PostgreSQL: практические примеры оптимизации SQL-запросов

Фролков Иван
Postgres Professional

HL 2016 HighLoad++

Профессиональная конференция разработчиков высоконагруженных систем

Эффективность

- Что такое «эффективный запрос»?
  - Быстрый? Но как? Время? Первой строки? Всего запроса?
  - Ввод-вывод? Процессор? Блокировки?
- Как мы можем сравнить эффективность?
  - Время выполнения
  - Количество операций ввода-вывода

Выполнение запроса

- PostgreSQL — Executor
- Можно попросить реальный план выполнения

```
select * from acc.ledger l where l.reference like 'IN/%'
```

```
Index Scan using ledger_pkey on ledger l  (cost=0.56..8.58
rows=8879325 width=162) (actual time=0.052..2573.333 rows=8880000
loops=1)
  Index Cond: ((reference >= 'IN/'::text) AND (reference <
'IN0'::text))
  Filter: (reference ~~ 'IN/%'::text)
  Buffers: shared hit=149021 read=168134
Planning time: 0.303 ms
Execution time: 2872.353 ms
```

Общий принцип

Чем меньше, тем лучше!

Данных

Индексов

Ввода-вывода

Страниц

Блокировок

Latches

Еще случаи

- uuid text — 32 байта
- uuid uuid — 16 байт
- На десятке таких колонок будет совсем интересно

Еще случаи

- uuid text — 32 байта
- uuid uuid — 16 байт
- На десятке таких колонок будет совсем интересно
- Жадничайте!

И еще

- ```
  select … from
  t1 join t2 join t2
  where
  ? in (t1.col, t2.col, t3.col)
  ```
- Такое условие можно вычислить только после соединения.
  ```

Индексы

- Все тот же принцип — чем меньше, тем лучше
  - Меньше индекс
  - Меньше индексов
- Иногда можно и вообще без индексов

Индексы btree

- Индекс — это отсортированная последовательность
- `(usr_id)`
- `(usr_id, added)`
  - Если оба реально используются, подумайте, нужны ли оба сразу — оптимизатор выбирает лучший индекс для запроса, а не для всего приложения

Порядок строк в индексе

- `(usr_id)` — `usr_id` равно/больше/меньше
- `(usr_id,added)` — `usr_id` равно/больше/меньше
  - `usr_id` равно, `added` равно/больше/меньше
  - НЕ РАБОТАЕТ (почти) -
    `added` равно/больше/меньше

Индексы — LIKE

- Для LIKE индекс используется для поиска по префиксу — `LIKE 'str%'`
- Не работает для `LIKE '%str'`
- Внимание — параметры!
  - Тонкий момент в PostgreSQL

Покрытие индексом

- Все колонки есть в индексе
- Меньше обращений к страницам
- Меньше ввод-вывод
- Меньше latches/buffer pin
- Больше индексов/больше индекс

Пример

```
create table ios(
 id int primary key,
 val text)

insert into ios select n, repeat('X', n%100) from
generate_series(1,1000000) as gs(n)
```

```
explain(analyze, verbose,buffers)
select count(val) from ios where id between 1 and 100000

explain(analyze, verbose,buffers)
select count(id) from ios where id between 1 and 100000
```

План 1

```
Aggregate  (cost=3998.45..3998.46 rows=1 width=8)
(actual time=22.241..22.242 rows=1 loops=1)
  Output: count(val)
  Buffers: shared hit=816
  ->  Index Scan using ios_pkey on public.ios
…
        Output: id, val
…
        Buffers: shared hit=816
```

План Бэ

```
Aggregate  (cost=3347.30..3347.31 rows=1 width=8)
(actual time=16.804..16.804 rows=1 loops=1)
  Buffers: shared hit=277
  ->  Index Only Scan using ios_pkey on public.ios
…
        Heap Fetches: 0
        Buffers: shared hit=277
```

Сравнение при параллельном выполнении

- 8 клиентов
  - Обычный доступ — 220.709118 tps
  - Index-only scan - 336.434901 tps

Покрытие индексом

- PostgresPro — `INCLUDING`
- Покрытие индексом — предпоследний способ повысить производительность
- Почему плохо
  - На каждый запрос делать индекс — это ж сколько их будет?
  - Что делать, если запрос поменялся?

Методы соединения

```
select * from first, second
where first.key=second.key
```

Методы соединения

```
select * from first, second
where first.key=second.key
```

```
select * from first, second
where first.key<>second.key
```

Методы соединения

```
select * from first, second
where first.key=second.key
```

```
select * from first, second
where first.key<>second.key
```

```
select * from first, second
where exists(select * from third where
third.first_key=first.key and
third.second_key=second.key)
```

Методы соединения

- **Nested loops**
  - `for i in first_table`
    - `For j in second_table where second_table.i=i`
      `проверяем условия и формируем строку`

Методы соединения

- Nested loops
  - `for i in first_table`
    - `For j in second_table where second_table.i=i`
      проверяем условия и формируем строку
- **Hash join**
  - **Строим хэш-таблицу из first_table**
    - `for j in second_table`
      `if key_exists(hash(second_table.j))`
      -проверяем условия и формируем строку
  - **Что делать, если таблица не помещается в память?**

Методы соединения

- Nested loops
  - `for i in first_table`
    - `For j in second_table where second_table.i=i`
      проверяем условия и формируем строку
- Hash join
  - 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**
  - Сливаем две отсортированных first_table & second_table
    - *проверяем условия и формируем строку*

Методы соединения

- Nested loops
  - 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

Методы соединения

- 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

Методы соединения

- Merge join
  - За
    - Быстрый на больших и малых объемах
    - Не требует много памяти
    - Умеет OUTER JOIN
    - Подходит для соединения более чем двух таблиц
  - Против
    - Требует отсортированные потоки данных, что подразумевает или индекс, или сортировку
    - Соединение только по равенству

Про Postgres

- Не умеет full outer join с соединением не по равенству
- Вот только что-то никто не жаловался :-)

Статистика

- В PostgreSQL — pg_statistics или на ее основе представление pg_stats
- Статистика — ключевой фактор для работы оптимизатора
- Проблемы — пары-тройки-четверки колонок
- Oracle — умеет. А вот PostgreSQL — нет :-(

Типовые проблемы

- Плохая схема БД
  - Объемы!
    - Лишние данные
    - Лишние индексы
    - Отсутствие нужных индексов
    - Неверные типы
  - Необходимость писать сложные запросы
- Бездумное использование ORM

Бездумное использование ORM

- Вообще говоря, я его не люблю
  - Но мало ли что я не люблю. А народу вот нравится
- Типовой запрос:
  - ```
    select distinct <от десятков до сотен колонок>
    from table1 left outer join table2 on …
    left outer join table3 on …
    left outer join table4 on …
    where table4.col='value'
    order by table1.id
    limit 100
    offset 20000
    ```
    ```

ORM-запрос. Что тут плохо

- **DISTINCT**
  - **Если вы не можете точно сказать, зачем вы используете DISTINCT, то у вас проблемы**

ORM-запрос. Что тут плохо

- DISTINCT
  - Если вы не можете точно сказать, зачем вы используете DISTINCT, то у вас проблемы
- **LEFT OUTER JOIN**
  - **Бьет по рукам оптимизатору, строго задавая порядок соединения**
  - **Более того, условия во WHERE делают внешнее соединение ненужным**

ORM-запрос. Что тут плохо

- DISTINCT
  - Если вы не можете точно сказать, зачем вы используете DISTINCT, то у вас проблемы
- LEFT OUTER JOIN
  - Бьет по рукам оптимизатору, строго задавая порядок соединения
  - Более того, условия во WHERE делают внешнее соединение ненужным
- **LIMIT/OFFSET почти всегда плохо**

Что делать?

- Четко определиться, какую бизнес-задачу решает запрос. Возможно, после этого необходимость в нем отпадет
- Разобраться с ORM
  - Выбирать только то, что нужно
  - Постараться перейти к INNER JOIN
  - Постараться избавиться от LIMIT/OFFSET

Выводы

- Чем меньше, тем лучше
- Знайте ваши данные