

# Выполнение запросов Планирование и выполнение



## Авторские права

© Postgres Professional, 2019–2024

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

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## 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

Простой протокол и этапы обработки запросов

Расширенный протокол

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

Этот курс посвящен 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 курсе), 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.

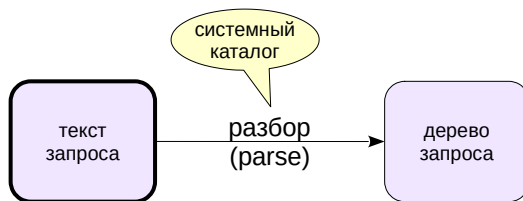
Разбор

Переписывание (трансформация)

Планирование (оптимизация)

Выполнение

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



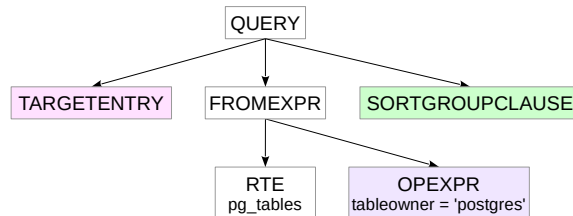
Обработка обычного запроса выполняется в несколько этапов.

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

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

Затем выполняется семантический анализ (parse analysis), 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;
```



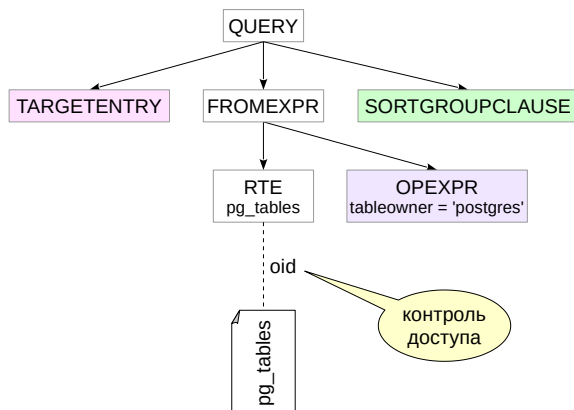
Рассмотрим простой пример: запрос, приведенный на слайде.

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

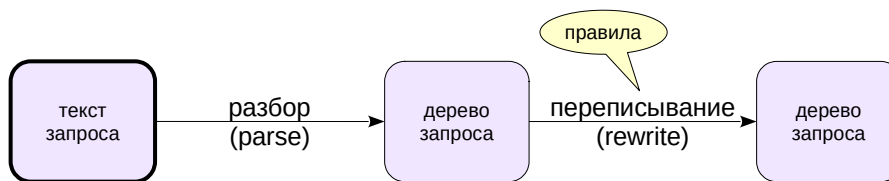
(RTE — неочевидное сокращение от Range Table Entry. Этим именем внутри PostgreSQL называются таблицы, подзапросы, результаты соединений — иными словами, наборы строк, с которыми может оперировать SQL.)

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).

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



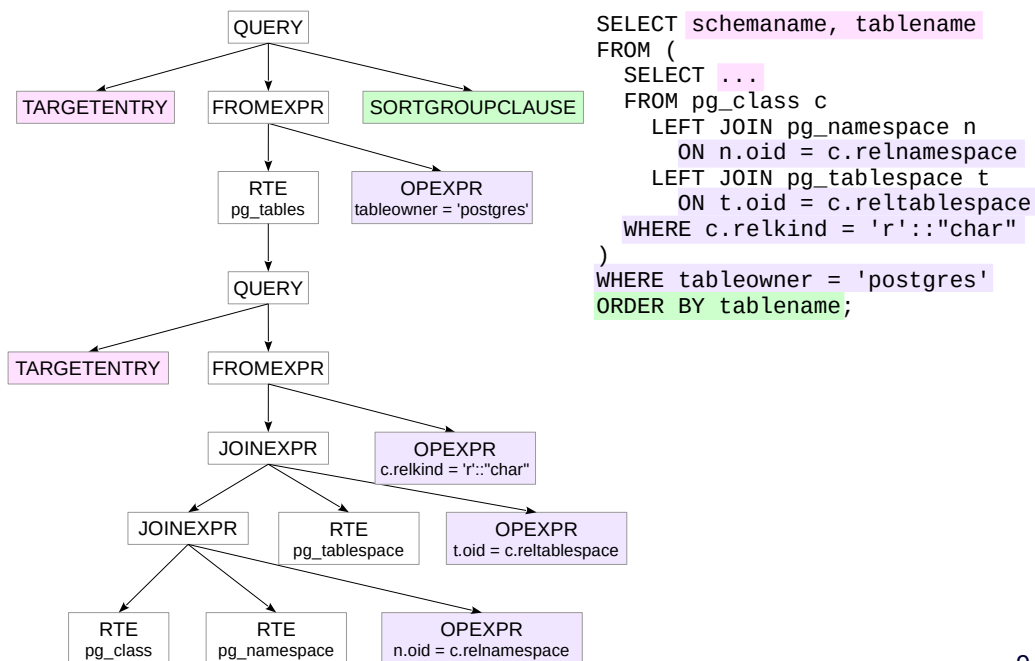
На этапе семантического разбора анализатор сверяется с системным каталогом и связывает имя «pg\_tables» с представлением, имеющим определенный идентификатор (oid) в системном каталоге. Также будут проверены права доступа к этому представлению.



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

Важный частный случай 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.

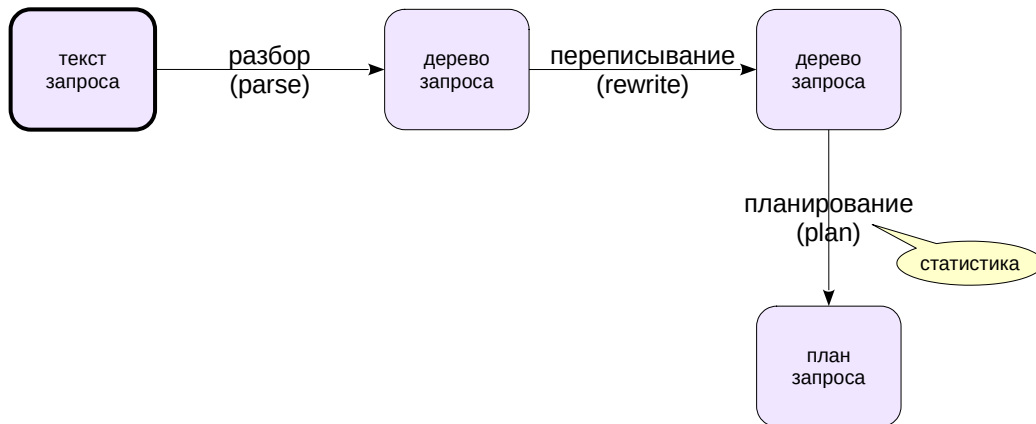




На слайде приведен запрос с подставленным текстом представления (это условность: реально в таком виде запрос не существует — вся работа по переписыванию происходит только с деревом запроса).

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`.

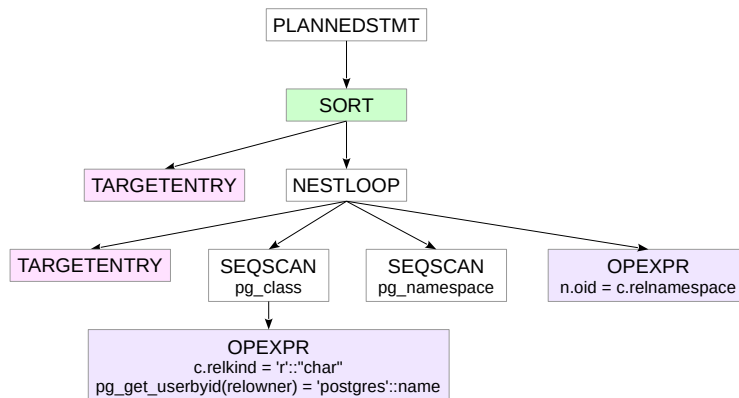


В-третьих, запрос *планируется* (plan).

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

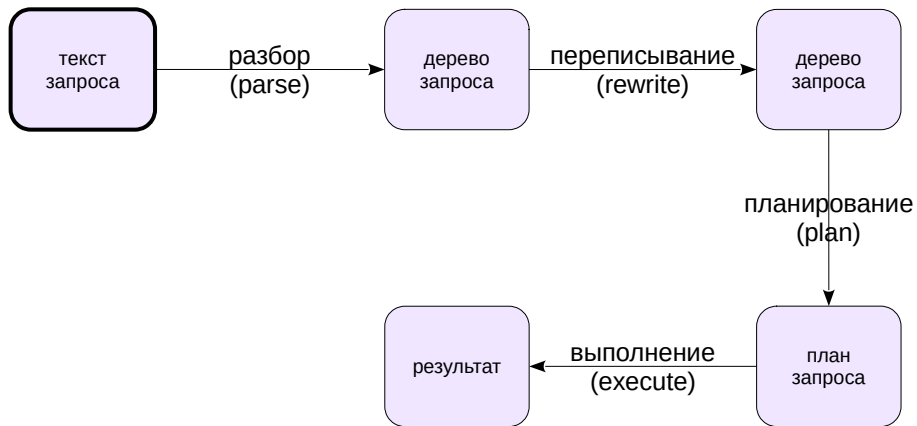
На слайде приведен пример дерева плана, которое представляет способ выполнения запроса.

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.

Задав параметрам `log_parser_stats`, `log_planner_stats` и `log_executor_stats` значение on, можно вывести в журнал подробную статистику каждого этапа. Но на практике это обычно не требуется.

<https://postgrespro.ru/docs/postgresql/16/query-path>

## 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)
```

Обычный удобный способ получить план запроса — команда EXPLAIN (пока отключаем вывод стоимости):

```
=> EXPLAIN (buffers, costs off)
SELECT * FROM airports;
```

```
          QUERY PLAN
-----
Seq Scan on airports_data ml
Planning:
  Buffers: shared hit=54 read=6 dirtied=1
(3 rows)
```

Если задан параметр buffers, в поле Planning Buffers будет показано число страниц буферного кеша, которое было прочитано при построении плана.

Если добавить параметр analyze, сервер выполнит запрос и отобразит его план с детальной информацией о выполнении. Будьте аккуратны с запросами, изменяющими данные!

```
=> EXPLAIN (analyze, buffers, costs off)
SELECT * FROM airports;
```

```
          QUERY PLAN
-----
Seq Scan on airports_data ml (actual time=0.061..0.306 rows=104 loops=1)
  Buffers: shared read=3
Planning Time: 0.052 ms
Execution Time: 0.331 ms
(4 rows)
```

В поле Buffers отображается тип кеша (shared — буферный кеш в общей памяти) и количество страниц:

- hit — найденных в кеше;
- read — не найденных в кеше и запрошенных у операционной системы;
- written — записанных в файлы;
- dirtied — ставших грязными (т.е. впервые измененных в кеше).

Теперь создадим временную таблицу и вставим в нее строки таблицы airports:

```
=> CREATE TEMP TABLE temp_airports (LIKE airports);
```

CREATE TABLE

```
=> EXPLAIN (analyze, buffers, costs off)
INSERT INTO temp_airports SELECT * FROM airports;
```

```
          QUERY PLAN
-----
Insert on temp_airports (actual time=0.748..0.748 rows=0 loops=1)
  Buffers: shared hit=3, local hit=104 dirtied=3 written=5
  -> Seq Scan on airports_data ml (actual time=0.064..0.404 rows=104 loops=1)
    Buffers: shared hit=3
Planning Time: 0.076 ms
Execution Time: 0.768 ms
(6 rows)
```

Временная таблица кешируется в локальной памяти сеанса (Buffers ... local).

Предупреждение: вывод времени выполнения каждого шага, как в этом примере, может существенно замедлять выполнение запроса на некоторых платформах. Если такая информация не нужна, лучше указывать timing off.

## Конвейер

обход дерева от корня вниз

данные передаются вверх — по мере поступления или все сразу

## Доступ к данным

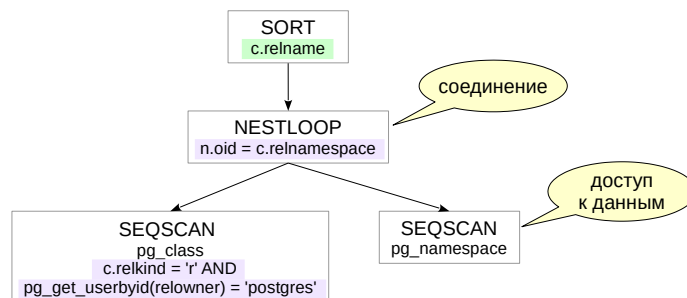
чтение таблиц, индексов

## Соединения

всегда попарно

важен порядок

## Другие операции



14

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 дочернему 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 дочерних 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.

## 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/16/jit-reason>

## JIT-компиляция

По умолчанию JIT-компиляция включена, но для демобазы она выключена:

```
=> SELECT setconfig
FROM pg_db_role_setting
WHERE setdatabase = (SELECT oid FROM pg_database WHERE datname='demo');
```

```
setconfig
-----
{"search_path=bookings, public",bookings.lang=ru,jit=off}
(1 row)
```

```
=> SELECT name, setting, boot_val
FROM pg_settings
WHERE name='jit';
```

```
name | setting | boot_val
-----+-----+-----
jit  | off     | on
(1 row)
```

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

```
=> SET jit = on;
```

SET

```
=> WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10_000_000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

```
val
-----
3.1410192
(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(1,10_000_000)
)
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: 0.094 ms
  JIT:
    Functions: 6
    Options: Inlining true, Optimization true, Expressions true, Deforming true
  Execution Time: 5311.483 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(1,10_000_000)
)
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: 1.061 ms
  Execution Time: 5681.739 ms
(6 rows)
```

Время выполнения запроса, скорее всего, немного увеличивается.

В рамках курса мы не будем подробно рассматривать оптимизации JIT. Чтобы сообщения о JIT-компиляции не загромождали планы запросов, на уровне демобазы JIT отключен.

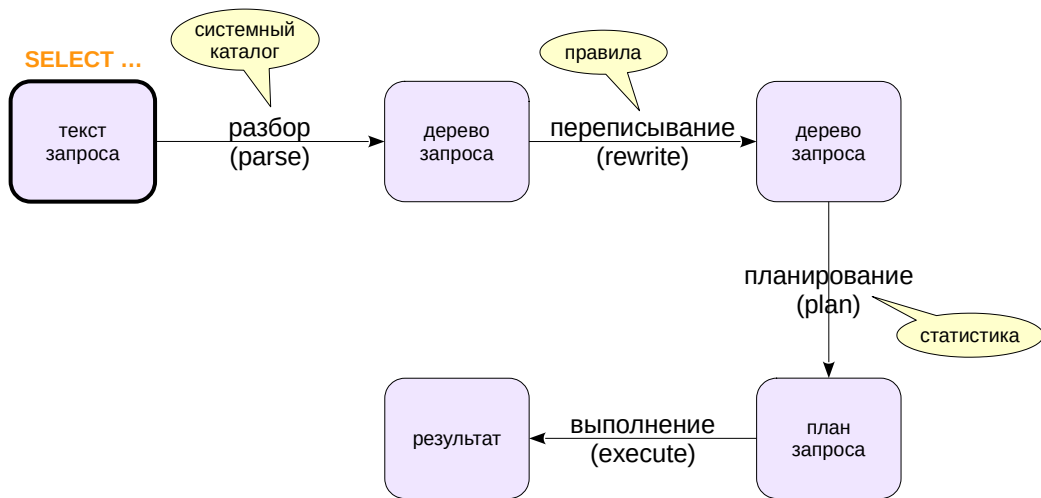
Уточнение схемы обработки запроса

Подготовленные операторы

Курсоры

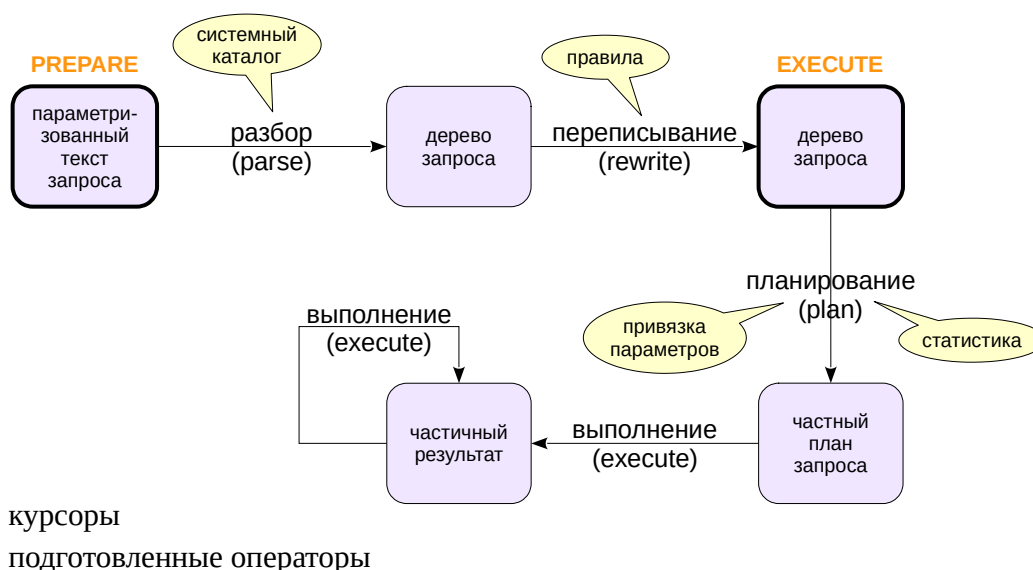
PostgreSQL также предусматривает расширенный протокол запросов. На практике это означает возможность использования подготовленных операторов и курсоров.

# Простой протокол



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

# Расширенный протокол



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.

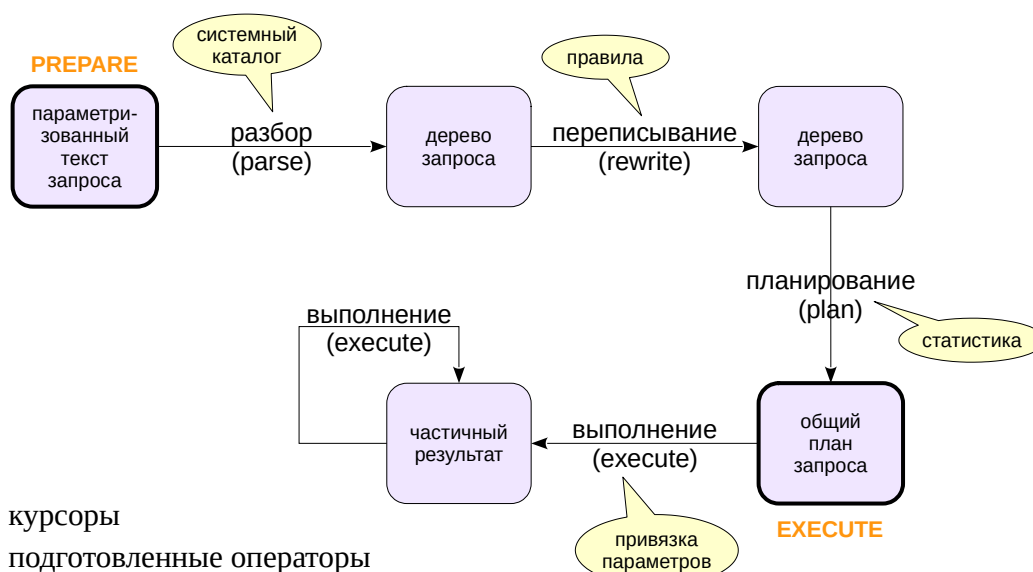
Чтобы выполнить подготовленный запрос, клиент называет его имя и указывает конкретные значения параметров. Сервер строит *частный план* запроса, учитывая значения переданных параметров, и выполняет его.

Подготовка позволяет 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/16/protocol-flow#PROTOCOL-FLOW-EXT-QUERY>

# Расширенный протокол



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 «Базовая статистика».

Есть и еще один повод использовать подготовленные операторы: гарантировать защиту 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 для запроса с параметром:

```
=> 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  | 2025-02-05 10:49:26.100531+03
parameter_types | {"character varying"}
result_types   | {text}
from_sql      | t
generic_plans  | 0
custom_plans  | 2
```

- name — имя подготовленного оператора,
- statement — создавшая его команда,
- prepare\_time — момент создания,
- parameter\_types — типы параметров (массив),
- result\_types — типы столбцов результата (массив),
- from\_sql — оператор создан командой SQL PREPARE (а не вызовом функции драйвера),
- generic\_plans — сколько раз использовался общий план,
- custom\_plans — сколько раз строились и использовались частные планы.

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 \gx
```

```
(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.

Если в psql выполнить запрос с параметрами, он будет использовать расширенный протокол запросов. В этом случае нужно предварительно задать значения параметров:

```
=> \bind '773'
```

Подготовленный оператор будет создан неявно.

```
=> SELECT model FROM aircrafts WHERE aircraft_code = $1;
```

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

После выполнения запроса подготовленный оператор удаляется:

```
=> SELECT * FROM pg_prepared_statements \gx
(0 rows)
```

Значения параметров после выполнения сбрасываются, поэтому задавать их нужно перед каждым запросом:

```
=> \bind '763'

=> EXPLAIN SELECT model FROM aircrafts WHERE aircraft_code = $1;

               QUERY PLAN
-----
Seq Scan on aircrafts_data ml  (cost=0.00..1.36 rows=1 width=32)
  Filter: (aircraft_code = '763'::bpchar)
(2 rows)
```

Такой подход можно использовать, если нужно отладить выполнение запроса с расширенным протоколом.

---

## 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
```

- 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.

При расчете стоимости узла плана оптимизатор учитывает тип этого узла (понятно, что стоимость чтения данных напрямую из таблицы будет отличаться от стоимости чтения с помощью индекса) и объем обрабатываемых узлом данных. Остальные факторы считаются менее существенными.

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 значений в столбцах и др.

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

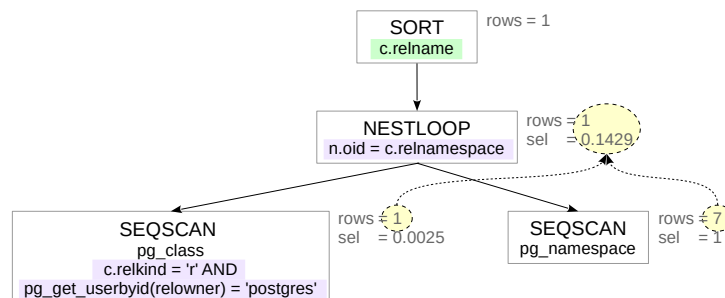
Если кардинальность оценена правильно, то и стоимость обычно рассчитывается довольно точно. Основные ошибки оптимизатора связаны именно с неправильной оценкой кардинальности. Это может происходить из-за неадекватной статистики, невозможности ее использования или несовершенства моделей, лежащих в основе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

## Кардинальность метода доступа

$$row_{SA \text{ where } cond} = row_{SA} \cdot sel_{cond}$$

## Кардинальность соединения

$$row_{SA \text{ join } B \text{ on } cond} = row_{SA} \cdot row_{SB} \cdot sel_{cond}$$



24

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 логических операций, рассчитываются по простым формулам:

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

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.

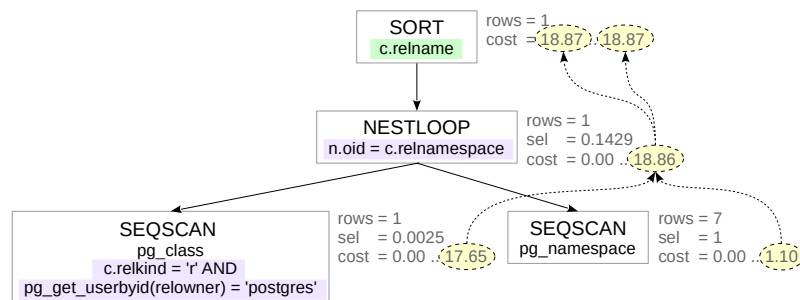
# Оценка стоимости

Вычисляется на основе математических моделей

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

Две компоненты

подготовительная работа .. получение всех строк



25

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.

## 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/16/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.

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

Имеются 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

1. Влияние подготовки на выполнение долгого оператора.  
Вычислите среднюю стоимость одного перелета; 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.  
Множественно 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?

28

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: 15905,117 ms (00:15,905)

Подготовленный оператор:

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

DO

Time: 14047,360 ms (00:14,047)

Время изменилось незначительно — большую часть занимает выполнение запроса.

## 2. Быстрый запрос

Обычный оператор:

```
=> DO $$  
BEGIN  
    FOR i IN 1..100_000 LOOP  
        EXECUTE 'SELECT * FROM bookings WHERE book_ref = ''0824C5''';  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 3170,243 ms (00:03,170)

Подготовленный оператор:

```
=> DO $$  
BEGIN  
    FOR i IN 1..100_000 LOOP  
        PERFORM * FROM bookings WHERE book_ref = '0824C5';  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 593,069 ms

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