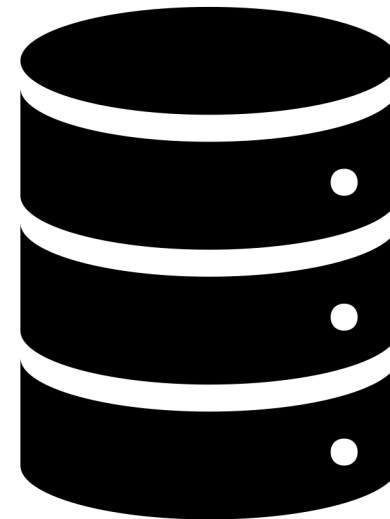


# Базы данных

Лекция 10. План запроса. Оптимизация запросов. Часть 1.



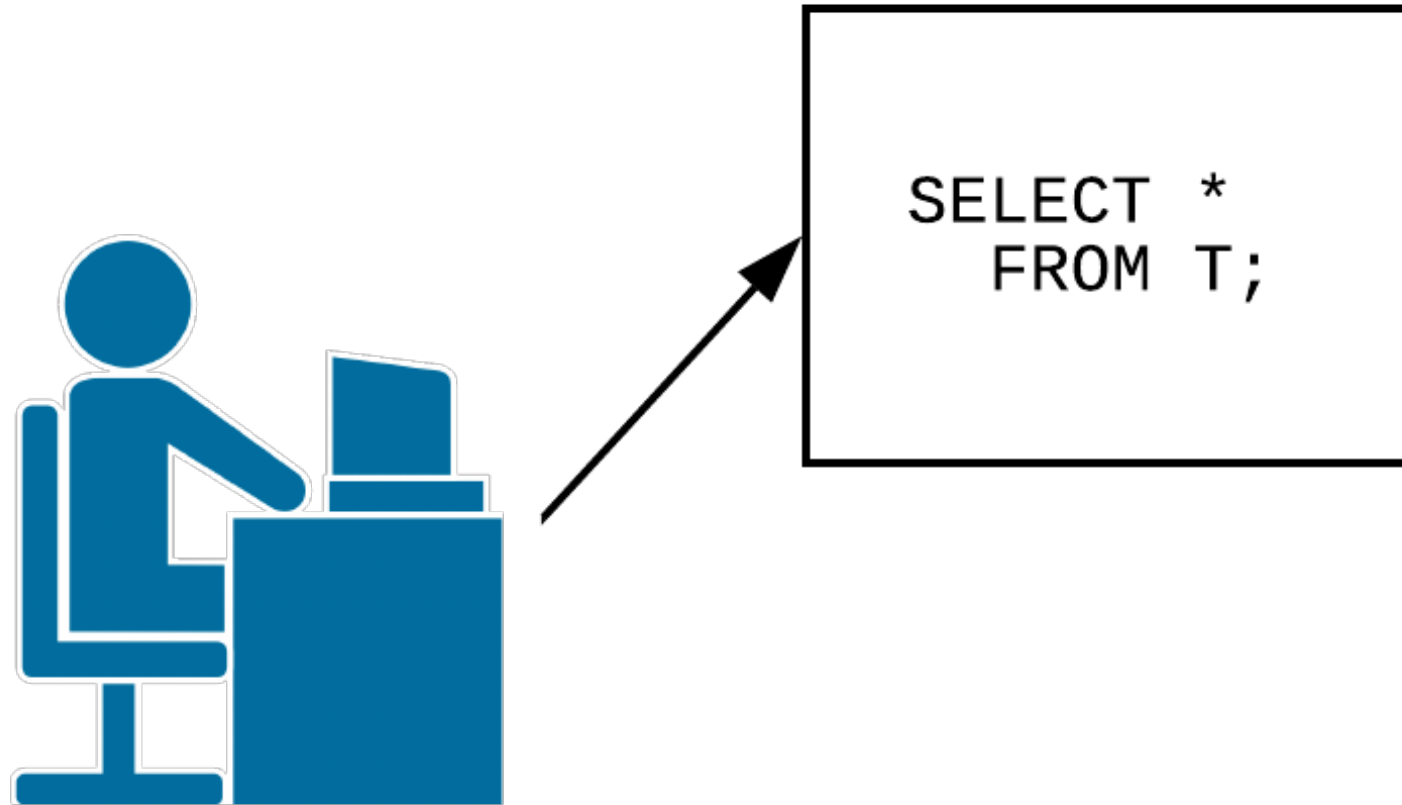
Меркурьева Надежда

✉ [merkurievanad@gmail.com](mailto:merkurievanad@gmail.com)

📧 [@merkurievanad](https://t.me/merkurievanad)

ФПМИ МФТИ, 2021

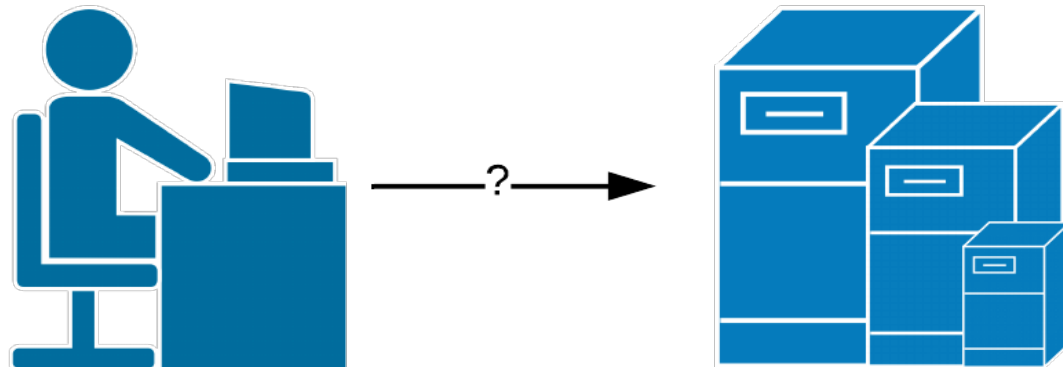
# Вы написали запрос. Что дальше?



На примере PostgreSQL

# 1. Подключение к СУБД

1. Клиентский процесс обращается к главному серверному процессу
2. Главный серверный процесс создает новый при запросе соединения
3. Когда соединение установлено, клиент может отправить на сервер запрос
4. Запрос будет передан в виде обычного текста
5. На этом этапе обработки запроса не происходит



## 2. Парсинг запроса

1. Парсер проверяет запрос на правильность синтаксиса
2. Парсер возвращает итог работы:
  - Дерево разбора, если синтаксис корректный
  - Ошибка, если синтаксис некорректный. Пользователь получает ошибку
3. Преобразователь принимает на вход дерево разбора и преобразует его в дерево запроса:
  - Формируется понимание, какие таблицы, функции и операторы использует запрос
  - Обращение к базе только в рамках транзакций

Если очень интересно, как происходит разбор:

- *scan.l* – лексер, который отвечает за распознавание идентификаторов и ключевых слов SQL
- *gram.y* – набор правил грамматики и действий, которые запускаются при каждом вызове файла

# Дерево запроса

Специальное внутреннее представление SQL запросов с полным его разбором по ключевым параметрам:

- Тип команды (SELECT, UPDATE, DELETE, INSERT)
- Список используемых отношений
- Целевое отношение, в которое будет записан результат
- Список полей (\* преобразуется в полный список всех полей)
- Список ограничений (которые указаны в WHERE)
- Список джоинов
- Прочие параметры, например, ORDER BY

# Система правил

Что делаете вы:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

Что происходит на самом деле:

```
CREATE VIEW myview (same column list as mytab);  
CREATE RULE "_RETURN" AS ON  
SELECT TO myview DO INSTEAD  
    SELECT * FROM mytab;
```

### 3. Система переписывания запросов

1. На вход принимается дерево запроса
2. Обращения к представлениям заменяются на таблицы с использованием системы правил
3. Система возвращает обновленное дерево запроса

## 4. Планировщик / оптимизатор

Задача *планировщика/оптимизатора* — построить наилучший план выполнения:

- Один и тот же SQL-запрос может быть выполнен разными способами
- Результат их выполнения будет идентичен
- На разные способы тратится разное число ресурсов

Результат работы планировщика – план запроса



## 5. Выполнение запроса

1. На выход принимает план (дерево) запроса
2. Обработчик рекурсивно обходит план запроса
3. При обращении к каждому узлу плана должны быть получены 1 или более строк, либо сообщение о том, что выдача строк завершена

# Жизненный цикл запроса

1. Создается подключение к СУБД. В СУБД отправляется запрос.
2. Парсер проверяет корректность синтаксиса запроса и создает дерево запроса.
3. Система переписывания запросов преобразует запрос.
4. Планировщик / оптимизатор создает план запроса.
5. Обработчик рекурсивно обходит план и получает строки.

# План запроса

Для оптимизации запросов очень важно понимать логику работы ядра PostgreSQL.

**Всё не так сложно.**

Описанное применимо к PostgreSQL 9.2 и выше.

# План запроса: задачи



научиться читать и понимать вывод команды EXPLAIN



понять, что же происходит в PostgreSQL при выполнении запроса



не сломать при этом мозг

# Back in ~~the USSR~~ time...

## **Жизненный цикл запроса:**

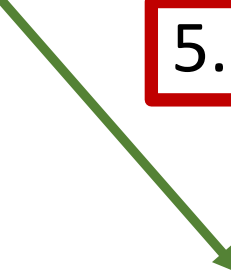
1. Создается подключение к СУБД. В СУБД отправляется запрос.
2. Парсер проверяет корректность синтаксиса запроса и создает дерево запроса.
3. Система переписывания запросов преобразует запрос.
4. Планировщик/оптимизатор создает план запроса.
5. Обработчик рекурсивно обходит план и получает строки.

Back in ~~the USSR~~ time...

**ШЛАК**



### Жизненный цикл запроса:

1. Создается подключение к СУБД. В СУБД отправляется запрос.
  2. Парсер проверяет корректность синтаксиса запроса и создает дерево запроса.
  3. Система переписывания запросов преобразует запрос.
  4. Планировщик/оптимизатор создает план запроса.
  5. Обработчик рекурсивно обходит план и получает строки.
- 

**КРУТО**

Back in ~~the USSR~~ time...

**Планировщик (planner)** – компонент PostgreSQL, пытающийся выработать наиболее эффективный способ выполнения запроса SQL.

В плане выполнения содержится информация о том, как будет организован просмотр таблиц, задействованных в запросе, сервером базы данных.

# Оператор EXPLAIN

- выводит план выполнения, генерируемый планировщиком PostgreSQL для заданного оператора.
- показывает, как будут сканироваться таблицы, затрагиваемые оператором — просто последовательно, по индексу и т.д.
- показывает, какой алгоритм соединения будет выбран для объединения считанных из таблиц строк
- показывает ожидаемую *стоимость* выполнения запроса
- **ОТСУТСТВУЕТ** в стандарте SQL



# Оператор EXPLAIN

```
EXPLAIN [ ( option [, ...] ) ] statement  
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

where *option* can be one of:

```
ANALYZE [ boolean ]  
VERBOSE [ boolean ]  
COSTS [ boolean ]  
BUFFERS [ boolean ]  
TIMING [ boolean ]  
FORMAT { TEXT | XML | JSON | YAML }
```

# Оператор ANALYZE

- собирает статистическую информацию о содержимом таблиц в базе данных и сохраняет результаты в системном каталоге pg\_statistic
- без параметров анализирует все таблицы в текущей базе данных
- если в параметрах передано имя таблицы, обрабатывает только заданную таблицу
- если в параметрах передан список имён столбцов, то сбор статистики запустится только по этим столбцам
- **ОТСУТСТВУЕТ** в стандарте SQL

# Оператор ANALYZE

```
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

# План запроса: тестовая таблица

```
CREATE TABLE foo (c1 integer, c2 text);  
  
INSERT INTO foo  
  SELECT  
    i  
    , md5(random())::text  
FROM  
  generate_series(1, 1000000) AS i;
```

# План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
```

Чтение данных из таблицы может выполняться несколькими способами. В нашем случае EXPLAIN сообщает, что используется Seq Scan — последовательное, блок за блоком, чтение данных таблицы.

# План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
```

Данные читаются из таблицы `foo`.

# План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
```

Что такое `cost`? Это не время, а некое сферическое в вакууме понятие, призванное оценить затратность операции.

- Первое значение `0.00` — затраты на получение первой строки.
- Второе — `18334.00` — затраты на получение всех строк.

# План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
```

`rows` — приблизительное количество возвращаемых строк при выполнении операции `Seq Scan`.

**Важно!** Сейчас никакие строки из таблицы не вычитываются. Совсем. Поэтому значение и *приблизительное*.



# План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo   (cost=0.00..18334.00 rows=1000000 width=37)
```

`width` — средний размер одной строки в байтах.

# План запроса: простая выборка

```
INSERT INTO foo
SELECT
    i
    , md5(random())::text
FROM
    generate_series(1, 10) AS i;
```

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo    (cost=0.00..18334.00 rows=1000000 width=37)
```

# План запроса: простая выборка

```
INSERT INTO foo
  SELECT
    i
    , md5(random())::text
  FROM
    generate_series(1, 10) AS i;
```

```
EXPLAIN SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo   (cost=0.00..18334.00 rows=1000000 width=37)
```

Как же так?



# План запроса: простая выборка

Значение `rows` не изменилось. Статистика по таблице старая. Для обновления статистики вызываем команду `ANALYZE`.

```
ANALYZE foo;  
EXPLAIN SELECT * FROM foo;  
----  
QUERY PLAN  
Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37)
```

Теперь `rows` отображает верное количество строк.

Всё, что мы видели выше в выводе команды `EXPLAIN` — только ожидания планировщика. Попробуем сверить их с результатами на реальных данных. Используем `EXPLAIN (ANALYZE)`.

# Откуда берется статистика?

```
SELECT relpages, reltuples
FROM pg_class
WHERE relname = 'foo';
```

```
----
relpages | reltuples
-----
8334      | 1000010
```

Статистика обновляется только при запуске `VACUUM` или `ANALYZE`

# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo   (cost=0.00..18334.10 rows=1000010 width=37)  
                  (actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

Такой запрос *будет исполняться реально*.

Если вы выполняете EXPLAIN (ANALYZE) для INSERT, DELETE или UPDATE, *ваши данные изменятся*.

Не забывайте про ROLLBACK 😊

# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37)
```

```
(actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

`actual time` — реальное время в миллисекундах, затраченное для получения первой строки и всех строк соответственно.

# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)  
                (actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

`rows` — реальное количество строк, полученных при Seq Scan.



# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)  
                (actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

loops — сколько раз пришлось выполнить операцию Seq Scan.

# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo   (cost=0.00..18334.10 rows=1000010 width=37)  
                  (actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

Planning time — время, потраченное планировщиком на построение плана запроса.

# План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
```

```
----
```

```
QUERY PLAN
```

```
Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)  
                (actual time=0.402..97.000 rows=1000010 loops=1)
```

```
Planning time: 0.042 ms
```

```
Execution time: 138.229 ms
```

Execution time — общее время выполнения запроса.

# Индексы

**Индексы** в PostgreSQL — специальные объекты базы данных, предназначенные в основном для ускорения доступа к данным. Это вспомогательные структуры: любой индекс можно удалить и восстановить заново по информации в таблице.

# План запроса: WHERE

```
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
```

```
----
```

```
QUERY PLAN
```

```
Seq scan on foo  (cost=0.00..12500.71 rows=416671 width=37)  
                (actual time=0.059..54.462 rows=333337 loops=3)
```

```
    Filter: (c1 > 500)
```

```
    Rows Removed by Filter: 510
```

```
Planning time: 0.175 ms
```

```
Execution time: 693.216 ms
```

# План запроса: WHERE + INDEX (много строк)

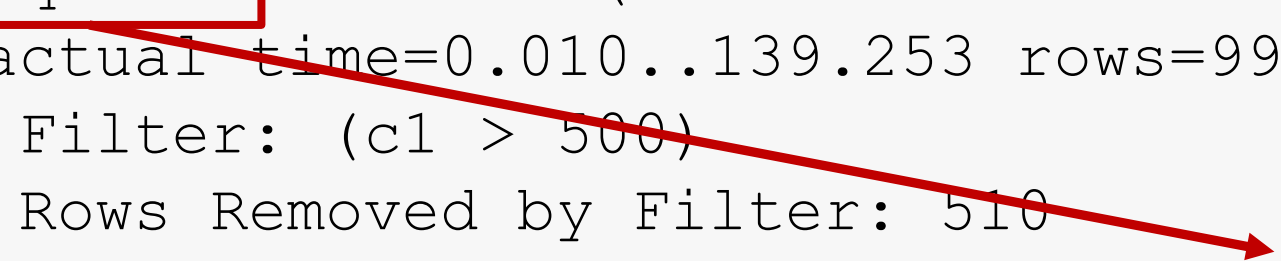
Добавим индекс:

```
CREATE INDEX ON foo (c1);  
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;  
----  
QUERY PLAN  
Seq Scan on foo (cost=0.00..20834.12 rows=999522 width=37)  
(actual time=0.010..139.253 rows=999500 loops=1)  
  Filter: (c1 > 500)  
  Rows Removed by Filter: 510  
Planning time: 0.096 ms  
Execution time: 180.288 ms
```

# План запроса: WHERE + INDEX (много строк)

Добавим индекс:

```
CREATE INDEX ON foo(c1);
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
----
QUERY PLAN
Seq Scan on foo (cost=0.00..20834.12 rows=999522 width=37)
(actual time=0.010..139.253 rows=999500 loops=1)
  Filter: (c1 > 500)
  Rows Removed by Filter: 510
Planning time: 0.096 ms
Execution time: 180.288 ms
```

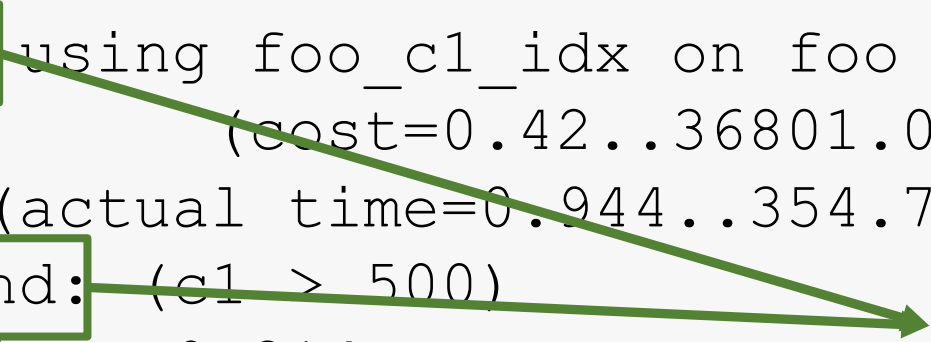


**Почему Seq Scan?**

# План запроса: WHERE + INDEX (много строк)

Запретим использовать Seq scan:

```
SET enable_seqscan TO off;  
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;  
----  
QUERY PLAN  
Index Scan using foo_c1_idx on foo  
      (cost=0.42..36801.06 rows=999522 width=37)  
      (actual time=0.944..354.736 rows=999500 loops=1)  
    Index Cond: (c1 > 500)  
Planning time: 0.314 ms  
Execution time: 396.225 ms
```




Стало лучше?



# План запроса: WHERE + INDEX (много строк)

Запретим использовать Seq scan:

```
SET enable_seqscan TO off;  
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;  
----  
QUERY PLAN  
Index Scan using foo_c1_idx on foo  
      (cost=0.42..36801.06 rows=999522 width=37)  
      (actual time=0.944..354.736 rows=999500 loops=1)  
    Index Cond: (c1 > 500)  
Planning time: 0.314 ms  
Execution time: 396.225 ms
```



Нет 😞

# План запроса: WHERE + INDEX (мало строк)

Возвращаем Seq scan и изменяем запрос:

```
SET enable_seqscan TO on;  
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 < 500;  
----  
QUERY PLAN  
Index Scan using foo_c1_idx on foo  
          (cost=0.42..24.96 rows=488 width=37)  
          (actual time=0.011..0.162 rows=509 loops=1)  
    Index Cond: (c1 < 500)  
Planning time: 0.148 ms  
Execution time: 0.201 ms
```

# План запроса: WHERE + INDEX (мало строк)

Возвращаем Seq scan и изменяем запрос:

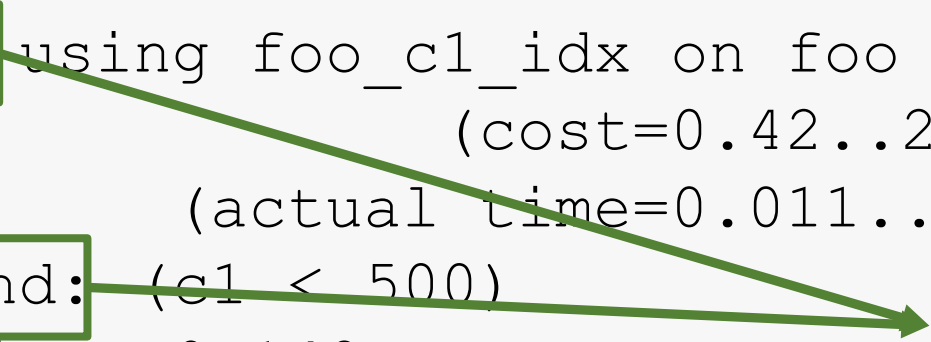
```
SET enable_seqscan TO on;
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 < 500;
----
```

QUERY PLAN

Index Scan using foo\_c1\_idx on foo  
(cost=0.42..24.96 rows=488 width=37)  
(actual time=0.011..0.162 rows=509 loops=1)

Index Cond: (c1 < 500)

Planning time: 0.148 ms  
Execution time: 0.201 ms



Сразу хорошо!

# План запроса: WHERE + INDEX (посложнее)

Добавим индекс по строке, перепишем запрос:

```
CREATE INDEX ON foo(c2);  
EXPLAIN SELECT * FROM foo WHERE c2 LIKE 'abcd%';  
-----  
QUERY PLAN  
Bitmap Heap Scan on foo (cost=4.58..55.20 rows=100 width=37)  
  Filter: (c2 ~~ 'abcd% '::text)  
    -> Bitmap Index Scan on foo_c2_idx1  
          (cost=0.00..4.55 rows=13 width=0)  
        Index Cond:  
          ((c2 ~>=~ 'abcd'::text) AND (c2 ~<~ 'abce'::text))
```

# План запроса: WHERE + INDEX (посложнее)

Добавим индекс по строке, перепишем запрос:

```
CREATE INDEX ON foo (c2);
```

```
EXPLAIN SELECT * FROM foo WHERE c2 LIKE 'abcd%';
```

```
----
```

```
QUERY PLAN
```

```
Bitmap Heap Scan on foo (cost=4.58..55.20 rows=100 width=37)
```

```
  Filter: (c2 ~~ 'abcd% '::text)
```

```
    -> Bitmap Index Scan on foo_c2_idx
```

```
          (cost=0.00..4.55 rows=13 width=0)
```

```
    Index Cond:
```

```
      ((c2 >= 'abcd'::text) AND (c2 < 'abce'::text))
```

**Что-то новое!**

# План запроса: WHERE + INDEX (последнее)

Попробуем выбирать не всё, а только поле фильтрации:

```
EXPLAIN SELECT c1 FROM foo WHERE c1 < 500;
```

```
----
```

```
QUERY PLAN
```

```
Index Only Scan using foo_c1_idx on foo
```

```
(cost=0.42..24.96 rows=488 width=4)
```

```
Index Cond: (c1 < 500)
```

# План запроса: WHERE + INDEX (последнее)

Попробуем выбирать не всё, а только поле фильтрации:

```
EXPLAIN SELECT c1 FROM foo WHERE c1 < 500;
```

```
----
```

```
QUERY PLAN
```

```
Index Only Scan using foo_c1_idx on foo  
      (cost=0.42..24.96 rows=488 width=4)  
    Index Cond: (c1 < 500)
```



**И опять!**

# План запроса: промежуточный итог

## Теперь знаем:

- Seq Scan — читается вся таблица.
- Index Scan — используется индекс для условий WHERE, читает таблицу при отборе строк.
- Bitmap Index Scan — сначала Index Scan, затем контроль выборки по таблице. Эффективно для большого количества строк.
- Index Only Scan — самый быстрый. Читается только индекс.



# План запроса: ORDER BY

Удалим все индексы:

```
DROP INDEX foo_c1_idx;  
DROP INDEX foo_c2_idx;
```

# План запроса: ORDER BY

```
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;
```

```
----
```

```
QUERY PLAN
```

```
Gather Merge (cost=63789.95..161019.97 rows=833342 width=37)
  (actual time=288.249..650.175 rows=1000010 loops=1)
    -> Sort (cost=62789.92..63831.60 rows=416671 width=37)
      (actual time=266.951..314.654 rows=333337 loops=3)
      Sort Key: c1
      Sort Method: external sort  Disk: 16888kB
      -> Parallel Seq Scan on foo
        (cost=0.00..12500.71 rows=416671 width=37)
        (actual time=0.059..54.462 rows=333337 loops=3)
```

```
Planning time: 0.175 ms
```

```
Execution time: 693.216 ms
```

# План запроса: LIMIT

```
EXPLAIN (ANALYZE, BUFFERS)
```

```
SELECT * FROM foo WHERE c2 LIKE 'ab%' LIMIT 10;
```

```
----
```

```
QUERY PLAN
```

```
Limit (cost=0.00..20.63 rows=10 width=37)
```

```
(actual time=0.186..0.577 rows=10 loops=1)
```

```
Buffers: shared hit=19
```

```
-> Seq Scan on foo (cost=0.00..20834.12 rows=10101 width=37)
```

```
(actual time=0.184..0.567 rows=10 loops=1)
```

```
Filter: (c2 ~~ 'ab% '::text)
```

```
Rows Removed by Filter: 2240
```

```
Buffers: shared hit=19
```

```
Planning time: 0.886 ms
```

```
Execution time: 0.691 ms
```

# План запроса: JOIN

Создадим вторую таблицу:

```
CREATE TABLE bar (c1 integer, c2 boolean);

INSERT INTO bar
  SELECT
    i
    , i % 2 = 1
  FROM
    generate_series(1, 500000) AS i;

ANALYZE bar;
```

# План запроса: JOIN

```
EXPLAIN (ANALYZE)
```

```
SELECT * FROM foo JOIN bar ON foo.c1 = bar.c1;
```

```
----
```

```
QUERY PLAN
```

```
Hash Join  (cost=15417.00..60081.14 rows=500000 width=42)
```

```
    (actual time=185.091..982.071 rows=500010 loops=1)
```

```
    Hash Cond: (foo.c1 = bar.c1)
```

```
    -> Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)
```

```
        (actual time=0.053..224.915 rows=1000010 loops=1)
```

```
    -> Hash  (cost=7213.00..7213.00 rows=500000 width=5)
```

```
        (actual time=182.530..182.530 rows=500000 loops=1)
```

```
        Buckets: 131072  Batches: 8  Memory Usage: 3282kB
```

```
        -> Seq Scan on bar  (cost=0.00..7213.00 rows=500000 width=5)
```

```
            (actual time=0.024..73.284 rows=500000 loops=1)
```

```
Planning time: 4.460 ms
```

```
Execution time: 1005.429 ms
```

# План запроса: JOIN

```
EXPLAIN (ANALYZE)
```

```
SELECT * FROM foo JOIN bar ON foo.c1 < bar.c1 LIMIT 1000;
```

```
----
```

```
QUERY PLAN
```

```
Limit (cost=0.00..56.72 rows=1000 width=42) (actual time=0.014..0.598 rows=1000  
loops=1)
```

```
  -> Nested Loop (cost=0.00..9454121337.10 rows=166668333333 width=42) (actual  
time=0.014..0.504 rows=1000 loops=1)
```

```
    Join Filter: (foo.c1 < bar.c1)
```

```
    Rows Removed by Join Filter: 1
```

```
      -> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual  
time=0.006..0.006 rows=1 loops=1)
```

```
        -> Materialize (cost=0.00..11667.00 rows=500000 width=5) (actual  
time=0.005..0.329 rows=1001 loops=1)
```

```
          -> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5)  
(actual time=0.003..0.079 rows=1001 loops=1)
```

```
Planning Time: 0.058 ms
```

```
Execution Time: 0.670 ms
```

# План запроса: JOIN + INDEX

```
CREATE INDEX ON foo(c1);
CREATE INDEX ON bar(c1);
EXPLAIN (ANALYZE)
    SELECT * FROM foo JOIN bar ON foo.c1 = bar.c1;
----
QUERY PLAN
Merge Join  (cost=1.33..39748.36 rows=500000 width=42)
    (actual time=0.014..428.590 rows=500010 loops=1)
    Merge Cond: (foo.c1 = bar.c1)
    ->  Index Scan using foo_c1_idx on foo
        (cost=0.42..34317.58 rows=1000010 width=37)
        (actual time=0.007..127.049 rows=500011 loops=1)
    ->  Index Scan using bar_c1_idx on bar
        (cost=0.42..15212.42 rows=500000 width=5)
        (actual time=0.005..112.125 rows=500010 loops=1)
Planning time: 0.435 ms
Execution time: 450.289 ms
```

# План запроса: методы соединения

- Nested loop (псевдокодом):

```
for i in first_table:  
    for j in second_table where second_table.i = i:  
        проверяем условия и формируем строку
```

- Hash join (псевдокодом):

```
строим хэш-таблицу из first_table  
for j in second_table:  
    if key_exists(hash(second_table.j)):  
        проверяем условия и формируем строку
```

- Merge join (псевдокодом):

```
сливаем две отсортированных first_table и second_table  
проверяем условия и формируем строку
```



# План запроса: Nested loop

## За:

- Очень дешевый
- Очень быстрый на небольших объемах
- Не требует много памяти
- Идеален для молниеносных запросов
- *Единственный умеет соединения не только по равенству*

## Против:

- Плохо работает для больших объемов данных

# План запроса: Hash join

## За:

- Не нужен индекс
- Относительно быстрый
- Может быть использован для FULL OUTER JOIN

## Против:

- Любит память
- Соединение только по равенству
- Не любит много значений в колонках соединения
- Велико время получения первой строки

# План запроса: Merge join

## **За:**

- Быстрый на больших и малых объемах
- Не требует много памяти
- Умеет OUTER JOIN
- Подходит для соединения более чем двух таблиц

## **Против:**

- Требуется отсортированные потоки данных, что подразумевает или индекс, или сортировку
- Соединение только по равенству

# Планировщик не дурак

```
DROP INDEX foo_c1_idx;  
DROP INDEX bar_c1_idx;
```

```
EXPLAIN (ANALYZE)  
SELECT * FROM foo, bar WHERE foo.c1 = bar.c1;
```

```
EXPLAIN (ANALYZE)  
SELECT * FROM foo CROSS JOIN bar WHERE foo.c1 = bar.c1;
```

```
EXPLAIN (ANALYZE)  
SELECT * FROM foo INNER JOIN bar ON foo.c1 = bar.c1;
```

Запустите локально и посмотрите на получившиеся планы. Есть разница?

# План запроса: что за бортом?

А ничего, дальше можно только смотреть и пробовать самостоятельно.

## **Где почитать:**

- [https://www.dalibo.org/media/understanding\\_explain.pdf](https://www.dalibo.org/media/understanding_explain.pdf)
- <http://langtoday.com/?p=229>
- <http://langtoday.com/?p=270>
- <https://habr.com/ru/post/203320/>

Вопросы?

**Задавайте.**