

# Оптимизация запросов Выполнение запросов



## **Авторские права**

© Postgres Professional, 2019 год.

Авторы: Егор Рогов, Павел Лузанов

## **Использование материалов курса**

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## **Обратная связь**

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## **Отказ от ответственности**

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Общие подходы к оптимизации

Простые запросы и этапы их обработки

Расширенные запросы

Подробнее о планировании

## Настройка параметров

- подстройка под имеющуюся нагрузку
- глобальное влияние на всю систему
- мониторинг

## Оптимизация запросов

- уменьшение нагрузки
- локальное воздействие (запрос или несколько запросов)
- профилирование

Эта тема начинает модуль, посвященный оптимизации запросов. Вообще «оптимизация» — очень широкое понятие; задумываться об оптимизации необходимо еще на этапе проектирования системы и выбора архитектуры. Мы будем говорить только о тех работах, которые выполняются при эксплуатации уже существующего приложения.

Можно выделить два основных подхода. Первый состоит в том, чтобы отслеживать состояние системы и добиваться того, чтобы она справлялась с имеющейся нагрузкой. Для этого можно настраивать параметры СУБД (основные из которых рассматриваются в курсе DBA2 и отчасти в этом модуле), а также настраивать операционную систему. Если настройки не помогают, при таком подходе остается только модернизировать аппаратуру (что тоже помогает не всегда).

Другой подход, который в основном мы и будем рассматривать далее, состоит в том, чтобы не приспособливаться под нагрузку, а уменьшать ее. «Полезная» нагрузка формируется запросами. Если удастся найти узкое место, то можно попробовать тем или иным способом повлиять на выполнение запроса и получить тот же результат, потратив меньше ресурсов. Такой способ действует более локально (на отдельный запрос или ряд запросов), но уменьшение нагрузки благоприятно сказывается и на работе всей системы.

Мы начнем с того, что в деталях разберем механизмы выполнения запросов, а уже после этого поговорим о том, как распознать неэффективную работу и о конкретных способах воздействия.

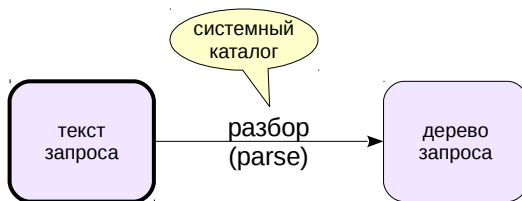
Разбор

Переписывание (трансформация)

Планирование (оптимизация)

Выполнение

Сначала рассмотрим, как выполняется запрос в простом случае – например, если в `psql` написать команду `SELECT`.



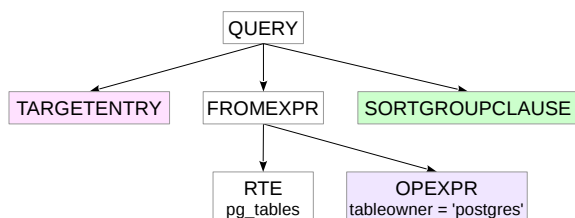
Обработка обычного запроса выполняется в несколько этапов.

Во-первых, запрос *разбирается* (parse).

Сначала производится синтаксический разбор: текст запроса представляется в виде дерева — так с ним удобнее работать.

Затем выполняется семантический анализ, в ходе которого определяется, на какие объекты БД ссылается запрос и есть ли у пользователя доступ к ним (для этого анализатор заглядывает в системный каталог).

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



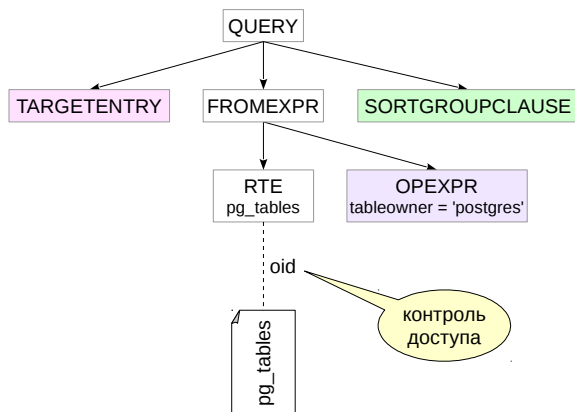
Рассмотрим простой пример: запрос, приведенный на слайде.

На этапе синтаксического разбора в памяти серверного процесса будет построено дерево, упрощенно показанное на рисунке ниже запроса. Цветом показано примерное соответствие частей текста запроса и узлов дерева.

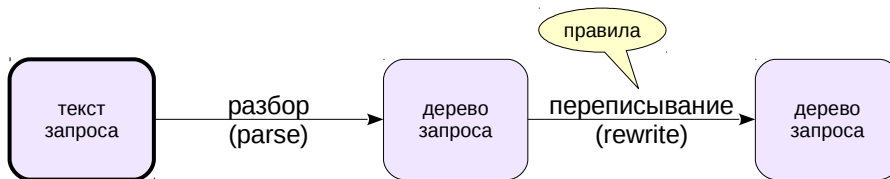
(RTE — неочевидное сокращение от Range Table Entry. Этим именем внутри PostgreSQL называются таблицы, подзапросы, результаты соединений — иными словами, наборы строк, с которыми может оперировать SQL.)

Если любопытно, то реальное дерево можно увидеть, установив параметр `debug_print_parse` и заглянув в журнал сообщений сервера. Практического смысла в этом нет (если, конечно, вы не разработчик ядра PostgreSQL).

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



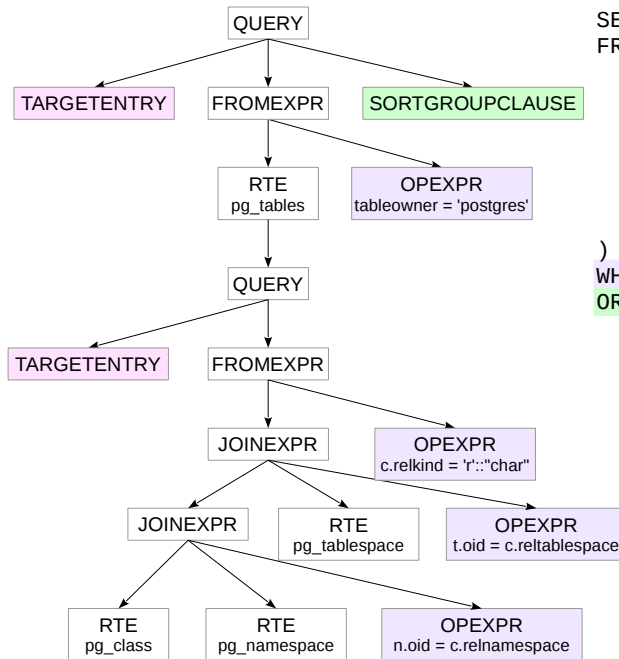
На этапе семантического разбора анализатор сверяется с системным каталогом и связывает имя «pg\_tables» с представлением, имеющим определенный идентификатор (oid) в системном каталоге. Также будут проверены права доступа к этому представлению.



Во-вторых, запрос *переписывается* или *трансформируется* (rewrite) с учетом правил.

Важный частный случай — подстановка текста запроса вместо имени представления. Заметим, что текст представления опять необходимо разобрать, поэтому мы несколько упрощаем, говоря, что первые два этапа происходят друг за другом.



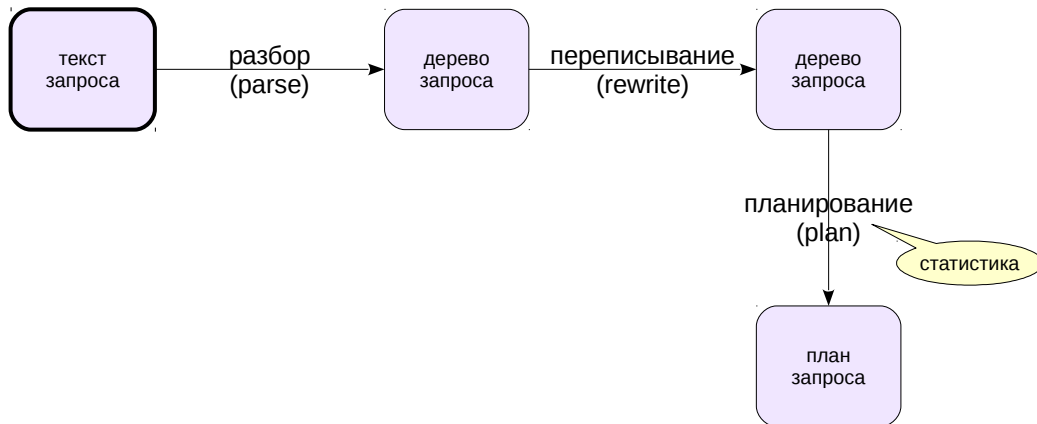


```
SELECT schemaname, tablename
FROM (
  SELECT ...
  FROM pg_class c
  LEFT JOIN pg_namespace n
    ON n.oid = c.relnamespace
  LEFT JOIN pg_tablespace t
    ON t.oid = c.reltablespace
  WHERE c.relkind = 'r'::"char"
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

На слайде приведен запрос с подставленным текстом представления (это условность: реально в таком виде запрос не существует — вся работа по переписыванию происходит только с деревом запроса).

Поддерево, соответствующее подзапросу, имеет своим родителем узел, ссылающийся на представление. На рисунке в этом поддереве хорошо видна древовидная структура запроса.

Трансформированное дерево можно увидеть в журнале сообщений сервера, установив параметр `debug_print_rewritten`.

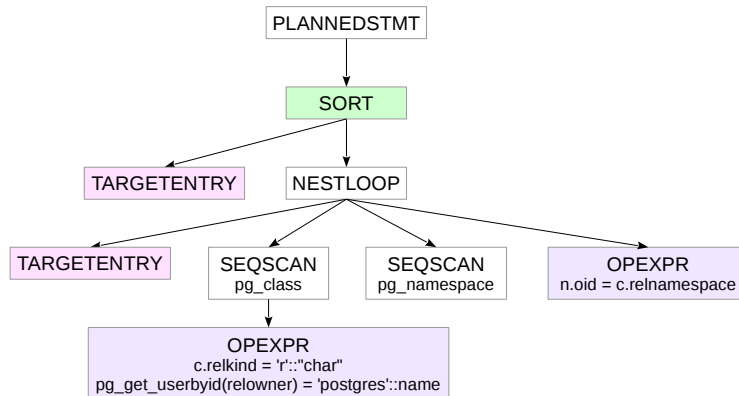


В-третьих, запрос *планируется* (plan).

SQL — декларативный язык, и один запрос можно выполнить разными способами. Планировщик (он же оптимизатор) перебирает различные способы выполнения и оценивает их. Оценка дается на основе некоторой математической модели исходя из информации об обрабатываемых данных (статистики).

Тот способ выполнения, для которого прогнозируется минимальная стоимость, представляется в виде дерева плана.

```
Sort (cost=19.59..19.59 rows=1 width=128)
  Sort Key: c.relname
  -> Nested Loop Left Join (cost=0.00..19.58 rows=1 width=128)
    Join Filter: (n.oid = c.relnamespace)
    -> Seq Scan on pg_class c (cost=0.00..18.44 rows=1 width=72)
      Filter: ((relkind = 'r'::"char") AND
              (pg_get_userbyid(reowner) = 'postgres'::name))
    -> Seq Scan on pg_namespace n (cost=0.00..1.06 rows=6 width=68)
```



11

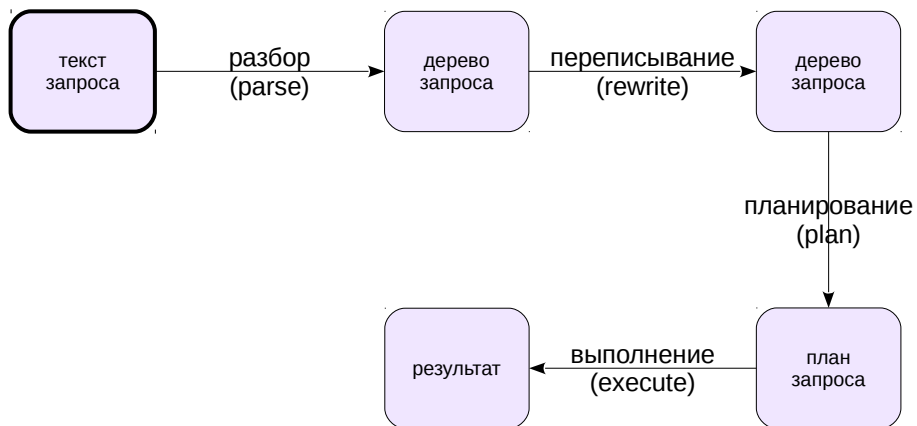
На слайде приведен пример дерева плана, которое представляет способ выполнения запроса.

Здесь операции Seq Scan — это чтение соответствующих таблиц, а Nested Loop — способ соединения двух таблиц. В виде текста на слайде приведен план выполнения в том виде, как его показывает команда EXPLAIN. Подробно о методах доступа к данным, о способах соединения и об EXPLAIN мы будем говорить в следующих темах.

Пока имеет смысл обратить внимание на два момента:

- из трех таблиц осталось только две: планировщик сообразил, что одна из таблиц не нужна для получения результата и ее можно безболезненно удалить из дерева плана;
- каждый узел дерева снабжен информацией о предполагаемом числе строк (rows) и о стоимости (cost).

Для интересующихся — увидеть «настоящее» дерево плана можно, установив параметр `debug_print_plan`.



В-четвертых, запрос *выполняется* (execute) в соответствии с выбранным планом, и результат возвращается клиенту.

<https://postgrespro.ru/docs/postgresql/10/query-path>

## Конвейер

обход дерева от корня вниз  
данные передаются вверх — по мере поступления или все сразу

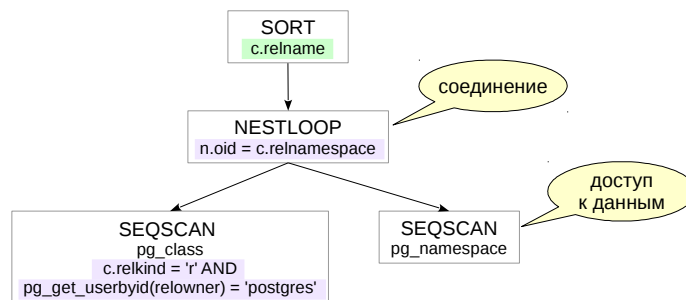
## Доступ к данным

чтение таблиц, индексов

## Соединения

всегда попарно  
важен порядок

## Другие операции



13

На этапе выполнения дерево плана (на слайде оно перерисовано так, чтобы показать только самое основное) работает как конвейер.

Выполнение начинается с корня дерева. Корневой узел (в нашем случае это операция сортировки **SORT**) обращается за данными к нижестоящему узлу; получив данные, он выполняет свою работу (сортировку) и отдает данные вверх (то есть клиенту).

Некоторые узлы (как **NESTLOOP** на рисунке) соединяют данные, полученные из разных источников. Здесь узел обращается по очереди к двум дочерним узлам (соединение всегда осуществляется попарно) и, получив от них строки, соединяет их и возвращает вверх (то есть узлу сортировки).

Два нижних узла представляют обращение к таблице за данными. Они читают строки из соответствующих таблиц и отдают вверх (то есть узлу соединения).

Некоторые узлы могут вернуть результат, только получив от нижестоящих узлов все данные. К таким узлам относится сортировка — нельзя отсортировать неполную выборку. Другие узлы могут возвращать данные по мере поступления. Например, доступ к данным с помощью чтения таблицы может отдавать вверх данные по мере чтения (это позволяет быстро получить первую часть результата — например, для страничного отображения на веб-странице).

Чтобы разобраться с планами выполнения, нужно понять, какие существуют методы доступа к данным, какие есть способы соединения этих данных, и посмотреть некоторые другие операции.

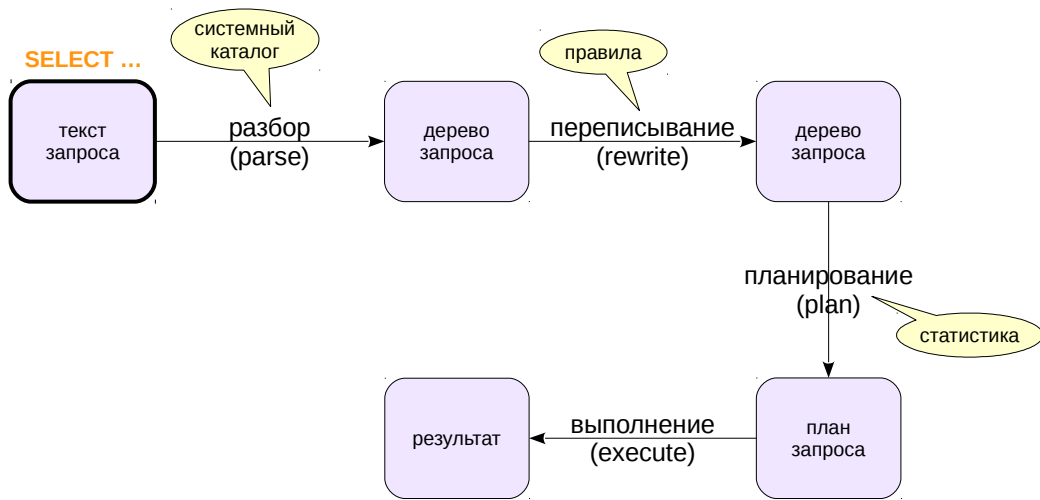
Уточнение схемы обработки запроса

Подготовленные операторы

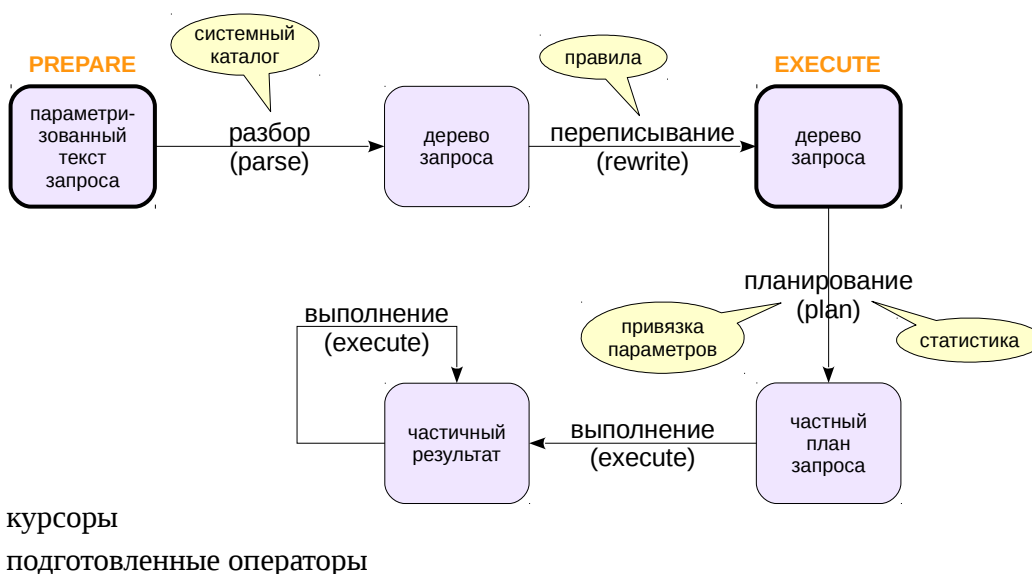
Курсоры

PostgreSQL также предусматривает протокол расширенных запросов. На практике это означает возможность использования подготовленных операторов и курсоров.

# Простые запросы



На слайде повторно приведена полная схема обработки простых запросов, которую мы уже рассмотрели.



Расширенный протокол позволяет более детально управлять обработкой запроса.

Во-первых, запрос может быть *подготовлен*. Для этого клиент передает запрос серверу (возможно, в параметризованном виде), а сервер выполняет разбор и переписывание и сохраняет подготовленное дерево запроса в локальную память обслуживающего процесса.

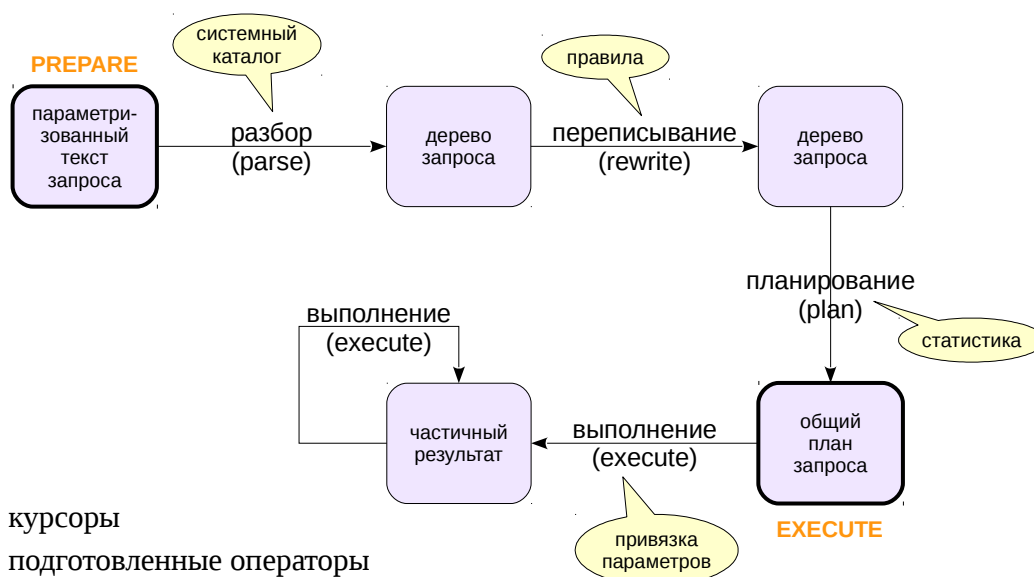
Чтобы выполнить подготовленный запрос, клиент называет его имя и указывает конкретные значения параметров. Сервер планирует запрос исходя из переданных параметров и выполняет его.

За счет подготовки удастся избежать повторного разбора и переписывания одного и того же запроса, если он выполняется в одном сеансе неоднократно.

Во-вторых, можно использовать *курсоры*. Механизм курсоров позволяет получать результат выполнения запроса не весь сразу, а построчно. Информация об открытом курсоре также хранится в локальной памяти обслуживающего процесса.

<https://postgrespro.ru/docs/postgresql/10/protocol-flow#PROTOCOL-FLOW-EXT-QUERY>





Если запрос не имеет параметров, серверу нет смысла перепланировать запрос при каждом выполнении. В этом случае он сразу запоминает общий (generic) план выполнения. Это позволяет еще больше экономить ресурсы.

Если же запрос имеет параметры, то сервер несколько раз (точнее, 5) строит частные (custom) планы в соответствии с переданными параметрами. Если после этого сервер замечает, что общий (generic) план получается «не хуже» частных, он переключается на него и перестает выполнять перепланирование. Подробнее о том, в каких случаях это происходит, говорится в теме «Статистика» этого модуля.

Есть и еще один повод использовать подготовленные операторы: гарантировать безопасность от внедрения SQL-кода, если входные данные для запроса получены из ненадежного источника (например, из поля ввода на веб-форме). Мы посмотрим на это в демонстрации.



Процесс планирования

Оценка кардинальности

Оценка стоимости

Выбор наилучшего плана

Планирование запроса – очень важный и достаточно сложный этап, поэтому остановимся на нем подробнее.

## Статистика

данные о размере таблиц и распределении данных

## Оценка кардинальности

селективность условий — доля выбираемых строк,  
кардинальность — итоговое число строк  
для расчета требуется статистика

## Оценка стоимости

в первую очередь зависит от типа узла и числа обрабатываемых строк

## Перебор планов оптимизатором

выбирается план с наименьшей стоимостью

Оптимизатор перебирает всевозможные планы выполнения, оценивает их и выбирает план с наименьшей стоимостью.

Чтобы оценить стоимость узла, нужно знать тип этого узла (понятно, что стоимость чтения данных напрямую из таблицы или с помощью индекса будет отличаться) и объем обрабатываемых этим узлом данных.

Для оценки объема данных важны два понятия:

- *кардинальность* — общее число строк;
- *селективность* — доля строк, отбираемых условиями (предикатами).

Для оценки селективности и кардинальности надо, в свою очередь, иметь сведения о данных: размер таблиц, распределение данных по столбцам.

Таким образом, в итоге все сводится к *статистике* — информации, собираемой и обновляемой процессом автоанализа или командой ANALYZE.

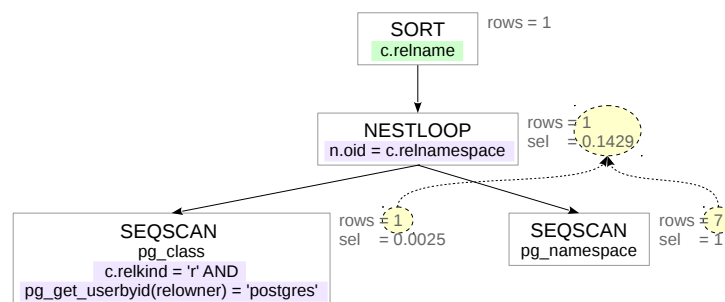
Если кардинальность оценена правильно, то и стоимость обычно рассчитывается довольно точно. Основные ошибки оптимизатора связаны именно с неправильной оценкой кардинальности. Это может происходить из-за неадекватной статистики, невозможности ее использования или несовершенства моделей, лежащих в основе оптимизатора. Об этом мы будем говорить подробнее.

## Кардинальность метода доступа

$$rows_{A \text{ where } cond} = rows_A \cdot sel_{cond}$$

## Кардинальность соединения

$$rows_{A \text{ join } B \text{ on } cond} = rows_A \cdot rows_B \cdot sel_{cond}$$



Оценку кардинальности удобно рассматривать как рекурсивный процесс. Чтобы оценить кардинальность узла, надо сначала оценить кардинальности дочерних узлов, а затем — зная тип узла — вычислить на их основе кардинальность самого узла.

Таким образом, сначала можно рассчитать кардинальности листовых узлов, в которых находятся методы доступа к данным. Для этого нам нужно знать размер таблицы и селективность условий, наложенных на нее. Как конкретно это делается, мы рассмотрим позже.

Пока отметим, что достаточно уметь оценивать селективность простых условий, а селективность условий, составленных с помощью логической операций, рассчитывается по простым формулам:

$$sel_{x \text{ and } y} = sel_x \cdot sel_y; \quad sel_{x \text{ or } y} = 1 - (1 - sel_x) (1 - sel_y)$$

К сожалению, эти формулы предполагают *независимость* предикатов. В случае коррелированных предикатов такая оценка будет неточной.

Затем можно рассчитать кардинальности соединений. Кардинальности соединяемых наборов данных нам уже известны, осталось оценить селективность условий соединения. Пока будем просто считать, что это как-то возможно.

Аналогично можно поступить и с другими узлами, например, с сортировками или агрегациями.

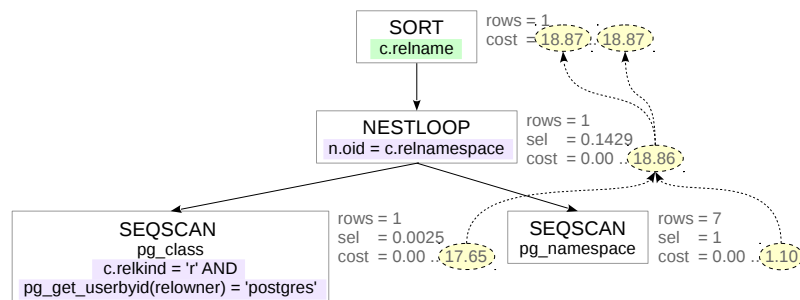
Важно отметить, что ошибка расчета кардинальности, возникшая в нижних узлах, будет распространяться выше, приводя в итоге к неверной оценке и выбору неудачного плана.

Вычисляется на основе математических моделей

$$cost = cost_A + \sum cost_{\text{дочерние узлы } A}$$

Две компоненты

подготовительная работа .. получение всех строк



Теперь рассмотрим общий процесс оценки стоимости. Он также рекурсивен по своей природе. Чтобы рассчитать стоимость поддерева, сначала надо рассчитать стоимости дочерних узлов, сложить их, и добавить стоимость самого узла.

Стоимость работы самого узла рассчитывается на основе математической модели, заложенной в планировщик, с учетом оценки числа обрабатываемых строк (которая уже рассчитана).

Стоимость состоит из двух компонентов, оцениваемых отдельно. Первый — стоимость подготовительной работы, второй — стоимость получения всех строк выборки.

Некоторые операции не требуют никакой подготовки; у таких узлов первая компонента стоимости будет равна нулю.

Некоторые операции, наоборот, требуют выполнения предварительных действий. Например, сортировка в приведенном примере должна сначала получить от дочернего узла все данные, чтобы начать работу. У таких узлов первая компонента будет отлична от нуля — эту стоимость придется «заплатить» независимо от того, сколько строк результата потребуется.

Важно понимать, что стоимость отражает *оценку* планировщика и может не коррелировать с реальным временем выполнения. Можно считать, что стоимость выражена в неких «условных единицах», которые сами по себе ни о чем не говорят. Стоимость нужна лишь для того, чтобы планировщик мог сравнивать разные планы одного и того же запроса.

## Перебор планов

порядок соединений, способы соединений, методы доступа  
по возможности полный перебор,  
при большом числе вариантов — генетический алгоритм (GEQO)

## Простые запросы и подготовленные операторы

минимальная общая стоимость  
оптимизируется время получения всех строк

## Курсоры

минимальная стоимость для получения *cursor\_tuple\_fraction* строк  
оптимизируется время получения части первых строк

Оптимизатор старается перебрать всевозможные планы выполнения запроса, чтобы выбрать из них лучший.

При большом количестве вариантов (в первую очередь на это влияет количество соединяемых таблиц) полный перебор за разумное время становится невозможен. В таком случае PostgreSQL переключается на использование генетического алгоритма (GEQO — Genetic Query Optimization). Это может привести к тому, что оптимизатор выберет не лучший план не из-за ошибок в оценке, а просто потому, что лучший план не рассматривался..

<https://postgrespro.ru/docs/postgresql/10/geqo>

Что же считается «лучшим планом»?

Для обычных запросов это план, минимизирующий время получения всех строк, то есть план с минимальной общей стоимостью (вторая компонента).

Однако при использовании курсоров может быть важно как можно быстрее получить первые строки. Поэтому существует параметр *cursor\_tuple\_fraction* (значение по умолчанию 0.1), задающий долю строк, которую надо получить как можно быстрее. Чем меньше значение этого параметра, тем больше на выбор плана влияет первая компонента стоимости, а не вторая.

Обработка запроса состоит из нескольких шагов:  
разбор и переписывание, планирование, выполнение

Имеются разные способы выполнения

- простой — непосредственное выполнение и получение результата
- расширенный — подготовленные операторы и курсоры

Время выполнения зависит от качества планирования

Планирование на основе стоимостного оптимизатора



1. Влияние подготовки на выполнение долгого оператора.  
Вычислите среднюю стоимость одного билета; посчитайте среднее время выполнения этого запроса.  
Подготовьте оператор для этого запроса; снова посчитайте среднее время выполнения.  
Во сколько раз ускорилось выполнение?
2. Влияние подготовки на выполнение коротких операторов.  
Множкратно запросите данные о бронировании с номером 0824С5; посчитайте среднее время выполнения.  
Подготовьте оператор для этого запроса; снова посчитайте среднее время выполнения.  
Во сколько раз ускорилось выполнение в этом случае?

Время выполнения одного и того же запроса может отличаться, причем довольно сильно (особенно время первого выполнения). Чтобы сгладить разницу, время надо усреднить, выполнив запрос несколько раз. Для этого удобно использовать язык PL/pgSQL, учитывая, что:

- динамический запрос, выполняемый командой PL/pgSQL EXECUTE (не путать с командой SQL EXECUTE!), каждый раз проходит все этапы;
- запрос SQL, встроенный в PL/pgSQL-код, выполняется с помощью подготовленных операторов.

Пример синтаксиса команды для обычного оператора:

```
DO $$
BEGIN
  FOR i IN 1..10 LOOP
    EXECUTE 'SELECT ... FROM ...';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Для подготовленного оператора (здесь SELECT заменяется на PERFORM, поскольку нас не интересует результат как таковой):

```
DO $$
BEGIN
  FOR i IN 1..10 LOOP
    PERFORM ... FROM ...;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```