ПЛАН ЗАПРОСА

Лекция 10

Анализ плана запроса

- в том или ином виде анализ плана запроса (команды EXPLAIN и ANALYZE) есть почти в каждой СУБД
- мы рассматриваем анализ плана запроса в СУБД PostgreSQL

Что такое план запроса?

- запрос поступает в СУБД, планировщик строит план запроса (на самом деле несколько, но в итоге выбирает самый быстрый), по этому плану выполняется запрос.

- EXPLAIN помогает понять, что и где можно улучшить

1. Создание таблицы

- Создаётся таблица, заполняется случайными значениями

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

2. Первый запрос

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

2. Первый запрос

```
EXPLAIN SELECT * FROM foo;
                        QUERY PLAN
Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=36)
```

- Seq Scan - таблица читается последовательно, блок за блоком
- Количество строк отличается от реально добавленных, но это вопрос сбора статистики

Значения

- Cost - это не время, а некое понятие для оценки затратность операции. За 1 единицу cost принимается время чтения одной страницы в 8Кб
- Первое значение - 0.00 — затраты на получение первой строки.
- Второе — 18334.00 — затраты на получение всех строк
- Rows - приблизительное количество строк
- Width - средний размер одной строки в байтах (она может меняться)

ANALYZE

- Для обновления статистики нужно запустить команду ANALYZE
- Для команды ANALYZE можно указывать отдельную таблицу, а можно не указывать.

```
ANALYZE foo;

EXPLAIN SELECT * FROM foo;

                QUERY PLAN

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

Что делает команда ANALYZE?

- читает случайные строки в таблице (300*default_statistics_target)
- default_statistics_target - поле в настройках PostgreSQL (по умолчанию 100)
- вычисляется разная статистическая информация
- процент NULL-значений, средний размер строки, количество уникальных значений и т.д.
- наиболее популярные значения и их частота

width

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

- width - средний размер (в байтах) строки в конечном наборе данных
- чем меньше, тем лучше
- средний width - это среднее значение суммы всех столбцов

EXPLAIN (ANALYZE)

- EXPLAIN не выполняет запрос, он только производит планирование
- А вот если запустить запрос как EXPLAIN (ANALYZE), то запрос реально выполнится и выведет дополнительную информацию
- реальное время выполнения в миллисекундах (для первой строки и для всех строк)
- реальное количество строк
- количество циклов

EXPLAIN (ANALYZE)

EXPLAIN (ANALYZE) SELECT * FROM foo;

EXPLAIN (ANALYZE)

```
EXPLAIN (ANALYZE) SELECT * FROM foo;
                        QUERY PLAN
Seq Scan on foo  (cost=0.00..18334.00 rows=1000000 width=37)
(actual time=0.015..50.096 rows=1000000 loops=1)
Planning time: 0.036 ms
Execution time: 66.856 ms
```

EXPLAIN (ANALYZE)

- чаще всего лучше наверняка знать, как выполняется запрос, чтобы узнать реальное время и реальное число строк
- нужно быть осторожным с EXPLAIN (ANALYZE) и командами модификации, потому что запрос реально выполняется и данные изменяются. Поэтому, чтобы данные оставались неизменными, их нужно оборачивать в транзакции

Добавим условие WHERE

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

Добавим условие WHERE

```
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
                        QUERY PLAN
Seq Scan on foo  (cost=0.00..20834.00 rows=999499 width=37)
Filter: (c1 > 500)
```

- появляется новая строчка Filter

EXPLAIN с WHERE

- стоимость выше
- меньше строк
- больше работы из-за фильтрации
- так как индекса нет, то СУБД приходится сканировать всю таблицу, и сравнивать столбец
- стоимость выше, так как появились дополнительные расходы на сравнение значения в столбце, это дольше, чем просто прочитать строку

Поможет ли индекс?

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

Поможет ли индекс?

```
CREATE INDEX ON foo(c1);
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
                QUERY PLAN
Seq Scan on foo  (cost=0.00..20834.00 rows=999489 width=37)
Filter: (c1 > 500)
```

- Индекс не помог

Почему так получилось?

- многим кажется, что индекс всегда помогает, если использовать его на поле в WHERE. Однако, это может быть и не так
- в текущей таблице 1кк строк, а пропускаются условием лишь 500
- СУБД "дешевле" прочитать всю таблицу и отфильтровать (использовать CPU), чем использовать индекс

Запрос в реальном запуске

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

Запрос в реальном запуске

```
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
                                QUERY PLAN
Seq Scan on foo  (cost=0.00..20834.00 rows=999489 width=37)
(actual time=0.052..74.723 rows=999500 loops=1)
Filter: (c1 > 500)
Rows Removed by Filter: 500
Planning time: 0.122 ms
Execution time: 91.520 ms
```

Новые данные

- Rows Removed by Filter: 500
- новая строчка - Rows removed by Filter: 500

(отображается только при ANALYZE)

- приходится прочитать 99.95% таблицы

Заставить использовать индекс

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

- лучше никогда не отключать возможность последовательного сканирования, потому что в случаях, когда нет индекса, это единственная возможность добыть данные

Принудительный индекс

- Index Scan using foo_c1_idx on foo (cost=0.42..36800.48 rows=999489 width=37) (actual time=0.038..111.394 rows=999500 loops=1)
- Index Cond: (c1 > 500)
- Planning Time: 0.079 ms
- Execution Time: 127.295 ms (а было 91ms)

Принудительный индекс

- работает, но медленно
- планировщик бы прав
- здесь индекс не даёт никакого выигрыша

Поменяем условие

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

Поменяем условие

- EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 < 500;

Index Scan using foo_c1_idx on foo (cost=0.42..25.37 rows=511 width=37) (actual time=0.007..0.063 rows=499 loops=1)
Index Cond: (c1 < 500)
Planning time: 0.074 ms
Execution time: 0.082 ms

Теперь выбирается индекс

- Планировщик выбрал индекс - Index Scan
- новая строчка Index Cond
- Index Cond: (c1 < 500)
- планировщик выбирает сканирование по индексу, потому что читается малое количество строк (только 499 из 1кк)

Усложняем фильтр

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

Усложняем фильтр

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

```
Index Scan using foo_c1_idx on foo (cost=0.42..26.65 rows=1 width=37)
Index Cond: (c1 < 500)
Filter: (c2 ~~ 'abcd%'::text)
```

- уже знакомые Index Cond и Filter

Пробуем фильтр только по c2

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

Пробуем фильтр только по c2

```
EXPLAIN (ANALYZE)
SELECT * FROM foo WHERE c2 LIKE 'abcd%';
Seq Scan on foo  (cost=0.00..20834.00 rows=100 width=37)
(actual time=21.435..134.153 rows=15 loops=1)
Filter: (c2 ~~ 'abcd%'::text)
Rows Removed by Filter: 999985
Total runtime: 134.198 ms
```

Пробуем фильтр только по c2

- на c2 нет индекса, поэтому PostgreSQL использует последовательное сканирование
- только 15 строчек удовлетворяют условию
- индекс может помочь ускорить запрос

Добавим индекс на c2

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

Добавим индекс на c2

```
CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT * FROM foo
WHERE c2 LIKE 'abcd%';
Seq Scan on foo  (cost=0.00..20834.00 rows=100 width=37)
(actual time=22.189..143.467 rows=15 loops=1)
Filter: (c2 ~~ 'abcd%'::text)
Rows Removed by Filter: 999985
Total runtime: 143.512 ms
```

Добавим индекс на c2

- добавили индекс, но он не используется
- проблема в том, что этот индекс не подходит
- c2 хранит значения в кодировке UTF8
- когда локаль базы данных не стандартная «C», нужно указывать класс оператора сравнения строк
- классы операторов text_pattern_ops, varchar_pattern_ops и bpchar_pattern_ops поддерживают индексы-B-деревья для типов text, varchar и char, соответственно
- https://postgrespro.ru/docs/postgrespro/12/indexes-opclass

Изменим индекс

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

Изменим индекс

```
CREATE INDEX ON foo(c2 text_pattern_ops);
EXPLAIN SELECT * FROM foo WHERE c2 LIKE 'abcd%';
Bitmap Heap Scan on foo  (cost=7.29..57.91 rows=100 width=37)
Filter: (c2 ~~ 'abcd%'::text)
 ->  Bitmap Index Scan on foo_c2_idx1
        (cost=0.00..7.26 rows=13 width=0)
        Index Cond: ((c2 ~>=~ 'abcd'::text)
                    AND (c2 ~<~ 'abce'::text))
```

- Мог получиться другой план запроса с Index Scan

Новые данные

- Bitmap Heap Scan – похоже на последовательное сканирование, но вместо посещения всех страниц, посещает только те, что были выбраны bitmap index scan
- Bitmap Index Scan – ещё один способ использования индекса. Получает весь индекс целиком и создаёт битовую карту TID кортежей. Условно, получает список страниц со смещением внутри страницы

Покрывающий индекс

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

Покрывающий индекс

EXPLAIN SELECT c1 FROM foo WHERE c1 < 500;

Index Only Scan using foo_c1_idx on foo (cost=0.42..25.02 rows=491 width=4)

Index Cond: (c1 < 500)

- вся нужная информация хранится в индексе и используется только индекс
- Index Only Scan – используется, когда доступен покрывающий индекс

Основные варианты сканирования

- Sequential Scan – последовательное сканирование всей таблицы
- Index Scan – чтение индекса для отсеивания по условию WHERE
- Bitmap Index Scan – почти то же самое, но сначала читается индекс, а потом таблица
- Index Only Scan – для покрывающего индекса