELECT GEN_ID(SEQ_SOME, 1) AS ID
FROM RDB$DATABASE
ORDER BY 1
```

Последовательность увеличиться не 1, как вы ожидали, а на 2.

Существует один трюк, который позволяет “материализовать” результаты вычислений выражений в табличных выражениях. Для этого достаточно сделать UNION с запросом возвращающим 0 записей, естественно общее количество столбцов обоих запросов должно быть одинаковым. Попробуем переписать первый запрос следующим образом:

```
WITH
  T AS (
    SELECT GEN_UUID() AS UUID
    FROM RDB$DATABASE
    UNION ALL
    SELECT NULL FROM RDB$DATABASE WHERE FALSE
  )
SELECT
  UUID_TO_CHAR(UUID) AS ID1,
  UUID_TO_CHAR(UUID) AS ID2
FROM T
```

Теперь значения ID1 и ID2 будут одинаковыми.

```
ID1                                  ID2
===================================  ===================================
9599C281-DD96-4CC7-9C05-6259CCAB467F  9599C281-DD96-4CC7-9C05-6259CCAB467F
```

4.2.2. Материализация подзапросов

А что будет, если в качестве столбца табличного выражения используется подзапрос? Подзапрос может быть достаточно тяжёлым чтобы выполнять его повторно. К счастью в этом случае оптимизатор делает всю работу за нас. Попробуйте выполнить следующий запрос:

```
WITH T
  AS (
    SELECT
      (SELECT COUNT(*)
       FROM RDB$RELATION_FIELDS RF
       WHERE RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME) AS CNT
    FROM RDB$RELATIONS R
  )
SELECT
  SUM(T.CNT) AS CNT,
  SUM(T.CNT) * 1e0 / COUNT(*) AS AVG_CNT
FROM T
```

Несмотря на повторное упоминание T.CNT подзапрос будет выполнен однократно для каждой записи из RDB$RELATIONS.

В Explain плане вы увидите следующее:

```
Sub-query
    -> Singularity Check
        -> Aggregate
            -> Filter
                -> Table "RDB$RELATION_FIELDS" as "T RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Materialize
            -> Table "RDB$RELATIONS" as "T R" Full Scan
```

Здесь “Materialize” обозначает материализацию результатов подзапроса для каждой записи из RDB$RELATIONS. Это не какой-то отдельный метод доступа, а внутреннее объединение (UNION) внутри табличного выражение с пустым множеством. То есть произошло тоже самое, что описано выше про материализацию недерминированных выражений.

4.3. Рекурсия

Данный метод доступа применяется для выполнения рекурсивных табличных выражений (рекурсивных CTE). Тело рекурсивного CTE представляет собой запрос с UNION ALL, который объединяет один или несколько подзапросов называемых закреплёнными элементами. Кроме закреплённых элементов есть один или несколько рекурсивных подзапросов, называемых рекурсивными элементами. Эти рекурсивные подзапросы ссылаются на сам рекурсивный CTE. Получается, у нас есть один или несколько закреплённых подзапросов и один или несколько рекурсивных подзапросов, объединенных UNION ALL.

Суть рекурсии проста — сначала выполняются и объединяются нерекурсивные подзапросы, и для каждой записи нерекурсивной части набор данных дополняется записями из рекурсивной части, которая может использовать результат полученный на предыдущем шаге. Рекурсия прекращается, когда все рекурсивные части не возвращают ни одной

записи.

При выполнении рекурсивных CTE есть ещё одна особенность — никакие предикаты из внешнего запроса не могут быть протолкнуты внутрь CTE.

Кардинальность и стоимость рекурсии оценить весьма сложно. Оптимизатор считает, что кардинальность равна сумме кардинальностей нерекурсивных частей, которая умножается на кардинальность добавленную соединениями (joins) в рекурсивной части. Стоимость суммируется из стоимости выборки всех записей рекурсивных и нерекурсивных частей. Стоит отметить, что подобная оценка иногда далека от истины.

В Legacy плане выполнения рекурсия не отображается.

В Explain плане выполнения рекурсия отображается в виде дерева, корень которого подписан как “Recursion”. Уровнем ниже описываются объединяемые потоки. Обращение к самому рекурсивному CTE внутри CTE не описывается как отдельный поток данных.

Пример 73. Простейший рекурсивный запрос

```
WITH RECURSIVE
  R(N) AS (
    SELECT 1 FROM RDB$DATABASE
    UNION ALL
    SELECT R.N + 1 FROM R
  )
SELECT N FROM R
```

```
PLAN (R RDB$DATABASE NATURAL, )
```

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Recursion
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" as "R RDB$DATABASE" Full Scan
```

У запроса описанного выше есть один существенный недостаток. Он делается “бесконечную” рекурсию. Но это в теории, а на практике Firebird ограничивает глубину рекурсии значением 1024. Давайте исправим это.

Пример 74. Рекурсивный запрос с ограничением глубины

```
WITH RECURSIVE
  R(N) AS (
    SELECT 1 FROM RDB$DATABASE
    UNION ALL
    SELECT R.N + 1 FROM R WHERE R.N < 10
  )
SELECT N FROM R
```

```
PLAN (R RDB$DATABASE NATURAL, )
```

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Recursion
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" as "R RDB$DATABASE" Full Scan
        [cardinality=1.0, cost=1.0]
        -> Filter (preliminary)
```

Здесь в explain плане становится видно рекурсивную часть, которая фильтруется, но сам поток не показан. Поскольку рекурсивный член не соединяется с другими источниками данных, кардинальность и стоимость берётся и из нерекурсивной части. Здесь оценка кардинальности равна 1, хотя на самом деле будет возвращено 10 записей.

Рекурсивная часть может быть более сложная и содержать соединения (только внутренние) с другими источниками данных.

Пример 75. Рекурсивный запрос с соединением по неуникальному индексу

```
WITH RECURSIVE
  R AS (
    SELECT
      DEPT_NO,
      DEPARTMENT,
      HEAD_DEPT
    FROM DEPARTMENT
    WHERE HEAD_DEPT IS NULL
    UNION ALL
    SELECT
      DEPARTMENT.DEPT_NO,
      DEPARTMENT.DEPARTMENT,
      DEPARTMENT.HEAD_DEPT
    FROM R JOIN DEPARTMENT ON DEPARTMENT.HEAD_DEPT = R.DEPT_NO
  )
SELECT * FROM R
```

```
PLAN (R DEPARTMENT INDEX (RDB$FOREIGN6), R DEPARTMENT INDEX (RDB$FOREIGN6))
```

```
[cardinality=6.890625, cost=20.390625]
Select Expression
    [cardinality=6.890625, cost=20.390625]
    -> Recursion
        [cardinality=2.625, cost=5.625]
        -> Filter
            [cardinality=2.625, cost=5.625]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$FOREIGN6" Range Scan (full match)
        [cardinality=2.625, cost=5.625]
        -> Filter
            [cardinality=2.625, cost=5.625]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$FOREIGN6" Range Scan (full match)
```

Здесь в рекурсивной части запроса производится соединение с таблицей DEPARTMENT по неуникальному индексу, именно поэтому кардинальности перемножаются (для каждой записи из нерекурсивного члена присоединяется 2.625 записей рекурсивного члена). На самом деле будет выбрана 21 запись.

Попробуем развернуть рекурсию.

Пример 76. Рекурсивный запрос с соединением по уникальному индексу

```
WITH RECURSIVE
  R AS (
    SELECT
      DEPT_NO,
      DEPARTMENT,
      HEAD_DEPT
    FROM DEPARTMENT
    WHERE DEPT_NO = '672'
    UNION ALL
    SELECT
      DEPARTMENT.DEPT_NO,
      DEPARTMENT.DEPARTMENT,
      DEPARTMENT.HEAD_DEPT
    FROM R JOIN DEPARTMENT ON DEPARTMENT.DEPT_NO = R.HEAD_DEPT
  )
SELECT * FROM R
```

```
PLAN (R DEPARTMENT INDEX (RDB$PRIMARY5), R DEPARTMENT INDEX (RDB$PRIMARY5))
```

```
[cardinality=1.0, cost=8.0]
Select Expression
    [cardinality=1.0, cost=8.0]
    -> Recursion
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY5" Unique Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY5" Unique Scan
```

Здесь в рекурсивной части запроса производится соединение с таблицей DEPARTMENT по уникальному индексу, поэтому общая кардинальность не увеличивается (для 1 записи из нерекурсивного члена присоединяется 1 запись рекурсивного члена). На практике будет возвращено 4 записи.

Глава 5. Стратегии оптимизации

В процессе рассмотрения методов доступа Firebird мы выяснили, что источники данных могут быть конвейерными и буферизованными. Конвейерный источник данных выдает записи в процессе чтения своих входных потоков, в то время как буферизованный источник сначала должен прочитать все записи из своих входных потоков и только потом сможет выдать первую запись на свой выход.

Большинство методов доступа являются конвейерными. К буферизованным методам доступа относится внешняя сортировка (SORT) и буферизация записей (Record Buffer). Кроме того некоторые методы доступа требуют использования буферизованных источников данных. Для соединения хешированием (Hash Join) требуется буферизация при построении хеш таблицы. Соединение слиянием (Merge Join) требует упорядоченности входных потоков по ключам соединения, для чего используется внешняя сортировка. Для вычисления оконных функций (Window) так же требуется буферизация.

В Firebird оптимизатор может работать в двух режимах:

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

В большинстве случаев требуется стратегия оптимизации `ALL ROWS`. Однако если у вас есть приложения с сетками данных, в которых отображаются только первые строки результата, а остальные извлекаются по мере необходимости, то стратегия `FIRST ROWS` может быть более предпочтительной, поскольку сокращается время отклика.

При использовании стратегии `FIRST ROWS` оптимизатор старается заменить буферизованные методы доступа на альтернативные конвейерные, если это возможно и дешевле с точки зрения стоимости извлечения первой строки. То есть Hash/Merge join заменяются на Nested Loop Join, а внешняя сортировка (Sort) на навигацию по индексу.

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`.

Стратегию оптимизации можно переопределить непосредственно в тексте SQL запроса используя предложение `OPTIMIZE FOR`. Кроме того, использование в запросе first/skip счётчиков неявно переключает стратегию оптимизации в `FIRST ROWS`. Сравним планы запроса, которые используют разные стратегии оптимизации.

Пример 77. Запрос со стратегией оптимизации ALL ROWS

```
SELECT
  R.RDB$RELATION_NAME,
  P.RDB$PAGE_SEQUENCE
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
OPTIMIZE FOR ALL ROWS
```

```
PLAN HASH (P NATURAL, R NATURAL)
```

```
Select Expression
    -> Filter
        -> Hash Join (inner)
            -> Table "RDB$PAGES" as "P" Full Scan
            -> Record Buffer (record length: 281)
                -> Table "RDB$RELATIONS" as "R" Full Scan
```

Пример 78. Запрос со стратегией оптимизации FIRST ROWS

```
SELECT
  R.RDB$RELATION_NAME,
  P.RDB$PAGE_SEQUENCE
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
OPTIMIZE FOR FIRST ROWS
```

```
PLAN JOIN (P NATURAL, R INDEX (RDB$INDEX_1))
```

```
Select Expression
    -> Nested Loop Join (inner)
        -> Table "RDB$PAGES" as "P" Full Scan
        -> Filter
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_1" Range Scan (full match)
```

Глава 6. Заключение

Выше мы рассмотрели все виды операций, которые участвуют в выполнении SQL-запросов. Для каждого алгоритма приведено подробное описание и примеры, включая детализацию дерева выполнения запроса (Explain план). Имеет смысл отметить, что Firebird реализует не так и много методов доступа по сравнению с конкурентами, однако зачастую это объясняется либо особенностями архитектуры, либо сравнительно большей эффективностью имеющихся методов.