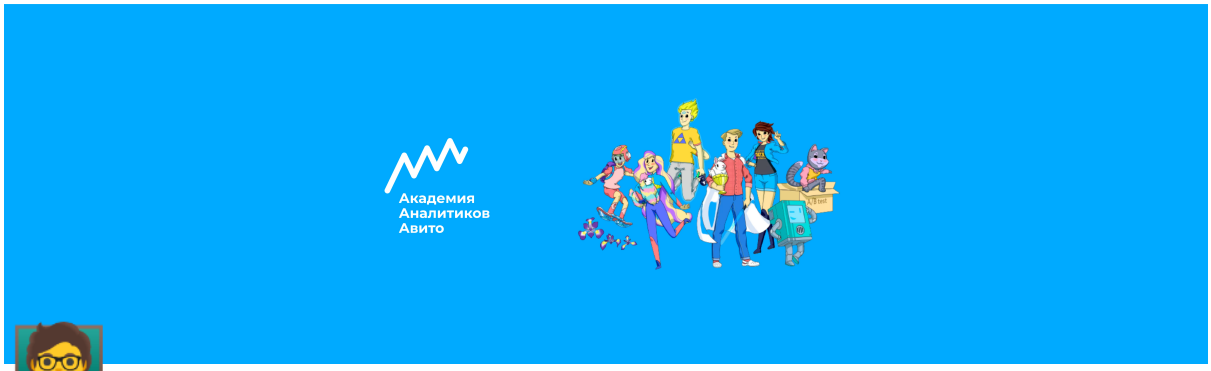

Week 11 - Внутреннее устройство СУБД и оптимизации запросов

Как база данных понимает, что делать?

Когда вы пишите запрос в базу данных, вы говорите ей, что вы хотите в итоге получить. При этом вы не даете никаких инструкций, как получить результат, иначе, вы пользуетесь декларативным интерфейсом. Такой интерфейс удобен с точки зрения пользователя, но с точки зрения разработчика базы постоянно возникают проблемы: запросы надо выполнять эффективно и не всегда понятно, как это делать правильно.
Рассмотрим несколько запросов:

```
select client_id, name
from clients
```

Здесь вы просите вытащить две колонки из таблицы clients . База предпримет следующие действия:

1. Найти в индексе данные о наличии таблицы и требуемых колонок
2. Вытащить из индекса информацию о местонахождении файлов, относящихся к таблице
3. Прочитать эти файлы. При наличии сжатия — разжать
4. Вытащить только нужные колонки среди этого массива информации.
5. Вывести данные пользователю в удобном виде (числа, даты и прочие структуры нужно привести в читаемый вид).

То, что я здесь описал называется планом запроса. Обычно такой план выглядит очень страшно и длинно, указывая исчерпывающую информацию о том, что конкретно будет делать база при том или ином запросе.

Посмотрим на план для запроса выше с помощью запроса explain :

```
explain
select client_id, name
from clients
```

В ответе базы мы получим что-то очень похожее на вот такой вывод:

```
                     QUERY PLAN
Seq Scan on clients (cost=0.00..18.50 rows=850 width=36)
```

Очень заметно, что тут нет ничего про "прочитать файлики" и прочее — здесь нам показывается только то, что база будет делать конкретно с данными, а не каждый ее шаг.

Для конкретно этого плана мы видим одну строку (или иначе один шаг пайплайна) — Seq Scan (от sequential scan —последовательное чтение). Это значит, что база прочитает всю таблицу строка за строкой последовательно. Числа в скобках означают следующее:

1. cost=0.00..18.50 — "время", необходимое для старта работы запроса (startup cost), и время, необходимое для работы всего запроса (total cost). Здесь время взято в кавычки так как это время является некоторой условной единицей. Чем меньше это значение, тем лучше. По умолчанию постгрес будет вычислять cost в виде числа запросов к диску для прочтения всех данных. (Это не совсем правда, но в качестве приближения можно считать и так. Интересующиеся приглашаются к прочтению документации).
2. rows=850 — оценка числа строк, которые вернет запрос. Данная оценка не является точной, она лишь служит для грубого понимания тяжести запроса.
3. width=36 — средняя длина одной строки в байтах. Вычисляется достаточно точно, на этот параметр можно полагаться.

Научившись читать простейшие запросы, давайте попробуем что-нибудь более сложное:

```
explain
select *
from clients
where client_id > 100
```

План получится следующий:

```
                        QUERY PLAN
Seq Scan on clients (cost=0.00..20.62 rows=283 width=68)
Filter: (client_id > 100)
```

На самом деле, план запроса это дерево, где самый верхний левый узел (в текстовом представлении) является результатом запроса, а поддеревья это какие-то части запроса. Здесь мы видим, что у нашего seq scan появился атрибут (узел) фильтрации. Нам подсказывают, по какому полю и с каким условием произойдет фильтр. Заметьте, как изменились числа внутри скобок. Total cost вырос, что не удивительно — нам надо в дополнение к чтению данных еще

выполнить операцию сравнения. Число строк уменьшилось — что естественно при фильтрации. Также увеличилась ширина данных — мы теперь вычитываем все колонки.

Посмотрим как выглядит join:

```
explain
select *
from clients l
join clients r
on l.client_id = r.client_id;
```

```
                    QUERY PLAN
Hash Join  (cost=11.57..22.47 rows=70 width=2096)
  Hash Cond: (l.client_id = r.client_id)
  ->  Seq Scan on clients l  (cost=0.00..10.70 rows=70 width=1048)
  ->  Hash  (cost=10.70..10.70 rows=70 width=1048)
        ->  Seq Scan on clients r  (cost=0.00..10.70 rows=70 width=1048)
```

Что мы можем узнать полезного из этого плана запроса?

- во-первых, будет использован Hash Join (подробнее об этом в секции "алгоритмы"),
- во-вторых, мы видим стоимость запроса. Она сильно (в 10 раз) больше простого чтения — намек на сложность джойна.

Если вы почувствовали, что что-то тут не так — вы правы: таблица сейчас пуста. База об этом не знает, она только знает, что таблица занимает минимальное количество места (0 байт на данные), при этом обращение за метаданными тоже что-то стоит, поэтому у нас ненулевой кост. Попросим базу собрать статистику по таблице с помощью запроса analyze:

```
analyze clients
```

После этого планировщик выдает правильные значения коста:

```
postgres=# explain
select *
from clients
where client_id > 100
;
                    QUERY PLAN
Seq Scan on clients  (cost=0.00..10.88 rows=23 width=1048)
  Filter: (client_id > 100)
(2 rows)
```

Когда СУБД строит план запроса - она смотрит на статистику по таблицам (количество строк, уникальность полей..). Статистика может устареть и план будет неоптимальный. Например, кто-то загрузит в таблицу несколько лет истории. Поэтому помните про статистику и собриайте ее чтобы помочь СУБД построить правильный план.

Давайте теперь создадим таблицу с заказами, с внешним ключом client_id:

```
create table orders (client_id int, order_id int, amount int);
```

И промежуточную таблицу с идентификаторами клиентов:

```
create table client_identifiers (pos int, client_id int);
```

Наполним таблицу с идентификаторами:

```
insert into client_identifiers
select generate_series(1, 10000000) id,
       floor(random() * 1000000000) client_id;
analyze client_identifiers;
```

Теперь посмотрим более близко на джойны. Давайте вставим в каждую таблицу по 1000 идентификаторов, при этом пересечение идентификаторов будет 100%:

```
insert into clients (client_id, name)
select client_id, md5(random()::varchar) as name
from client_identifiers;
```

```
insert into orders (client_id, order_id)
select client_id, floor(random() * 1000000) as order_id
from client_identifiers;
```

Давайте попробуем поджойнить эти таблицы:

```
explain
select *
from clients
  join orders
    using(client_id)
```

Получим следующий план:

```
                      QUERY PLAN
Hash Join (cost=3182.00..9224.19 rows=100019 width=77)
  Hash Cond: (clients.client_id = orders.client_id)
  -> Seq Scan on clients (cost=0.00..1834.00 rows=100000 width=69)
  -> Hash (cost=1443.00..1443.00 rows=100000 width=12)
    -> Seq Scan on orders (cost=0.00..1443.00 rows=100000 width=12)
```

Давайте добавим еще 10 миллионов записей в таблицу заказов:

```
insert into client_identifiers
select generate_series(1, 10000000) id,
    floor(random() * 1000000000) client_id;

analyze client_identifiers;
```

```
insert into orders (client_id, order_id)
select client_id, floor(random() * 1000000) as order_id
from client_identifiers;
```

И посмотрим на план:

```
                        QUERY PLAN
Gather (cost=5256.00..162157.23 rows=102263 width=77)
  Workers Planned: 2
  -> Hash Join (cost=4256.00..150930.93 rows=42610 width=77)
      Hash Cond: (orders.client_id = clients.client_id)
      -> Parallel Seq Scan on orders (cost=0.00..87633.24 rows=4250024 width=12)
      -> Hash (cost=1834.00..1834.00 rows=100000 width=69)
        -> Seq Scan on clients (cost=0.00..1834.00 rows=100000 width=69)
JIT:
  Functions: 9
  Options: Inlining false, Optimization false, Expressions true, Deforming true
```

База все еще хочет сделать Hash Join, но при этом она решила распараллелить работу. Поменяем таблицы местами:

```
explain
select *
from orders
  join clients
    using(client_id)
```

```
                       QUERY PLAN
Gather (cost=5256.00..162157.23 rows=102263 width=77)
  Workers Planned: 2
  -> Hash Join (cost=4256.00..150930.93 rows=42610 width=77)
    Hash Cond: (orders.client_id = clients.client_id)
    -> Parallel Seq Scan on orders (cost=0.00..87633.24 rows=4250024 width=12)
    -> Hash (cost=1834.00..1834.00 rows=100000 width=69)
      -> Seq Scan on clients (cost=0.00..1834.00 rows=100000 width=69)
JIT:
  Functions: 10
  Options: Inlining false, Optimization false, Expressions true, Deforming true
```

База умная, она уже вынесла бОльшую таблицу для параллельного чтения. Давайте теперь попробуем отсортировать данные внутри джойна:

```
explain
select *
from (select * from orders order by client_id) orders
join (select * from clients order by client_id) clients
using(client_id)
```

```
                           QUERY PLAN
Merge Join (cost=716052.32..1816828.37 rows=102263 width=77)
  Merge Cond: (orders.client_id = clients.client_id)
  -> Gather Merge (cost=701811.50..1693553.00 rows=8500048 width=12)
    Workers Planned: 2
    -> Sort (cost=700811.47..711436.53 rows=4250024 width=12)
      Sort Key: orders.client_id
      -> Parallel Seq Scan on orders (cost=0.00..87633.24 rows=4250024 width=12)
```

```
   -> Materialize (cost=14240.82..15740.82 rows=100000 width=69)
     -> Sort (cost=14240.82..14490.82 rows=100000 width=69)
       Sort Key: clients.client_id
       -> Seq Scan on clients (cost=0.00..1834.00 rows=100000 width=69)
JIT:
  Functions: 5
  Options: Inlining true, Optimization true, Expressions true, Deforming true
```

Стоило нам отсортировать данные по ключу соединения, мы получили Merge Join.

Большой совет: постоянно смотреть за тем, что будет делать база. Внутри каждой базы есть оптимизатор запросов, который смотрит на то, что ему скормили и пытается построить самый оптимальный план. Иногда это происходит с ошибками, например, база не знает примерный размер таблицы в соединении, и поэтому использует nested loop вместо hash join. Решение — выполнение запроса analyze над нужной таблицей. В общем случае оптимизатор — ваш друг, но за ним тоже нужно следить.

Слово analyze также можно использовать в запросе explain:

```
explain analyze
select *
from (select * from orders order by client_id) orders
join (select * from clients order by client_id) clients
using(client_id)
```

Этот запрос будет выполняться сильно медленне — он форсит базу данных собрать максимально точную статистику для отображения, поэтому базы обычно просто исполняют запрос, наблюдая за характеристиками каждого выражения. Для больших таблиц такой запрос может исполняться очень и очень долго.

```
                        QUERY PLAN
Merge Join (cost=10.64..14.89 rows=100 width=77) (actual time=3.070..8.189 rows=100 loops=1)
  Merge Cond: (orders.client_id = clients.client_id)
  -> Sort (cost=5.32..5.57 rows=100 width=12) (actual time=1.579..2.308 rows=100 loops=1)
    Sort Key: orders.client_id
    Sort Method: quicksort Memory: 29kB
    -> Seq Scan on orders (cost=0.00..2.00 rows=100 width=12) (actual time=0.009..0.757 rows=100 loops=1)
  -> Materialize (cost=5.32..6.82 rows=100 width=69) (actual time=1.465..3.585 rows=100 loops=1)
    -> Sort (cost=5.32..5.57 rows=100 width=69) (actual time=1.453..2.207 rows=100 loops=1)
      Sort Key: clients.client_id
      Sort Method: quicksort Memory: 32kB
      -> Seq Scan on clients (cost=0.00..2.00 rows=100 width=69) (actual time=0.013..0.710 rows=100 loops=1)

Planning Time: 0.195 ms
Execution Time: 8.893 ms
(13 rows)
```

Здесь вместе с базовой информацией также показывается время, требуемое на планирование запроса, время на исполнение и более точные оценки времени/количества строк.

А что там с алгоритмами?

Есть три основных алгоритма расчета соединения двух таблиц: Merge Join, Hash Join и Nested Loop.

Merge Join

Алгоритм работает следующим образом. Пусть у нас есть две отсортированные по одному ключу таблицы и мы хотим получить их соединение по этому же ключу. Тогда алгоритм будет выглядеть вот так:

1. Сохраняем информацию о текущей строке в левой таблице и в правой таблице.
2. Сохраняем указатель на левую и правую таблицы
3. Если ключ текущей строки слева равен ключу текущей строки справа
    - в результирующую таблицу пишем соединение этих двух строк в одну
    - двигаем указатель строки правой таблицы на строку вперед, возвращаемся в пункт 3
4. Если ключ текущей строки слева больше ключа справа:
    - двигаем указатель строки справа на строку вперед, возвращаемся в пункт 3
5. Если ключ текущей строки справа больше ключа слева:
    - двигаем указатель строки слева на строку вперед, возвращаемся в пункт 3
6. Оставшийся необработанный хвост таблицы — строки таблицы, которым не хватило пары из другой таблицы. В зависимости от вида джойна возможно добавление в результирующую таблицу строк с пустыми правыми/левыми частями.

В зависимости от вида джойна возможно добавление в результирующую таблицу строк с пустыми правыми/левыми частями.

Такой алгоритм очень эффективен за счет того, что он смотрит на данные каждой из таблиц только один раз и не хранит много состояния в оперативной памяти — нам нужно только лишь два указателя на таблицы слева и справа. При этом здесь необходимо выполнение требования отсортированности таблиц, что не всегда выполняется.

Hash Join

Достаточно простой алгоритм: правая таблица целиком загружается в оперативную память в виде хэш-таблицы, где ключом является ключ соединения, а значением — строка, соответствующая этому ключу либо список строк, если таких ключей несколько. Тогда, чтобы сделать соединение, нужно для каждой строки слева сделать обращение в хэш-таблицу и получить соответствующую строку справа.

Данный подход работает прекрасным образом, если таблица справа маленькая (влезает в оперативную память), а ключ не является длинным — операция хэширования данных сравнительно дорогая штука.

Nested Loop

Простой как топор алгоритм. Для каждой строки левой таблицы мы смотрим на каждую строку правой таблицы, и, если видим совпадение по ключу соединения, выводим в результат соединение пары строк.
Самый неэффективный алгоритм на хоть сколько-нибудь больших данных. При этом отлично работает на очень маленьких таблицах.

Все эти виды реализации соединения могут быть использованы базой. Это происходит в зависимости от размера данных, системы хранения (жесткий диск или SSD), настроек самой

базы и прочих фаз луны.

Оптимизация

Базовый флоу оптимизации выглядит следующим образом:

1. Написать бенчмарк и убедиться, что запрос плохой и выполняется неприемлемо медленно.
2. Почитать документацию по используемой базе, желательно раздел "Optimizing Performance".
3. Собрать статистику используемых таблиц (analyze).
4. Посмотреть на план запроса, понять, какие участки запроса тормозят и требуют очень много ресурсов.
    - Есть ли тяжелые джойны? Если есть, то по какому косту они тяжелые?
    - Как можно уменьшить число читаемых данных? Может есть смысл отфильтровать что-нибудь перед расчетом?
    - Если выбран nested loop для больших таблиц, нужно искать ответ на вопрос "Почему?", так как это очень нежелательное поведение.
    - Если выбран hash join и базу данных убивает система с ошибкой "недостаточно памяти", надо посмотреть на использование памяти базой. Были ли большие hash джойны между таблицами?
      Происходили ли они одновременно?
4. Посмотреть на сами запросы.
    - На какой вопрос они отвечают?
    - Правильно ли написан код?
    - Точно ли нужно было написать код именно так?
    - Спросить у автора кода вопрос что он имел ввиду?
5. Найти в документации к базе данных способы влияния на план запроса (хинты запроса).
6. Попробовать найти комбинацию параметров и такой переписанный запрос, что получается посчитать целевые данные быстрее.

Обычно во время ручной оптимизации происходит борьба с планировщиком, переписывание порядка джойнов, изменение структуры хранения данных, ресегментация/решардирование данных, изменение
процесса загрузки данных в хранилище, выбор другой модели хранения, выбор другой СУБД. (пункты приведены в примерной корреляции со сложностью каждого шага).