ПОСТРОЕНИЕ И ОПТИМИЗАЦИЯ ЗАПРОСОВ SQL

Денис ЛУПАШКО

Департамент Программной Инженерии и Автоматики, группа TI-192 F/R, Факультет Вычислительной Техники, Информатики и Микроэлектроники,
Технический Университет Молдовы, Кишинев, Республика Молдова

Автор корреспондент: Денис ЛУПАШКО, e-mail: denis.lupasco@isa.utm.md

Научный руководитель: Дориан САРАНЧУК, DISA, FCIM, UTM

Аннотация: *Данная статья посвящена решению проблемы высокопроизводительных и времязатратным запросов к реляционным базам данных SQL, а также инструментам для анализа sql запросов.*

Ключевые слова: *sql, оптимизация sql запросов, построение запросов sql.EXPLAIN.*

Введение:

Работая программным инженером я часто сталкиваюсь с высокопроизводительными и неэффективными запросами к базе данных и необходимостью повысить их эффективность и время исполнения. В этой статье я хочу поделиться способами поиска проблем и их решением.

1. Построения эффективных sql запросов

Для того чтобы написанные вами запросы к базе данных были производительными надо понимать что же делает их не производительными и строить свои запросы так чтобы избегать бутылочных горлышек замедляющих ваш запрос [1].

- **Извлекайте только необходимые данные -** избегайте '**SELECT ***' , если в своем запросе используете 'EXISTS' используйте константу в операторе '**SELECT**' этого подзапроса вместо и если возможно избавьтесь от подзапросов используя оператор **'INNER JOIN'**
- **Избегайте оператора 'DISTINCT'-** оператор используется для возврата только различных значений, хорошо обдумайте есть ли в нем необходимость чтобы получить результат который вам необходим.
- **Ограничьте результаты -** используйте '**TOP', 'LIMIT'** чтобы ограничить результат запроса и тем самым увеличить его скорость.
- **Замените оператор OR вашего запроса на IN** если вы не используете индексы в вашей базе данных на сравниваем ом столбце.
- **Не используйте UNION -** нужно понимать что используя этот оператор вы просматриваете одну и ту же таблицу дважды всегда старайтесь вместо **'UNION'** использовать '**OUTER JOIN'**
- **Оператор 'NOT' -** также не использует индексы лучше вместо него использовать '<, <>, !>'
- **Оператор 'AND'** - так же как 'NOT' и 'OR' не использует индексы и является ресурсоемким, рекомендуется вместо него использовать оператор **'BETWEEN'.**
- **Операторы 'ANY', 'ALL' -** с этими операторами нужно быть осторожным так как при включении их в запрос индексы не будут использоваться и если вам необходимы индексы лучше использовать альтернативу 'MIN' или **'MAX'** но функции агрегации над многими строками могут занимать много времени так что в данном случае все зависит от желаемого результата.
- В тех случаях когда столбец используется в вычислениях или скалярных функциях индекс не используется например вместо **'where birth_year + 2 = 2002'** лучше написать запрос без вычислений **'where birth_year = 2000'**

- **Предпочитайте обходиться без оператора ‘HAVING’-** этот оператор используется вместе с оператором **‘GROUP BY’** чтобы ограничить результат запроса но является более ресурсозатратным чем **‘WHERE’**

Выше были представлены базовые антипаттерны но даже их достаточно чтобы справиться с низкой производительностью и большим временем исполнения SQL запросов в большинстве случаев.

2. Инструменты для оптимизация SQL запросов

На практике часто встречаются SQL запросы в которых иногда бывает крайне сложно найти проблему медленного исполнения, а порой и не возможно, здесь в игру вступают различные инструменты для анализа времени и плана выполнения всех частей запроса которые помогают в поиске самых медленных и неэффективных частей запроса. На моей практике самый большой запрос был около 700 строк и время выполнения составляло минуту, мне удалось снизить время выполнения до 20 секунд используя встроенный инструмент PostgreSQL ‘EXPLAIN’ команда возвращает план выполнения запроса, похожие команды или функции есть во многих базах данных но я буду пользоваться именно данной командой так как ее я применял для решения большинства проблем связанных с производительностью [2].

Чтобы использовать команду ‘EXPLAIN’ просто укажите ее перед запросом, ниже приведен синтаксис [3]

```
EXPLAIN [ ( option [, ...] ) ] sql_statement;
```

В качестве ‘option’ может быть использовано семь вариантов но чаще всего используется ‘ANALYZE’ пример результата выполнения:

```
EXPLAIN analyze SELECT * FROM test WHERE i < 100000;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------

 Bitmap Heap Scan on test  (cost=4.37..39.99 rows=10 width=8)
(actual time=0.023..0.043 rows=10 loops=1)
   Recheck Cond: (i < 100000)
   Heap Blocks: exact=9
   ->  Bitmap Index Scan on i1  (cost=0.00..4.37 rows=10 width=0)
(actual time=0.011..0.011 rows=10 loops=1)
         Index Cond: (i < 100000)
 Planning Time: 0.182 ms
 Execution Time: 0.070 ms
```

анализируя результат данной функции можно понять на какую часть операции тратится больше всего времени и изменить ее используя советы приведенные в части 1. Схожими инструментами обладает большинство популярных на сегодняшний день реляционных баз данных изучив документацию по которым вы смело можете применять ее в действии.

Заключение

В заключении хотелось бы отметить что нет универсального метода для решения всех проблем связанных с эффективностью sql запросов но есть комплекс мер и подходов основные из которых были рассмотрены в рамках данной статьи которые позволят писать запросы эффективней а также оптимизировать уже существующие запросы.

Библиография

1. Руководство по SQL: Как лучше писать запросы (Часть 1) - [online] [дата обращения 02.03.2023], Доступно: https://habr.com/ru/post/465547/
2. Mihail Curchi, “PostgreSQL – Explaining the EXPLAIN Command” - [online] [дата обращения 02.03.2023], Доступно: https://isd-soft.com/tech_blog/postgresql-explaining-explain-command/
3. Mihail Curchi, “PostgreSQL – Explaining the EXPLAIN command part 2” - [online] [дата обращения 02.03.2023], Доступно: https://isd-soft.com/tech-blog/enpostgresql-explaining-the-explain-command-part-2/