Стратегии и средства. Оптимизации. Планы запросов. Модернизация запросов на базе стоимостной оптимизации. Хорошие практики по работе с базой данных.

Почему запрос выполняется так долго? Почему не используются индексы? Наверное, все слышали об EXPLAIN в PostgreSQL. Но не так много тех, кто понимает, как его использовать. Сам длительное время не мог найти доступного для понимания учебника (плохо искал?).
Надеюсь, эта статья поможет желающим разобраться с этим замечательным инструментом.

Это авторская переработка материалов Understanding EXPLAIN от Guillaume Lelarge. Часть информации опущено, так что настоятельно рекомендую ознакомиться с оригиналом.

Для оптимизации запросов очень важно понимать логику работы ядра PostgreSQL. Постараюсь объяснить. На самом деле всё не так сложно. EXPLAIN выводит информацию, необходимую для понимания, что же делает ядро при каждом конкретном запросе. Будем рассматривать вывод команды EXPLAIN, параллельно разбираясь, что же происходит внутри PostgreSQL. Описанное применимо к PostgreSQL 9.2 и выше.

Наши задачи:

- научиться читать и понимать вывод команды EXPLAIN
- понять, что же происходит в PostgreSQL при выполнении запроса

Первые шаги

Тренироваться будем на ~~кошках~~ тестовой таблице в миллион строк

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

Попробуем прочитать данные

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
(1 row)

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

блоком, чтение данных таблицы `foo`.
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.
`rows` — приблизительное количество возвращаемых строк при выполнении операции `Seq Scan`. Это значение возвращает планировщик. В моём случае оно совпадает с реальным количеством строк в таблице.
`width` — средний размер одной строки в байтах.
Попробуем добавить 10 строк.

```
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)
(1 row)

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)
(1 row)

Теперь `rows` отображает правильное количество строк.

Что происходит при выполнении ANALYZE?

- Считывается определённое количество строк таблицы, выбранных случайным образом
- Собирается статистика значений по каждой из колонок таблицы:

Сколько строк будет считывать ANALYZE — зависит от параметра `default_statistics_target`.

Реальные данные

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

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.012..61.524 rows=1000010 loops=1)
Total runtime: 90.944 ms
(2 rows)

Такой запрос будет исполняется реально. Так что если вы выполняете EXPLAIN (ANALYZE) для INSERT, DELETE или UPDATE, ваши данные изменятся. Будьте внимательны! В таких случаях используйте команду ROLLBACK.
В выводе команды информации добавилось.
`actual time` — реальное время в миллисекундах, затраченное для получения первой строки и всех строк соответственно.
`rows` — реальное количество строк, полученных при `Seq Scan`.
`loops` — сколько раз пришлось выполнить операцию `Seq Scan`.
`Total runtime` — общее время выполнения запроса.

Кэш

Что происходит на физическом уровне при выполнениии нашего запроса? Разберёмся. Мой сервер поднят на Ubuntu 13.10. Используются дисковые кэши уровня ОС.
Останавливаю PostgreSQL, принудительно фиксирую изменения в файловой системе, очищаю кэши, запускаю PostgreSQL:

```
> sudo service postgresql-9.3 stop
> sudo sync
> sudo su -
# echo 3 > /proc/sys/vm/drop_caches
# exit
> sudo service postgresql-9.3 start
```

Теперь кэши очищены, пробуем выполнить запрос с опцией BUFFERS

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.525..734.754 rows=1000010 loops=1)
Buffers: shared read=8334
Total runtime: 1253.177 ms
(3 rows)

Таблица считывается частями — блоками. Кэш пуст. Таблица полностью считывается с диска. Для этого пришлось считать 8334 блока.
`Buffers: shared read` — количество блоков, считанное с диска.

Повторим последний запрос

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.173..693.000 rows=1000010 loops=1)

Buffers: shared hit=32 read=8302
Total runtime: 1208.433 ms
(3 rows)

`Buffers: shared hit` — количество блоков, считанных из кэша PostgreSQL.
Если повторите этот запрос несколько раз, то увидите, как PostgreSQL с каждым разом всё больше данных берёт из кэша. С каждым запросом PostgreSQL наполняет свой кэш.
Операции чтения из кэша быстрее, чем операции чтения с диска. Можете заметить эту тенденцию, отслеживая значение `Total runtime`.
Объём кэша определяется константой `shared_buffers` в файле `postgresql.conf`.

WHERE

Добавим в запрос условие

```
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
```

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=999522 width=37)
Filter: (c1 > 500)
(2 rows)

Индексов у таблицы нет. При выполнении запроса последовательно считывается каждая запись таблицы (`Seq Scan`). Каждая запись сравнивается с условием `c1 > 500`. Если условие выполняется, запись вводится в результат. Иначе — отбрасывается. `Filter` означает именно такое поведение.
Значение `cost`, что логично, увеличилось.
Ожидаемое количество строк результата — `rows` — уменьшилось.
В оригинале даются объяснения, почему `cost` принимает именно такое значение, а также каким образом рассчитывается ожидаемое количество строк.

Пора создать индексы.

```
CREATE INDEX ON foo(c1);
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
```

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=999519 width=37)
Filter: (c1 > 500)
(2 rows)

Ожидаемое количество строк изменилось. Уточнилось. В остальном ничего нового.
Что же с индексом?

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=999519 width=37) (actual time=0.572..848.895 rows=999500 loops=1)

Filter: (c1 > 500)
Rows Removed by Filter: 510
Total runtime: 1330.788 ms
(4 rows)

Отфильтровано только 510 строк из более чем миллиона. Пришлось считать более 99,9% таблицы.

Принудительно заставим использовать индекс, запретив 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..34623.01 rows=999519 width=37) (actual time=0.178..1018.045 rows=999500 loops=1)
Index Cond: (c1 > 500)
Total runtime: 1434.429 ms
(3 rows)

`Index Scan`, `Index Cond` вместо `Filter` — используется индекс `foo_c1_idx`. При выборке практически всей таблицы использование индекса только увеличивает `cost` и время выполнения запроса. Планировщик не глуп!

Не забываем отменить запрет на использование Seq Scan:

```
SET enable_seqscan TO on;
```

Изменим запрос:

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

QUERY PLAN
— Index Scan using foo_c1_idx on foo (cost=0.42..25.78 rows=491 width=37)
Index Cond: (c1 < 500)
(2 rows)

Тут планировщик решил использовать индекс.

Усложним условие. Используем текстовое поле.

```
EXPLAIN SELECT * FROM foo
        WHERE c1 < 500 AND c2 LIKE 'abcd%';
```

QUERY PLAN
— Index Scan using foo_c1_idx on foo (cost=0.42..27.00 rows=1 width=37)
Index Cond: (c1 < 500)
Filter: (c2 ~~ 'abcd%'::text)
(3 rows)

Как видим, используется индекс `foo_c1_idx` для условия `c1 < 500`. Для `c2 ~~ 'abcd%'::text` используется фильтр.
Обратите внимание, что в выводе результатов используется POSIX формат оператора LIKE.

Если в условии только текстовое поле:

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

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=100 width=37) (actual time=14.497..412.030 rows=10 loops=1)
Filter: (c2 ~~ 'abcd%'::text)
Rows Removed by Filter: 1000000
Total runtime: 412.120 ms
(4 rows)

Ожидаемо, `Seq Scan`.

Строим индекс по `c2`:

```
CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT * FROM foo
WHERE c2 LIKE 'abcd%';
```

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=100 width=37) (actual time=20.992..424.946 rows=10 loops=1)
Filter: (c2 ~~ 'abcd%'::text)
Rows Removed by Filter: 1000000
Total runtime: 425.039 ms
(4 rows)

Опять `Seq Scan`? Индекс не используется потому, что база у меня для текстовых полей использует формат UTF-8.
При создании индекса в таких случаях надо использовать класс оператора `text_pattern_ops`:

```
CREATE INDEX ON foo(c2 text_pattern_ops);
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))
(4 rows)

Ура! Получилось!
`Bitmap Index Scan` — используется индекс `foo_c2_idx1` для определения нужных

нам записей, а затем PostgreSQL лезет в саму таблицу: (`Bitmap Heap Scan`) -, чтобы убедиться, что эти записи на самом деле существуют. Такое поведение связано с версионностью PostgreSQL.

Если выбирать не всю строку, а только поле, по которому построен индекс

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

QUERY PLAN
— Index Only Scan using foo_c1_idx on foo (cost=0.42..25.78 rows=491 width=4)
Index Cond: (c1 < 500)
(2 rows)

`Index Only Scan` выполняется быстрее, чем `Index Scan` за счёт того, что не требуется читать строку таблицы полностью: `width=4`.

Итог

- 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.

ORDER BY

```
DROP INDEX foo_c1_idx;
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;
```

QUERY PLAN
— Sort (cost=117993.01..120493.04 rows=1000010 width=37) (actual time=571.591..651.524 rows=1000010 loops=1)
Sort Key: c1
Sort Method: external merge Disk: 45952kB
-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.007..62.041 rows=1000010 loops=1)
Total runtime: 690.984 ms
(5 rows)

Сначала производится `Seq Scan` таблицы `foo`. Затем сортировка `Sort`. В выводе команды EXPLAIN знак `->`указывает на иерархию действий (*node*). Чем раньше выполняется действие, тем с б**о**льшим отступом оно отображается.
`Sort Key` — условие сортировки.
`Sort Method: external merge Disk` — при сортировке используется временный файл на диске объёмом`45952kB`.

Прошу разбирающихся в теме разъяснить различия между `external merge` *и* `external sort`.

Проверим с опцией BUFFERS:

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

QUERY PLAN
— Sort (cost=117993.01..120493.04 rows=1000010 width=37) (actual time=568.412..652.308 rows=1000010 loops=1)
Sort Key: c1
Sort Method: external merge Disk: 45952kB
Buffers: shared hit=8334, temp read=5745 written=5745
-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.010..68.203 rows=1000010 loops=1)
Buffers: shared hit=8334
Total runtime: 698.032 ms
(7 rows)

Действительно, `temp read=5745 written=5745` — во временный файл было записано и прочитано 5745 блоков по 8Kb = 45960Kb. Операции с 8334 блоками были произведены в кэше.

Операции с файловой системой более медленные, чем операции в оперативной памяти.
Попробуем увеличить объём используемой памяти `work_mem`:

```
SET work_mem TO '200MB';
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;
```

QUERY PLAN
— Sort (cost=117993.01..120493.04 rows=1000010 width=37) (actual time=265.301..296.777 rows=1000010 loops=1)
Sort Key: c1
Sort Method: quicksort Memory: 102702kB
-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.006..57.836 rows=1000010 loops=1)
Total runtime: 328.746 ms
(5 rows)

`Sort Method: quicksort Memory: 102702kB` — сортировка целиком проведена в оперативной памяти.

Индекс:

```
CREATE INDEX ON foo(c1);
```

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

QUERY PLAN
— Index Scan using foo_c1_idx on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual time=0.023..126.076 rows=1000010 loops=1)
Total runtime: 153.452 ms
(2 rows)

Из действий осталось только `Index Scan`, что заметно отразилось на скорости выполнения запроса.

LIMIT

Удалим ранее созданный индекс.

```
DROP INDEX foo_c2_idx1;
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM foo WHERE c2 LIKE 'ab%';
```

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=100 width=37) (actual time=0.033..94.757 rows=3824 loops=1)
Filter: (c2 ~~ 'ab%'::text)
Rows Removed by Filter: 996186
Buffers: shared hit=8334
Total runtime: 94.924 ms
(5 rows)

Ожидаемо, используются `Seq Scan` и `Filter`.

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM foo WHERE c2 LIKE 'ab%' LIMIT 10;
```

QUERY PLAN
— Limit (cost=0.00..2083.41 rows=10 width=37) (actual time=0.037..0.607 rows=10 loops=1)
Buffers: shared hit=26
-> Seq Scan on foo (cost=0.00..20834.12 rows=100 width=37) (actual time=0.031..0.599 rows=10 loops=1)
Filter: (c2 ~~ 'ab%'::text)
Rows Removed by Filter: 3053
Buffers: shared hit=26
Total runtime: 0.628 ms
(7 rows)

Производится сканирование `Seq Scan` строк таблицы и сравнение `Filter` их с условием. Как только наберётся 10 записей, удовлетворяющих условию, сканирование закончится. В нашем случае для того, чтобы получить 10 строк результата пришлось прочитать не всю таблицу, а только 3063 записи, из них 3053 были отвергнуты (`Rows Removed by Filter`).
То же происходит и при `Index Scan`.

JOIN

Создадим новую таблицу, соберём для неё статистику.

```
CREATE TABLE bar (c1 integer, c2 boolean);
INSERT INTO bar
  SELECT i, i%2=1
  FROM generate_series(1, 500000) AS i;
ANALYZE bar;
```

Запрос по двум таблицам

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

QUERY PLAN
— Hash Join (cost=13463.00..49297.22 rows=500000 width=42) (actual time=87.441..907.555 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=0.008..67.951 rows=1000010 loops=1)
-> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=87.352..87.352 rows=500000 loops=1)
Buckets: 65536 Batches: 1 Memory Usage: 18067kB
-> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.007..33.233 rows=500000 loops=1)
Total runtime: 920.967 ms
(7 rows)

Сначала просматривается (`Seq Scan`) таблица `bar`. Для каждой её строки вычисляется хэш (`Hash`).
Затем сканируется `Seq Scan` таблица `foo`, и для каждой строки этой таблицы вычисляется хэш, который сравнивается (`Hash Join`) с хэшем таблицы `bar` по условию `Hash Cond`. Если соответствие найдено, выводится результирующая строка, иначе строка будет пропущена.
Использовано 18067kB в памяти для размещения хэшей таблицы `bar`.

Добавим индекс

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

QUERY PLAN
— Merge Join (cost=1.69..39879.71 rows=500000 width=42) (actual time=0.037..263.357 rows=500010 loops=1)
Merge Cond: (foo.c1 = bar.c1)
-> Index Scan using foo_c1_idx on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual time=0.019..58.920 rows=500011 loops=1)
-> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual time=0.008..71.719 rows=500010 loops=1)
Total runtime: 283.549 ms
(5 rows)

`Hash` уже не используется. `Merge Join` и `Index Scan` по индексам обеих таблиц дают впечатляющий прирост производительности.

LEFT JOIN:

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

QUERY PLAN
— Hash Left Join (cost=13463.00..49297.22 rows=1000010 width=42) (actual time=82.682..926.331 rows=1000010 loops=1)
Hash Cond: (foo.c1 = bar.c1)
-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.004..68.763 rows=1000010 loops=1)
-> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=82.625..82.625 rows=500000 loops=1)
Buckets: 65536 Batches: 1 Memory Usage: 18067kB
-> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.003..31.890 rows=500000 loops=1)
Total runtime: 950.625 ms
(7 rows)

`Seq Scan`?
Посмотрим, какие результаты будут, если запретить Seq Scan.

```
SET enable_seqscan TO off;
EXPLAIN (ANALYZE)
SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;
```

QUERY PLAN
— Merge Left Join (cost=0.85..58290.02 rows=1000010 width=42) (actual time=0.024..353.819 rows=1000010 loops=1)

Merge Cond: (foo.c1 = bar.c1)
-> Index Scan using foo_c1_idx on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual time=0.011..112.095 rows=1000010 loops=1)
-> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual time=0.008..63.125 rows=500010 loops=1)
Total runtime: 378.603 ms
(5 rows)

По мнению планировщика, использование индексов затратнее, чем использование хэшей. Такое возможно при достаточно большом объёме выделенной памяти. Помните, мы увеличивали `work_mem`?
Но, если память в дефиците, планировщик будет вести себя иначе:

```
SET work_mem TO '15MB';
SET enable_seqscan TO ON;
EXPLAIN (ANALYZE)
SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;
```

QUERY PLAN
— Merge Left Join (cost=0.85..58290.02 rows=1000010 width=42) (actual time=0.014..376.395 rows=1000010 loops=1)
Merge Cond: (foo.c1 = bar.c1)
-> Index Scan using foo_c1_idx1 on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual time=0.005..124.698 rows=1000010 loops=1)
-> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual time=0.006..66.813 rows=500010 loops=1)
Total runtime: 401.990 ms
(5 rows)

А как будет выглядеть вывод EXPLAIN при запрещённом `Index Scan`?

```
SET work_mem TO '15MB';
SET enable_indexscan TO off;
EXPLAIN (ANALYZE)
SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;
```

QUERY PLAN
— Hash Left Join (cost=15417.00..63831.18 rows=1000010 width=42) (actual time=93.440..712.056 rows=1000010 loops=1)
Hash Cond: (foo.c1 = bar.c1)
-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.008..65.901 rows=1000010 loops=1)
-> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=93.308..93.308 rows=500000 loops=1)
Buckets: 65536 Batches: 2 Memory Usage: 9045kB
-> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.007..33.718 rows=500000 loops=1)
Total runtime: 736.726 ms
(7 rows)

`cost` явно увеличился. Причина в `Batches: 2`. Весь хэш не поместился в память, его пришлось разбить на 2 пакета по 9045kB.