Мадока ВТ Вики

БД:Теория:Глава 6

В предыдущих главах мы научились создавать таблицы, наполнять их данными и извлекать эти данные с помощью SQL-запросов. Теперь разберемся, как СУБД выполняет эти запросы “под капотом” и как можно ускорить их выполнение с помощью **индексов**. Эта глава основана на материалах Лекций 6: “Индексы. Выполнение запросов” и Лекции 7: “Выполнение запросов”.

Часть 1: Повышение Производительности Запросов

Когда данных в таблицах становится много (тысячи, миллионы строк), даже простые запросы могут выполняться долго. Почему? Потому что в самом простом случае СУБД приходится последовательно просматривать **всю** таблицу, чтобы найти нужные строки (это называется **полное сканирование таблицы** или **Sequential Scan**).

Способы повышения производительности:

1. **Использование индексов:** Создание дополнительных структур данных, которые позволяют СУБД быстро находить нужные строки по значениям в определенных столбцах, не просматривая всю таблицу. (Основная тема этой главы).
2. **Оптимизация запросов:** Переписывание SQL-запросов так, чтобы они были более понятны оптимизатору СУБД и могли быть выполнены более эффективно (например, избегая ненужных соединений или используя `EXISTS` вместо `IN` в некоторых случаях).
3. **Настройка физических параметров СУБД:** Оптимизация конфигурации сервера PostgreSQL (выделение памяти, настройка параллелизма, параметры ввода-вывода и т.д.).
4. **Денормализация:** (Рассматривали ранее) Иногда используется для ускорения чтения за счет избыточности.
5. **Партиционирование таблиц:** Разделение очень больших таблиц на физически отдельные части (партиции) для ускорения доступа к конкретным диапазонам данных.

Мы сосредоточимся на **индексах** и понимании **выполнения запросов**.

Часть 2: Индексы

Что такое Индекс?

Представьте себе предметный указатель в конце большой книги. Вместо того чтобы листать всю книгу в поисках нужного термина, вы смотрите в указатель, находите термин и видите номера страниц, где он встречается.

Индекс SQL работает похожим образом:

- Это отдельная структура данных (обычно хранящаяся в отдельном файле), связанная с таблицей.
- Он содержит **копии значений** из одного или нескольких столбцов таблицы.

- Эти значения в индексе **упорядочены** определенным образом (например, по алфавиту для строк, по возрастанию для чисел).
- Каждое значение в индексе имеет **указатель** (ссылку, адрес) на те строки в основной таблице, где это значение встречается.

Как индекс помогает?

Рассмотрим запрос:

```
SELECT * FROM student WHERE StudID = 18;
```

- **Без индекса:** СУБД читает *каждую* строку таблицы `student` и сравнивает значение в колонке `StudID` с числом 18. Если таблица большая, это долго.
- **С индексом по `StudID`:**
  1. СУБД обращается к индексу по `StudID`.
  2. Так как значения в индексе упорядочены, СУБД **быстро** находит значение 18 (например, используя бинарный поиск или структуру B-дерева).
  3. Из индекса СУБД получает указатели на строки в основной таблице, где `StudID = 18`.
  4. СУБД читает **только** эти конкретные строки из основной таблицы.

Пример (из слайдов 10-17):

- Есть таблица `STUDENT`.
- Создан индекс по колонке `StudID`. Индекс содержит значения `StudID` и указатели на строки.
- Запрос: `SELECT * FROM STUDENT WHERE StudID = 18;`
- СУБД смотрит в индекс, быстро находит записи со значением 18.
- Получает из индекса указатели на строки с ID=18 (в примере это строки с Eugene Serov, Petr Menchikov, Roman Klever).
- Читает только эти три строки из таблицы `STUDENT`.

Важные моменты:

- Индексы работают **неявно**: Вам не нужно указывать в `SELECT`-запросе, какой индекс использовать. Оптимизатор СУБД сам решает, использовать ли индекс и какой именно, на основе статистики данных и стоимости выполнения разных планов.
- Иногда СУБД может **не использовать** индекс, даже если он есть. Например, если запрос выбирает очень большую часть таблицы (проще прочитать всю таблицу), или если условия в `WHERE` таковы, что индекс неприменим (например, функция над индексированным столбцом).

Когда Индексы Полезны?

Индексы ускоряют операции, включающие:

- **Поиск строк по условиям (`WHERE`)**: Особенно для условий равенства (=), диапазонных запросов (>, <, `BETWEEN`) по индексированному столбцу.
- **Соединения таблиц (`JOIN`)**: Индексы по столбцам, участвующим в условии `ON`, значительно ускоряют поиск совпадающих строк в соединяемых таблицах. Первичные и уникальные ключи автоматически индексируются в PostgreSQL, что ускоряет соединения по ним.
- **Поиск минимального/максимального значения (`MIN()`, `MAX()`):** Если столбец индексирован, СУБД может просто взять первое/последнее значение из индекса, не

сканируя таблицу.

- **Сортировку (`ORDER BY`)**: Если порядок сортировки совпадает с порядком индекса, СУБД может избежать фактической сортировки данных, просто прочитав строки в порядке индекса.
- **Группировку (`GROUP BY`)**: Индексы могут помочь СУБД быстрее найти строки, относящиеся к одной группе.

Недостатки Индексов:

1. **Занимают место:** Индекс — это дополнительная структура данных, которая хранится на диске и занимает место (иногда сравнимое с размером самой таблицы).
2. **Замедляют операции записи (`INSERT`, `UPDATE`, `DELETE`):** При изменении данных в индексированном столбце или добавлении/удалении строки СУБД должна обновить не только саму таблицу, но и **все** связанные с ней индексы. Это добавляет накладные расходы. Чем больше индексов на таблице, тем медленнее будут операции записи.
3. **Неэффективны на маленьких таблицах:** Если таблица настолько мала, что целиком помещается в память, СУБД прочитает ее полностью быстрее, чем будет обращаться к индексу.
4. **Неэффективны при выборке больших объемов данных:** Если запрос выбирает значительную часть таблицы (например, `WHERE column > 0`, где почти все значения больше нуля), СУБД, скорее всего, решит, что проще прочитать всю таблицу последовательно (`Seq Scan`), чем много раз обращаться к индексу и затем к разным частям таблицы за строками (`Index Scan` + много чтений с диска).

Стратегии Применения Индексов:

- **Анализируйте запросы:** Какие столбцы часто используются в `WHERE` и `JOIN`? Какие операции преобладают — чтение или запись?
- **Индексируйте столбцы в `WHERE`:** Особенно те, по которым идет поиск на равенство или по диапазону и которые обладают хорошей **селективностью** (т.е. по значению в этом столбце можно отобрать небольшую долю строк таблицы). Индексировать столбец “пол” (М/Ж) обычно бессмысленно, так как каждое значение выбирает примерно половину таблицы.
- **Индексируйте столбцы внешних ключей:** Это критически важно для ускорения `JOIN`.
- **Индексируйте столбцы в `ORDER BY`:** Если часто нужна сортировка по определенному столбцу.
- **Используйте составные индексы:** Если часто ищете по комбинации столбцов (`WHERE col1 = ? AND col2 = ?`), создайте индекс по `(col1, col2)`. Порядок столбцов в составном индексе важен! Индекс по `(col1, col2)` может использоваться для запросов по `col1` и для запросов по `col1` и `col2`, но *не* для запросов только по `col2`.
- **Не создавайте лишних индексов:** Каждый индекс замедляет запись. Удаляйте неиспользуемые индексы.
- **Первичные и уникальные ключи индексируются автоматически.**

Создание Индексов (`CREATE INDEX`)

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] имя_индекса
ON имя_таблицы [ USING метод ]
( { имя_колонки | ( выражение ) } [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] );
```

- `UNIQUE`: Создает уникальный индекс (запрещает дубликаты значений в индексируемых столбцах, кроме `NULL`).
- `CONCURRENTLY`: (PostgreSQL) Позволяет создать индекс без блокировки операций записи в таблицу (полезно для больших таблиц на “живой” системе, но выполняется дольше).

- `имя_индекса`: Имя для вашего индекса.
- `имя_таблицы`: Таблица, для которой создается индекс.
- `USING метод`: Указывает тип индекса (метод доступа). По умолчанию в PostgreSQL это `btree`. Другие варианты: `hash`, `gist`, `gin`, `spgist`, `brin`.
- `( имя_колонки | ( выражение ) ... )`: Список столбцов или выражений, по которым строится индекс.
- `ASC | DESC`: Порядок сортировки для индекса (по умолчанию `ASC`).
- `NULLS FIRST | NULLS LAST`: Где размещать `NULL`-значения в индексе (по умолчанию `NULLS LAST` для `ASC` и `NULLS FIRST` для `DESC`).

Примеры:

```
-- Простой индекс по одной колонке (используется B-Tree по умолчанию)
CREATE INDEX idx_student_groupid ON student (GroupID);

-- Уникальный индекс
CREATE UNIQUE INDEX idx_student_email ON student (email);

-- Составной индекс по двум колонкам
CREATE INDEX idx_student_surname_name ON student (Surname, Name);

-- Индекс по выражению (например, по фамилии в нижнем регистре для регистронезависимого поиска)
CREATE INDEX idx_student_surname_lower ON student (lower(Surname));

-- Хеш-индекс (только для равенства '=')
CREATE INDEX idx_student_name_hash ON student USING hash (Name);
```

Часть 3: Типы Индексов

PostgreSQL поддерживает несколько типов индексов, оптимизированных для разных задач.

1. **B-дерево (B-Tree):**
   - Используется **по умолчанию**.
   - **Структура:** Сбалансированное, сильно ветвистое дерево поиска. Состоит из корневого узла, промежуточных узлов и листовых узлов. Листовые узлы содержат индексируемые значения и указатели (TID - Tuple Identifier, адрес строки на диске) на строки таблицы, отсортированные по индексируемому значению. Внутренние узлы содержат “разделители” и ссылки на дочерние узлы для быстрой навигации по дереву.
   - **Преимущества:** Очень универсальный. Эффективен для:
     - Поиска по равенству (=).
     - Диапазонных запросов (>, <, `BETWEEN`).
     - Сортировки (`ORDER BY`).
     - Поиска `NULL` (`IS NULL`, `IS NOT NULL`).
     - Частичного поиска по префиксу (`LIKE 'abc%'`).
     - Работы с `MIN()` и `MAX()`.
   - **Пример поиска (StudID = 85):** СУБД спускается от корня B-дерева, сравнивая 85 с ключами в узлах, быстро находит нужный листовой узел, а в нем — запись с ключом 85 и указателем на соответствующую строку в таблице. Для диапазона `>= 85` находится первый узел с 85, а затем последовательно просматриваются следующие листовые узлы.
2. **Хеш-индекс (Hash):**

- **Структура:** Основан на хеш-таблице. Для индексируемого значения вычисляется хеш-код, который используется для определения “корзины” (bucket), где хранятся указатели на строки с этим значением.
- **Преимущества:** Очень быстрый для операций поиска по **точному равенству (=)**.
- **Недостатки:**
  - **Бесполезен** для диапазонных запросов (>, <), сортировки (`ORDER BY`), поиска по префиксу (`LIKE 'abc%'`). Хеш-коды не сохраняют порядок значений.
  - Требует больше дискового пространства при увеличении числа записей (хеш-таблица может расширяться).
  - До PostgreSQL 10 хеш-индексы не логировались в WAL, что делало их ненадежными после сбоев (требовалась переиндексация). Сейчас эта проблема решена.
- **Когда использовать:** Редко. Только если запросы к столбцу — это исключительно проверки на точное равенство, и B-дерево по какой-то причине не устраивает.

3. **GiST (Generalized Search Tree):**
   - Обобщенное дерево поиска. Каркас для построения индексов для сложных типов данных (геометрические данные, текстовый поиск, диапазоны и т.д.).
   - Позволяет индексировать операции типа “пересекается”, “содержит”, “находится внутри”.
   - Используется расширениями, такими как PostGIS (для геоданных) и `pg_trgm` (для поиска по похожести строк).
4. **GIN (Generalized Inverted Index):**
   - Обобщенный инвертированный индекс. Оптимизирован для индексации **составных** значений, где одно значение в столбце может содержать несколько “ключей” (например, массив слов в документе, элементы массива `jsonb`).
   - Хранит карту “ключ -> список строк, где ключ встречается”.
   - **Преимущества:** Очень эффективен для поиска строк, содержащих определенные значения внутри составного типа (например, найти все документы, содержащие слово ‘postgresql’, или все записи, где массив содержит число 5).
   - Используется для полнотекстового поиска, индексации `jsonb`, `hstore`, массивов.
5. **SP-GiST (Space-Partitioned GiST):** Для не сбалансированных структур данных (например, префиксные деревья).
6. **BRIN (Block Range Indexes):** Хранит *сводную* информацию (минимум, максимум) для больших диапазонов физических блоков таблицы. Очень компактный, но эффективен только для данных, которые сильно коррелируют с их физическим расположением в таблице (например, даты или ID в таблице, куда данные только добавляются).

Часть 4: Выполнение Запросов и Оптимизация

Как СУБД (на примере PostgreSQL) выполняет SQL-запрос:

1. **Разбор запроса (Parser):** Текст SQL-запроса преобразуется во внутреннее представление — **дерево разбора (parse tree)**. Проверяется синтаксис.
2. **Преобразование запроса (Rewriter):** Дерево разбора преобразуется с учетом правил и представлений (например, запрос к представлению заменяется на запрос к базовым таблицам). Результат — **дерево запроса (query tree)**.
3. **Планировщик/Оптимизатор (Planner/Optimizer):** Самый сложный этап.
   - Генерирует **множество возможных планов выполнения** для дерева запроса, используя правила реляционной алгебры и информацию об имеющихся индексах и

алгоритмах выполнения операций (Seq Scan, Index Scan, Nested Loop Join, Hash Join, Merge Join, Sort, Aggregate и т.д.).
   - **Оценивает стоимость** каждого плана (на основе статистики таблиц: количество строк, распределение значений, размер таблицы и т.д.). Стоимость обычно измеряется в условных единицах, отражающих дисковый ввод-вывод и использование CPU.
   - Выбирает план с **минимальной оцененной стоимостью**.
   - Результат — **план выполнения (execution plan)**.
4. **Выполнение плана (Executor):** План выполнения передается исполнителю, который шаг за шагом выполняет операции плана (чтение данных, соединение, фильтрацию, сортировку и т.д.) и возвращает результат пользователю.

Оптимизация и Реляционная Алгебра:

Оптимизатор активно использует законы реляционной алгебры для генерации эквивалентных планов. Ключевые идеи оптимизации:

- **Выполнять выборку (`WHERE`) как можно раньше:** Уменьшает количество строк для последующих операций (особенно для `JOIN`). Оптимизатор “проталкивает” условия `WHERE` вниз по дереву плана.
- **Выполнять проекцию (`SELECT`-список) как можно раньше:** Уменьшает количество и размер данных, передаваемых между операциями (особенно если используется конвейерная обработка).
- **Выбирать оптимальный порядок и алгоритм соединений:** Порядок соединения таблиц (`A JOIN B JOIN C` vs `A JOIN C JOIN B`) и алгоритм (`Nested Loop`, `Hash Join`, `Merge Join`) сильно влияют на производительность. Оптимизатор оценивает разные варианты.

Материализация vs Конвейерная Обработка (Pipelining):

- **Материализация:** Результат одной операции (например, `JOIN`) полностью вычисляется и сохраняется во временную структуру (в памяти или на диске), а затем передается следующей операции. Требует доп. памяти/диска и времени на запись/чтение.
- **Конвейерная обработка:** Как только первая операция производит первую строку результата, эта строка немедленно передается на вход следующей операции, не дожидаясь завершения первой. Значительно экономит ресурсы и время, если это возможно. Оптимизатор старается использовать конвейер.

Левосторонние Деревья Планов:

- В плане выполнения, где узлы — это операции `JOIN`, левостороннее дерево — это план, где правым входом для каждой операции `JOIN` всегда является одна из исходных таблиц, а левым — результат предыдущего соединения.
- Многие оптимизаторы (включая PostgreSQL по умолчанию для большого числа таблиц) рассматривают в основном левосторонние планы, так как они хорошо подходят для конвейерной обработки и сокращают пространство поиска планов.

Алгоритмы Выполнения Соединений (`JOIN`):

СУБД использует разные алгоритмы для физического выполнения операции `JOIN`. Выбор зависит от размера таблиц, наличия индексов и условий соединения.

1. **Nested Loop Join (Соединение вложенными циклами):**
   - **Принцип:** Для *каждой* строки из внешней таблицы (R) просматриваются *все* строки внутренней таблицы (S). Если строки удовлетворяют условию соединения, они комбинируются и возвращаются.

- **Простой вариант:** Очень медленный, если таблицы большие (число сравнений = |R| * |S|).
- **Block Nested Loop:** Оптимизация. Внешняя таблица читается блоками. Для каждого блока внешней таблицы читается вся внутренняя таблица (или ее блок) и выполняется соединение в памяти. Уменьшает количество чтений внутренней таблицы.
- **Index Nested Loop:** Если по столбцу соединения во *внутренней* таблице (S) есть **индекс**, то для каждой строки из внешней таблицы (R) выполняется быстрый поиск по индексу в S, вместо полного сканирования S. Очень эффективен, если внешняя таблица (R) маленькая, а по внутренней (S) есть подходящий индекс.
- **Когда используется:** Часто для соединения маленькой таблицы с большой (при наличии индекса у большой) или для не-эквисоединений.

2. **Sort-Merge Join (Соединение слиянием):**
   - **Принцип:**
     1. *Обе* таблицы сортируются по столбцам соединения.
     2. Отсортированные таблицы “сливаются”: СУБД одновременно проходит по обеим таблицам, сравнивая текущие строки. Если значения в столбцах соединения совпадают, строки комбинируются. Если значение в одной таблице меньше, СУБД продвигается по этой таблице до следующего значения.
   - **Когда используется:** Эффективен для эквисоединений, особенно если таблицы уже отсортированы (например, читаются из индекса B-дерева в нужном порядке) или если результат соединения нужно будет отсортировать по тем же столбцам. Требует затрат на сортировку, если таблицы не отсортированы.
3. **Hash Join (Соединение по хешу):**
   - **Принцип:**
     1. Строится **хеш-таблица** в памяти по *меньшей* из двух таблиц (фаза построения). Ключом хеш-таблицы является значение столбца(ов) соединения.
     2. Сканируется *бОльшая* таблица (фаза зондирования). Для каждой строки вычисляется хеш от столбца(ов) соединения и выполняется поиск в хеш-таблице. Если совпадение найдено, строки комбинируются.
   - **Когда используется:** Очень эффективен для **эквисоединений** больших таблиц, особенно если хеш-таблица помещается в память. Может требовать записи временных данных на диск, если хеш-таблица слишком велика. Не подходит для не-эквисоединений.

Часть 5: Просмотр Плана Выполнения в PostgreSQL

Чтобы понять, как PostgreSQL собирается выполнить ваш запрос (или как он его выполнил), используется команда `EXPLAIN`.

- **`EXPLAIN SQL_ЗАПРОС;`**: Показывает **план выполнения**, который *построил* оптимизатор, но **не выполняет** сам запрос. Полезно для анализа сложных запросов без их реального запуска.
- **`EXPLAIN ANALYZE SQL_ЗАПРОС;`**: **Выполняет** запрос и показывает **реальный план выполнения** с фактическим временем выполнения каждой операции и количеством строк. Дает более точную картину, но требует реального выполнения запроса (будьте осторожны с `UPDATE`, `DELETE`, `INSERT`).

Чтение вывода `EXPLAIN`:

Вывод представляет собой дерево операций плана. Каждая строка — узел дерева, с отступом показана вложенность.

```
QUERY PLAN
-------------------------------------------------------------------------------------------
Index Scan using students_studid_pkey on students  (cost=0.32..8.34 rows=1 width=150)
  Index Cond: (StudId = 942)
```

- **Тип узла:** `Index Scan` (использование индекса), `Seq Scan` (полное сканирование), `Nested Loop`, `Hash Join`, `Merge Join`, `Sort`, `Aggregate` и т.д.
- **`on table_name`**: Таблица, к которой применяется операция.
- **`using index_name`**: Используемый индекс.
- **`cost=startup..total`**: Оценочная стоимость. `startup` - стоимость до получения первой строки, `total` - стоимость получения всех строк. Единицы условные.
- **`rows=N`**: Оценочное количество строк, которое вернет этот узел.
- **`width=N`**: Оценочный средний размер строки в байтах.
- **Дополнительные условия:** `Index Cond`, `Filter`, `Join Filter`, `Hash Cond` — условия, применяемые на этом узле.

Пример `EXPLAIN ANALYZE`:

```
EXPLAIN ANALYZE SELECT * FROM students WHERE StudID = 942;

QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
---------
 Index Scan using students_studid_pkey on students  (cost=0.32..8.34 rows=1 width=150) (actual
time=0.025..0.026 rows=1 loops=1)
   Index Cond: (StudId = 942)
 Planning time: 0.150 ms
 Execution time: 0.050 ms
(4 rows)
```

- **`actual time=startup..total`**: Фактическое время выполнения узла в миллисекундах.
- **`rows=N`**: Фактическое количество строк, возвращенное узлом.
- **`loops=N`**: Сколько раз был выполнен этот узел (важно для вложенных циклов).
- **`Planning time`**: Время, затраченное на планирование/оптимизацию.
- **`Execution time`**: Общее время выполнения запроса.

Анализ вывода `EXPLAIN` и `EXPLAIN ANALYZE` — ключевой навык для оптимизации медленных запросов. Он позволяет понять, какие операции являются “бутылочным горлышком”, используются ли индексы, корректны ли оценки оптимизатора.
