Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Общие подходы к оптимизации

Простой протокол и этапы обработки запросов

Расширенный протокол

Подробнее о планировании

Подходы к оптимизации

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 приспосабливаться под нагрузку, а уменьшать ее. «Полезная» нагрузка формируется запросами. Если удается 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.

Этапы обработки запроса

Разбор

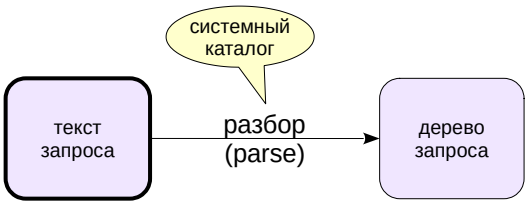
Переписывание (трансформация)

Планирование (оптимизация)

Выполнение

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

Разбор



Обработка обычного запроса выполняется в несколько этапов.

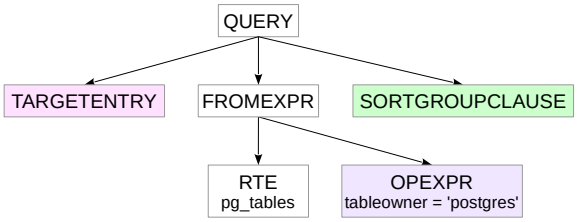
Во-первых, запрос *разбирается* (parse).

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).

Синтаксический разбор

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



Рассмотрим простой пример: запрос, приведенный на слайде.

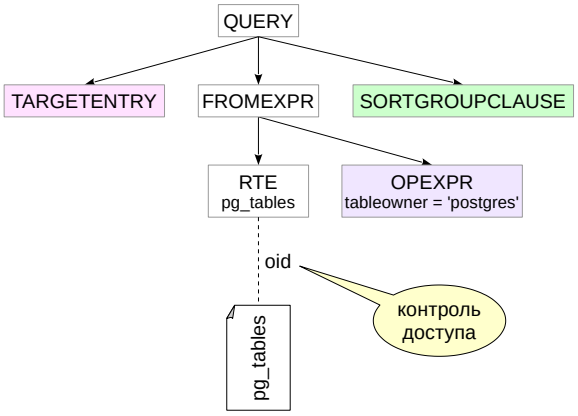
На этапе синтаксического разбора в памяти обслуживающего процесса будет построено дерево, упрощенно показанное на рисунке ниже запроса. Цветом показано примерное соответствие частей текста запроса и узлов дерева.

(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).

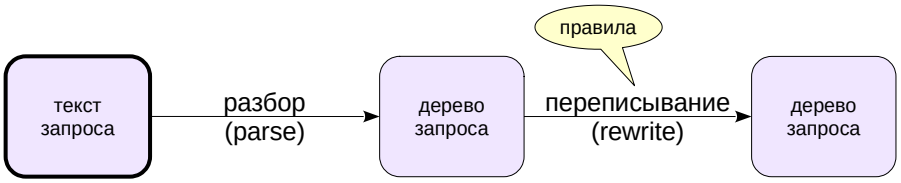
Семантический разбор

```
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.

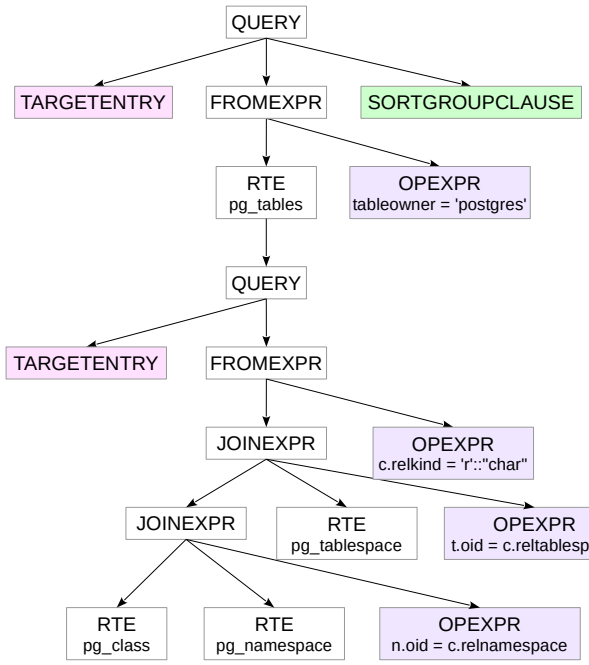
Переписывание



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

Важный частный случай переписывания — 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.

Переписывание



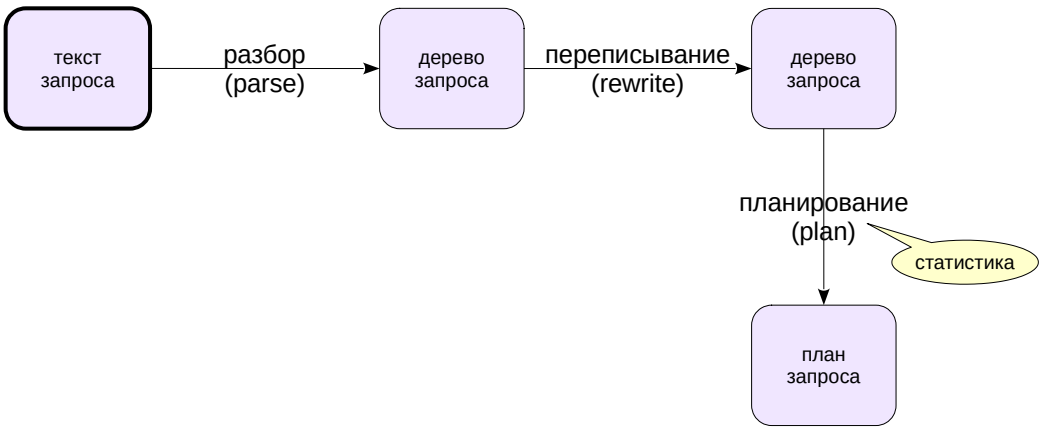

```
SELECT schemaname, tablename
FROM (
  SELECT ...
  FROM pg_class c
    LEFT JOIN pg_namespace n
      ON n.oid = c.relnamespace
    LEFT JOIN pg_tablespace t
      ON t.oid = c.reltablespace
  WHERE c.relkind = 'r'::"char"
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```


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).

Родительский узел поддерева, соответствующего подзапросу, — тот узел, который ссылается на это представление. На рисунке в этом поддереве хорошо видна древовидная структура запроса.

Дерево после переписывания можно увидеть в журнале сообщений сервера, установив параметр *debug_print_rewritten*.

Планирование



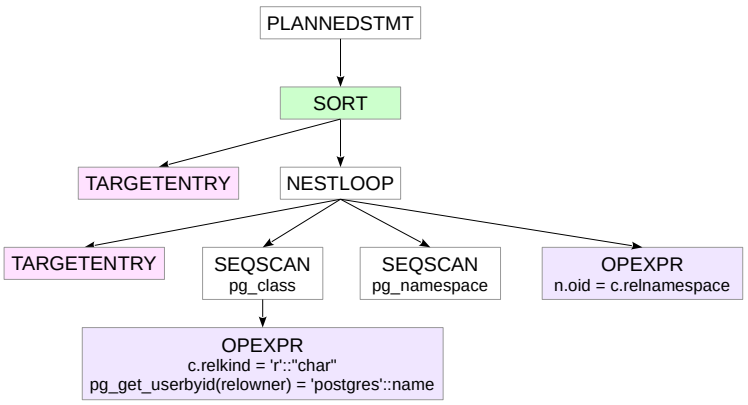
В-третьих, запрос *планируется* (plan).

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.

Планирование

```
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(relowner) = 'postgres'::name))
        ->  Seq Scan on pg_namespace n  (cost=0.00..1.06 rows=6 width=68)
```



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.

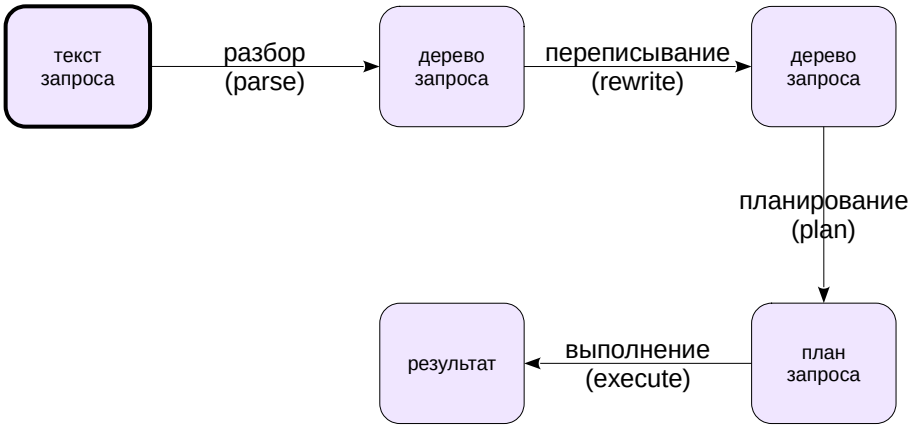
Пока имеет смысл обратить внимание на два момента:

- 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*.

Выполнение



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

https://postgrespro.ru/docs/postgresql/13/query-path

Выполнение

Конвейер

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

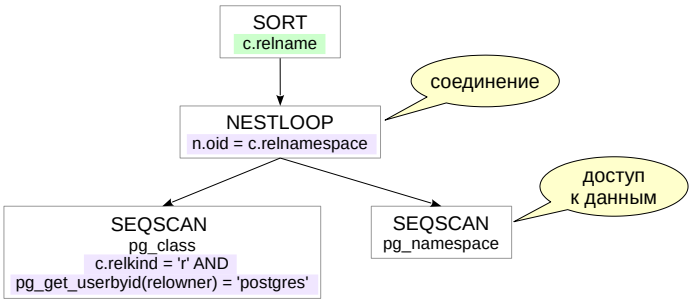
Доступ к данным

- чтение таблиц, индексов

Соединения

- всегда попарно
- важен порядок

Другие операции



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 наверх (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 наверх (то есть узлу сортировки).

Два нижних узла представляют обращение к таблице за данными. Они читают строки из соответствующих таблиц и отдают наверх (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 наверх 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.

Простой протокол

Простой протокол запросов применяется, когда на сервер отправляется оператор и, если это SELECT или команда с фразой RETURNING, мы ожидаем получение всех строк результата. Например:

```
=> SELECT model FROM aircrafts WHERE aircraft_code = '773';
```

```
     model
---------------
 Боинг 777-300
(1 row)
```

Установим параметры, которые покажут примерное время выполнения этапов обработки запроса:

```
=> ALTER SYSTEM SET log_parser_stats = on;
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET log_planner_stats = on;
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET log_executor_stats = on;
```

```
ALTER SYSTEM
```

```
=> SELECT pg_reload_conf();
```

```
 pg_reload_conf
----------------
 t
(1 row)
```

Обычный удобный способ узнать время выполнения и время планирования — команда EXPLAIN ANALYZE. Ее основное назначение — показать план выполнения запроса, но о плане как таковом мы будем говорить позже. Пока обратите внимание на две последние строки:

```
=> EXPLAIN (analyze, costs off, timing off)
SELECT * FROM ticket_flights;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ticket_flights (actual rows=8391852 loops=1)
 Planning Time: 0.158 ms
 Execution Time: 1847.408 ms
(3 rows)
```

Благодаря установленным нами параметрам, из журнала сообщений можно получить более подробную информацию, хотя обычно это и не требуется. Выведем только основные цифры (elapsed):

```
postgres$ tail -n 50 /var/log/postgresql/postgresql-13-main.log | egrep 'LOG: |elapsed'
```

```
2022-12-17 18:50:21.708 MSK [9566] postgres@demo LOG:  PARSE ANALYSIS STATISTICS
        !       0.000059 s user, 0.000059 s system, 0.000118 s elapsed
2022-12-17 18:50:21.708 MSK [9566] postgres@demo LOG:  REWRITER STATISTICS
        !       0.000000 s user, 0.000000 s system, 0.000001 s elapsed
2022-12-17 18:50:21.708 MSK [9566] postgres@demo LOG:  PLANNER STATISTICS
        !       0.000071 s user, 0.000071 s system, 0.000143 s elapsed
2022-12-17 18:50:23.556 MSK [9566] postgres@demo LOG:  EXECUTOR STATISTICS
        !       0.276168 s user, 0.604304 s system, 1.847644 s elapsed
```

Разные способы измерения выдают несколько отличающиеся результаты.

JIT-компиляция

Компиляция части запросов в исходный код

- вычисление выражений в предложении WHERE
- вычисления целевого списка выражений (SELECT)
- агрегаты и проекции

Преобразование версий строк

- перенос версий строк с диска в развернутое представление в памяти

Динамическая компиляция JIT (just-in-time, «точно в нужное время») используется для компиляции кода или его фрагментов в момент выполнения программы. Эта технология позволяет ускорить выполнение интерпретируемого кода и используется во многих системах.

В PostgreSQL с помощью JIT-компиляции можно скомпилировать часть кода, который выполняется при работе запросов SQL. Для этого PostgreSQL должен быть собран с поддержкой LLVM.

JIT-компиляция лучше подходит для длительных, нагружающих процессор аналитических запросов. Для коротких OLTP-запросов накладные расходы на JIT-компиляцию могут превышать время выполнения самих запросов.

Влиять на JIT-компиляцию можно с помощью конфигурационных параметров. Есть несколько оптимизаций, связанных с JIT; они включаются, только если стоимость запроса превышает указанное в соответствующих параметрах граничное значение.

https://postgrespro.ru/docs/postgresql/13/jit-reason

JIT-компиляция

По умолчанию JIT-компиляция включена:

```
=> SHOW jit;
```

```
 jit
-----
 on
(1 row)
```

Выполним запрос, рассчитывающий значение числа π. В запросе активно используются вычисления, которые JIT может оптимизировать:

```
=> WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10000000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

```
    val
-----------
 3.1415916
(1 row)
```

Команда EXPLAIN ANALYZE покажет подробную информацию о том, какие оптимизации JIT сработали:

```
=> EXPLAIN (analyze, timing off)
WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10000000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

```
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=525000.00..525000.02 rows=1 width=8) (actual rows=1 loops=1)
   CTE pi
     ->  Function Scan on generate_series  (cost=0.00..150000.00 rows=10000000 width=16) (actual rows=10000000 loops=1)
   ->  CTE Scan on pi  (cost=0.00..200000.00 rows=10000000 width=16) (actual rows=10000000 loops=1)
 Planning Time: 0.690 ms
 JIT:
   Functions: 6
   Options: Inlining true, Optimization true, Expressions true, Deforming true
 Execution Time: 4871.412 ms
(9 rows)
```

Выключим JIT-компиляцию и повторим запрос:

```
=> SET jit = off;
```

```
SET
```

```
=> EXPLAIN (analyze, timing off)
WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10000000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

```
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=525000.00..525000.02 rows=1 width=8) (actual rows=1 loops=1)
   CTE pi
     ->  Function Scan on generate_series  (cost=0.00..150000.00 rows=10000000 width=16) (actual rows=10000000 loops=1)
   ->  CTE Scan on pi  (cost=0.00..200000.00 rows=10000000 width=16) (actual rows=10000000 loops=1)
 Planning Time: 0.077 ms
 Execution Time: 5255.514 ms
(6 rows)
```

Время выполнения запроса немного увеличивается.

В рамках курса мы не будем подробно рассматривать оптимизации JIT. Чтобы сообщения о JIT-компиляции не загромождали планы запросов, в дальнейших демонстрациях JIT отключен.

```
=> SET jit = off;
```

```
SET
```

Расширенный протокол

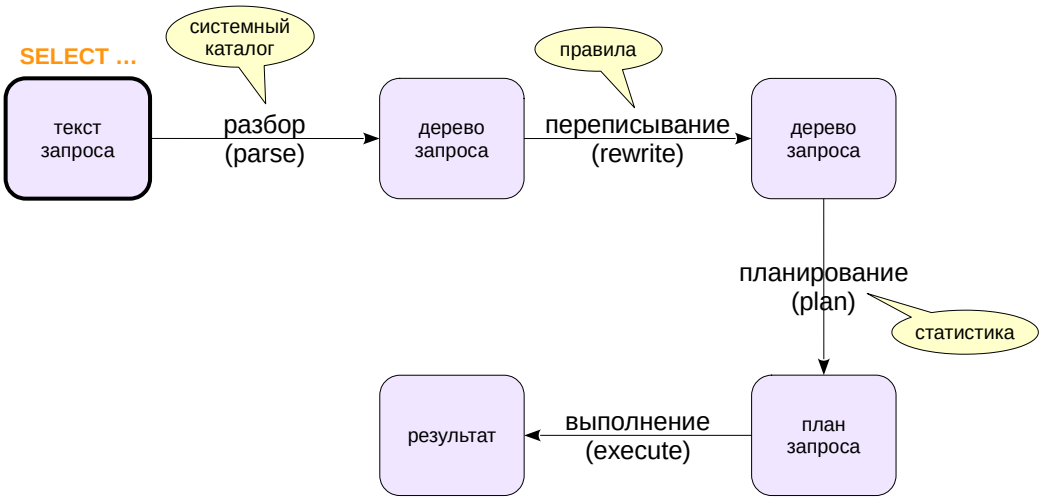
Уточнение схемы обработки запроса

Подготовленные операторы

Курсоры

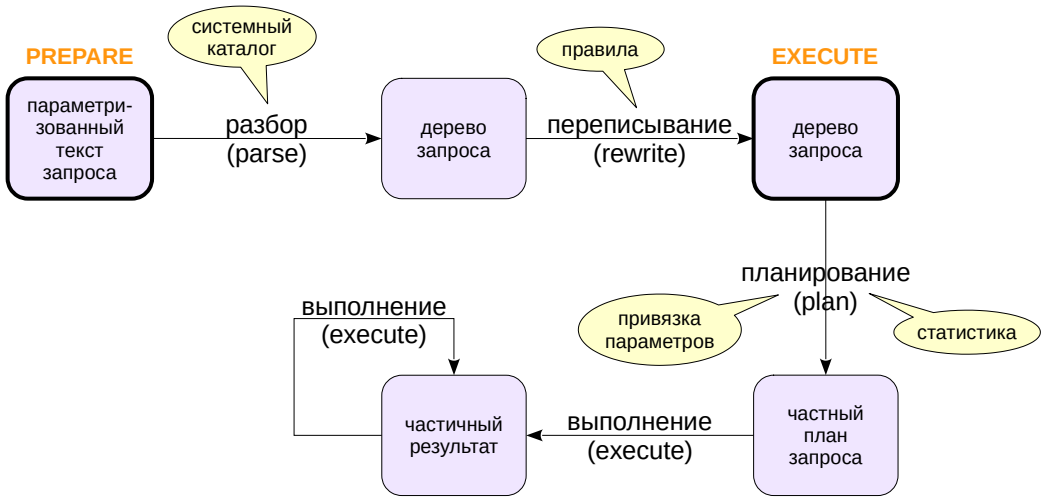
PostgreSQL также предусматривает расширенный протокол запросов. На практике это означает возможность использования подготовленных операторов и курсоров.

Простой протокол



На слайде повторно приведена полная схема обработки запросов по простому протоколу, которую мы уже рассмотрели.

Расширенный протокол



курсоры

подготовленные операторы

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.

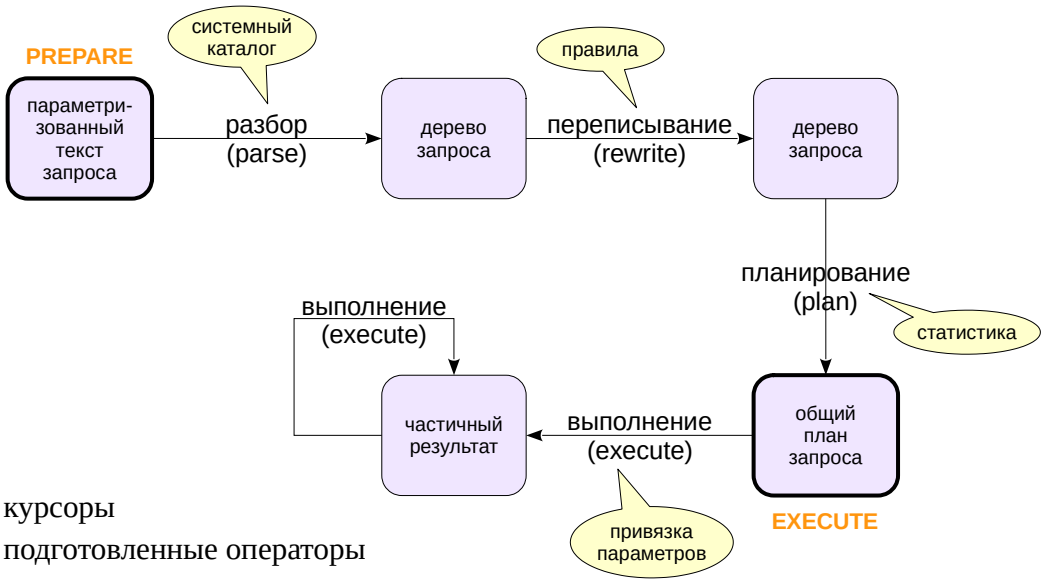
Чтобы выполнить подготовленный запрос, клиент называет его имя и указывает конкретные значения параметров. Сервер планирует запрос исходя из переданных параметров и выполняет его.

За счет подготовки удается 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/13/protocol-flow#PROTOCOL-FLOW-EXT-QUERY

Расширенный протокол



курсоры

подготовленные операторы

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 может перейти на общий план, если сочтет, что в среднем он получается не хуже частных планов. Подробнее о том, в каких случаях происходит переключение, говорится в теме «Статистика».

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).

Подготовленные операторы

Создадим подготовленный оператор с параметром для запроса:

```
=> PREPARE model(varchar) AS
  SELECT model FROM aircrafts WHERE aircraft_code = $1;
```

```
PREPARE
```

Теперь мы можем вызывать оператор по имени:

```
=> EXECUTE model('773');
```

```
     model
---------------
 Боинг 777-300
(1 row)
```

```
=> EXECUTE model('763');
```

```
     model
---------------
 Боинг 767-300
(1 row)
```

Все подготовленные операторы можно увидеть в представлении:

```
=> SELECT * FROM pg_prepared_statements \gx
```

```
-[ RECORD 1 ]---+-----------------------------------------------------
name            | model
statement       | PREPARE model(varchar) AS                          +
                |   SELECT model FROM aircrafts WHERE aircraft_code = $1;
prepare_time    | 2022-12-17 18:50:39.371184+03
parameter_types | {"character varying"}
from_sql        | t
```

Если подготовленный оператор больше не нужен, его можно удалить командой DEALLOCATE, но в любом случае оператор пропадет при завершении сеанса.

```
=> \c
```

```
You are now connected to database "demo" as user "postgres".
```

```
=> SELECT * FROM pg_prepared_statements;
```

```
 name | statement | prepare_time | parameter_types | from_sql
------+-----------+--------------+-----------------+----------
(0 rows)
```

Команды PREPARE, EXECUTE, DEALLOCATE — команды SQL. Клиенты на других языках программирования будут использовать операции, определенные в соответствующем драйвере. Но любой драйвер использует один и тот же протокол для взаимодействия с сервером.

Курсоры

Курсоры дают возможность построчной обработки результата. Они часто используются в приложениях и имеют особенности, связанные с оптимизацией.

На SQL использование курсоров можно продемонстрировать следующим образом. Объявляем курсор (при этом он сразу же открывается) и выбираем первую строку:

```
=> BEGIN;
```

```
BEGIN
```

```
=> DECLARE c CURSOR FOR SELECT * FROM aircrafts;
```

```
DECLARE CURSOR
```

```
=> FETCH c;
```

```
 aircraft_code |     model     | range
---------------+---------------+-------
 773           | Боинг 777-300 | 11100
(1 row)
```

Читаем вторую строку результата и закрываем открытый курсор (курсор закроется и автоматически по окончании транзакции):

```
=> FETCH c;
```

```
 aircraft_code |     model     | range
---------------+---------------+-------
 763           | Боинг 767-300 |  7900
(1 row)
```

```
=> CLOSE c;
```

```
CLOSE CURSOR
```

```
=> COMMIT;
```

```
COMMIT
```

Подробнее о планировании

Процесс планирования

Оценка кардинальности

Оценка стоимости

Выбор наилучшего плана

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

Процесс планирования

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.

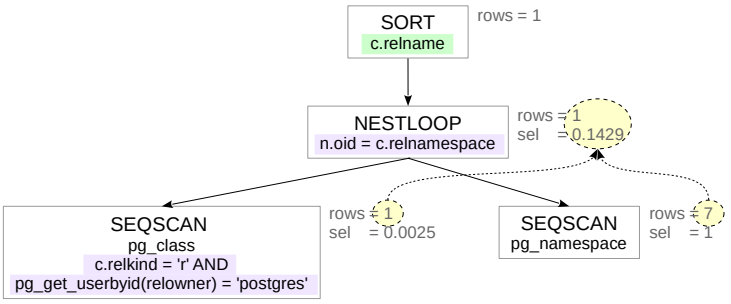
Оценка кардинальности

Кардинальность метода доступа

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

Кардинальность соединения

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



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 логический операций, рассчитываются по простым формулам:

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

Следует учитывать, что эти формулы предполагают *независимость* предикатов. В случае коррелированных предикатов такая оценка будет неточной (ее можно улучшить с помощью расширенной статистики).

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.

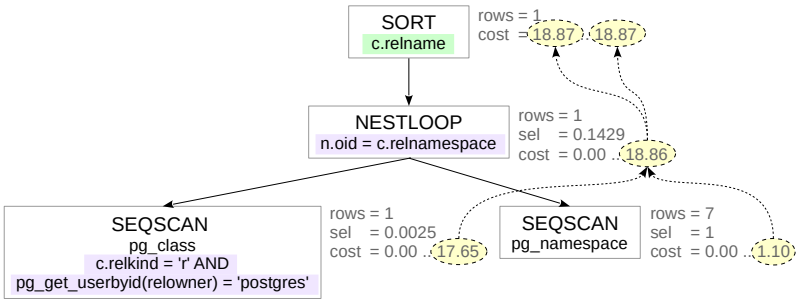
Оценка стоимости

Вычисляется на основе математических моделей

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

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 вычислить стоимости дочерних узлов и сложить их, а затем добавить стоимость самого узла.

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

Стоимость состоит из двух компонент, оцениваемых отдельно. Первая — стоимость подготовительной работы (*начальная стоимость*), вторая — стоимость получения всех строк выборки (*полная стоимость*).

Некоторые операции не требуют никакой подготовки; у таких узлов начальная стоимость будет равна нулю.

Другие 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 начальная стоимость 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.

Выбор лучшего плана

Перебор планов

- порядок соединений, способы соединений, методы доступа
- по возможности полный перебор,
  при большом числе вариантов — сокращение пространства поиска

Простые запросы и подготовленные операторы

- оптимизируется время получения всех строк
- минимальная полная стоимость

Курсоры

- оптимизируется время получения части первых строк
- минимальная стоимость получения *cursor_tuple_fraction* строк

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

Для сокращения перебора используется алгоритм динамического программирования, но при большом количестве вариантов (в первую очередь из-за числа соединяемых таблиц) точное решение задачи оптимизации за разумное время становится невозможным. В таком случае планировщик сокращает количество перебираемых планов, либо рассматривая не все варианты попарных соединений, либо переключаясь на генетический алгоритм оптимизации (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/13/geqo

Что же считается «лучшим планом»?

Для обычных запросов это план, минимизирующий время получения всех строк, то есть план с минимальной полной стоимостью.

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 начальная стоимость, а не полная.

Итоги

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

Имеется два протокола выполнения запросов

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

Время выполнения зависит от качества планирования

План строится оптимизатором на основе стоимости

Практика

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.

   Многократно запросите данные о бронировании с номером 0824C5; 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;
```

1. Долгий запрос

```
=> \timing on
```

```
Timing is on.
```

Обычный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..10 LOOP
    EXECUTE 'SELECT avg(amount) FROM ticket_flights';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
DO
Time: 14469,018 ms (00:14,469)
```

Подготовленный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..10 LOOP
    PERFORM avg(amount) FROM ticket_flights;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
DO
Time: 12966,729 ms (00:12,967)
```

Время изменилось незначительно — большую часть занимает выполнение запроса.

2. Быстрый запрос

Обычный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..100000 LOOP
    EXECUTE 'SELECT * FROM bookings WHERE book_ref = ''0824C5''';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
DO
Time: 3001,470 ms (00:03,001)
```

Подготовленный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..100000 LOOP
    PERFORM * FROM bookings WHERE book_ref = '0824C5';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
DO
Time: 649,588 ms
```

Время сократилось существенно — разбор и планирование занимает большую часть общего времени.