EXPLAIN, оптимизация JOIN, подзапросов и индексов

План выполнения запроса (EXPLAIN)

Что такое EXPLAIN

EXPLAIN - это команда SQL, которая показывает план выполнения запроса, не выполняя его фактически. Это основной инструмент для анализа производительности запросов.

Синтаксис в PostgreSQL

```
EXPLAIN [ANALYZE] [VERBOSE] ваш_запрос;
```

- `ANALYZE` - выполняет запрос и показывает фактическое время выполнения
- `VERBOSE` - добавляет дополнительную информацию

Пример в PostgreSQL

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE customer_id = 100;
```

Результат показывает:

- Тип сканирования (Seq Scan, Index Scan и т.д.)
- Оценочное и фактическое время выполнения
- Количество строк
- Использование индексов

Ключевые элементы плана, на которые нужно обращать внимание

1. **Тип операции (Node Type)** - определяет, что именно делает данный шаг
2. **Стоимость (Cost)** - состоит из двух чисел:
    - Первое число - начальная стоимость (стоимость до получения первой строки)
    - Второе число - полная стоимость (стоимость до получения всех строк)
3. **Количество строк (Rows)** - оценка количества строк, которые вернет операция
4. **Размер строки (Width)** - средний размер одной строки в байтах
5. **Фактические метрики (при EXPLAIN ANALYZE)**:

- Actual time - реальное время выполнения в миллисекундах
- Actual rows - реальное количество строк
- Loops - сколько раз выполнялась операция

Типы сканирования таблиц

1. Sequential Scan (Seq Scan) - Последовательное сканирование

Что означает: Полное чтение всей таблицы "строка за строкой" с диска или памяти.

Когда используется:

- Когда нет подходящего индекса
- Когда нужно прочитать большую часть таблицы (обычно >5-10% строк)
- Для очень маленьких таблиц

Пример (PostgreSQL):

```
EXPLAIN SELECT * FROM customers;
```

```
Seq Scan on customers  (cost=0.00..15.00 rows=1000 width=36)
```

Проблемы: Может быть медленным для больших таблиц.

2. Index Scan - Сканирование по индексу

Что означает: Использование индекса для поиска строк, затем обращение к таблице за остальными данными.

Когда используется:

- Когда условие WHERE использует индексированное поле
- Когда нужен доступ к небольшому проценту строк

Пример (PostgreSQL):

```
EXPLAIN SELECT * FROM customers WHERE id = 100;
```

```
Index Scan using customers_pkey on customers  (cost=0.15..8.17 rows=1 width=36)
  Index Cond: (id = 100)
```

Преимущества: Быстрый доступ к конкретным строкам.

3. Index Only Scan - Только индексное сканирование

Что означает: Данные полностью берутся из индекса, без обращения к таблице.

Когда используется:

- Когда все запрашиваемые поля содержатся в индексе
- В PostgreSQL при "visibility map", показывающей какие строки видны

Пример (PostgreSQL):

```
EXPLAIN SELECT id FROM customers;
```

```
Index Only Scan using customers_pkey on customers  (cost=0.15..30.15 rows=1000 width=4)
```

Преимущества: Самый быстрый тип доступа.

4. Bitmap Heap Scan + Bitmap Index Scan

Что означает:

1. Bitmap Index Scan - создает битовую карту подходящих строк из индекса
2. Bitmap Heap Scan - использует битовую карту для эффективного чтения строк из таблицы

Когда используется:

- Для условий с промежуточной селективностью (несколько процентов строк)
- При комбинации нескольких условий через AND/OR

Пример (PostgreSQL):

```
EXPLAIN SELECT * FROM customers WHERE age > 30 AND status = 'active';
```

```
Bitmap Heap Scan on customers  (cost=12.34..45.67 rows=100 width=36)
  Recheck Cond: ((age > 30) AND (status = 'active'::text))
  ->  Bitmap Index Scan on idx_customers_age_status  (cost=0.00..12.34 rows=100 width=0)
        Index Cond: ((age > 30) AND (status = 'active'::text))
```

Преимущества: Эффективен для многоколоночных условий.

Анализ операций соединения (JOIN)

Типы соединений в плане выполнения

1. **Nested Loop**:
    - Для каждой строки внешней таблицы ищет совпадения во внутренней
    - Эффективен при маленьких таблицах или когда есть индексы
2. **Hash Join**:
    - Строит хеш-таблицу по одной таблице, затем сканирует другую
    - Эффективен для средних и больших таблиц без индексов
3. **Merge Join**:
    - Сортирует обе таблицы по ключу соединения и затем сливает
    - Эффективен когда данные уже отсортированы или есть подходящие индексы

Оптимизация JOIN

Основные принципы оптимизации JOIN

1. **Порядок соединения таблиц** - меньшие таблицы следует соединять первыми
2. **Использование индексов** - соединение должно выполняться по индексированным полям
3. **Тип соединения** - выбор между INNER, LEFT, RIGHT JOIN
4. **Фильтрация перед соединением** - уменьшение количества строк до JOIN

Пример оптимизации в PostgreSQL

Неоптимизированный запрос:

```
SELECT *
FROM large_table l
JOIN small_table s ON l.id = s.large_id
WHERE l.some_condition = 'value';
```

Оптимизированный:

```
SELECT *
FROM (SELECT * FROM large_table WHERE some_condition = 'value') l
JOIN small_table s ON l.id = s.large_id;
```

Оптимизация подзапросов

Проблемы подзапросов

1. **Коррелированные подзапросы** - выполняются для каждой строки внешнего запроса
2. **Избыточные вычисления** - повторные вычисления одного и того же
3. **Ограничения оптимизатора** - не все подзапросы могут быть хорошо оптимизированы

Оптимизация в PostgreSQL

Неоптимизированный:

```
SELECT name,
       (SELECT COUNT(*) FROM orders WHERE orders.customer_id = customers.id)
FROM customers;
```

Оптимизированный:

```
SELECT c.name, o.order_count
FROM customers c
LEFT JOIN (SELECT customer_id, COUNT(*) as order_count
           FROM orders GROUP BY customer_id) o
ON c.id = o.customer_id;
```

Практические советы по анализу плана

1. **Ищите самые дорогие операции** - обычно это операции с наибольшей стоимостью (cost)
2. **Проверяйте расхождения между оценками и реальностью** - большая разница между rows и actual rows указывает на проблемы со статистикой
3. **Обращайте внимание на отсутствие индексов** - Seq Scan для больших таблиц часто требует добавления индекса
4. **Следите за операциями в памяти** - операции типа "Hash" требуют много памяти
5. **Проверяйте порядок соединений** - большие таблицы должны соединяться после фильтрации

Пример полного анализа

Рассмотрим запрос:

```
EXPLAIN ANALYZE
SELECT c.name, COUNT(o.id)
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE c.status = 'active'
GROUP BY c.name;
```

Пример вывода:

```
HashAggregate  (cost=125.50..127.50 rows=200 width=40) (actual time=3.456..3.478 rows=180 loop
  Group Key: c.name
  ->  Hash Join  (cost=45.60..115.25 rows=1025 width=36) (actual time=1.234..2.987 rows=1050 l
      Hash Cond: (o.customer_id = c.id)
    ->  Seq Scan on orders o  (cost=0.00..35.00 rows=1000 width=8) (actual time=0.012..0.456 r
    ->  Hash  (cost=38.25..38.25 rows=300 width=36) (actual time=1.210..1.210 rows=300 loops=1
      ->  Seq Scan on customers c  (cost=0.00..38.25 rows=300 width=36) (actual time=0.010..0.
                Filter: (status = 'active'::text)
                Rows Removed by Filter: 700
```

Пошаговый анализ:

1. Сначала фильтруются customers по статусу (Seq Scan + Filter)
2. Результат хешируется (Hash)
3. Сканируются все заказы (Seq Scan)
4. Выполняется соединение по customer_id (Hash Join)
5. Результаты агрегируются по имени (HashAggregate)

Потенциальные проблемы:

- Seq Scan по orders - возможно, нужно добавить индекс
- Фильтрация customers удаляет 700 из 1000 строк - возможно, стоит добавить индекс по status