Способы повышения производительности запросов:

- Использование индексов
- Настройка физических параметров СУБД
  (СУБД: Выбор наименьшего по стоимости плана выполнения запроса)

Индекс - список всех значений в группе из одного или нескольких столбцов, упорядоченный в каком-то смысле.
То есть значение этого списка будет ссылаться на значение в реальной таблице.

! Индексы работают неявно !
То есть в каком то запросе могут использоваться, в каком-то нет.

Индексы ускоряют:

- работу с большими таблицами
- работу WHERE и JOIN
- Агрегатные функции MIN и MAX
- Сортировку и группировку столбцов таблицы

Недостатки:

- Индекс занимает место в памяти
- При удалении изменении содержимого в реальной таблице также надо будет менять и индексацию, то есть замедляет операции
- При работе с маленькими таблицами будет неэффективен
- Также неэффективен если по условию выборки ожидается выбор большого кол-ва данных
  Создание индекса
  обычный
  CREATE INDEX index_name ON table_name (column_name);
  составной
  CREATE INDEX index_name ON table_name (column1_name, column2_name);

Вдерево:
Связанный ациклический граф
Btree Index - индес сгрупированный по листьям Btree
Свойства BTree:

- значения внутри каждой ноды отсортированы
- ключ равномерно распределены по узлам
- полезно при использовании с операторами сравнения

HashIndex: Для построения такого индекса используется хэшфункция
Хэш-функция — функция для преобразования входных данных в результирующие данные фиксированного формата. **Полезен**: для оператора =
GiST —Это сбалансированное дерево поиска, точно так же, как и рассмотренный ранее b-tree.
Он позволяет задать принцип распределения данных произвольного типа по сбалансированному дереву, и метод использования этого представления для доступа по некоторому оператору.
GIN расшифровывается как Generalized Inverted Index — это так называемый *обратный индекс*. Он работает с типами данных, значения которых не являются атомарными, а состоят из элементов. При этом индексируются не сами значения, а отдельные элементы; каждый элемент ссылается на те значения, в которых он встречается.

Структура выполнения запроса:

1. Парсер
2. Рерайтер (преобразователь)
3. Оптимизатор (планнер)
4. Экзекутор (исполнитель)

План выполнения запроса:
Соответствующий SQL запросу алгоритм выполнения запроса.

Таких планов может быть много, СУБД должна выбирать наиболее эффективный из построенных планов

Критерии оценивания плана:

- Число обменов с внешней памятью
- Среднее время обмена

Реляционная алгебра и построение планов выполнения:

- R, S — отношения (таблицы)
- φ — предикат (условие), $\varphi_1 \wedge \varphi_2$ — составное условие

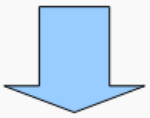
$\sigma_{\varphi}(R)$ — операция выборки

В результате данной операции формируются результат, который содержит только те

строки, которые удовлетворяют предикату **φ**. R - таблица откуда будет произведена выборка.

Пример:

```
SELECT * FROM STUDENTS WHERE
        STUDENTS.GROUP = '3100' AND
        STUDENTS.ID >= 150000;
```



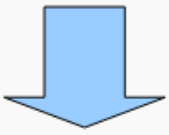
$$\sigma_{(STUDENTS.GROUP='3100')\Lambda(STUDENTS.ID>=150000)}(STUDENTS)$$

$\pi_{attr}(R)$ — проекция

Операция в результате которой будут выбраны только те атрибуты из R, который указаны в **attr**.

Пример:

```
SELECT name, group FROM STUDENTS;
```

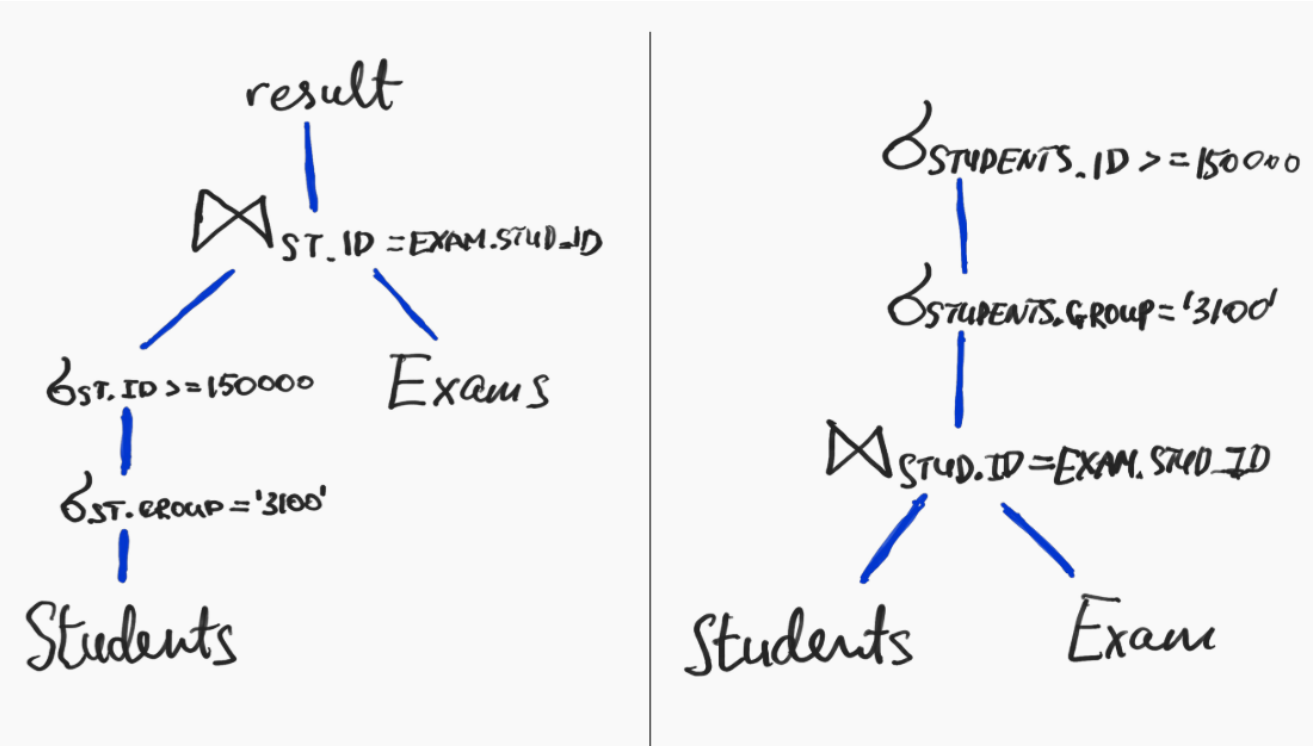


$$\pi_{name,\ group}(STUDENTS)$$

$R \bowtie_{\theta} S$ — соединение

Операция в результате которой будет получено соединение таблиц R с таблицей S по условию **θ**.

Пример выполнения одного и того же запроса разными планами:



Законы реляционной алгебры:

$R \bowtie_\theta S \equiv S \bowtie_\theta R$ (коммутативность)

$R \bowtie_\theta (S \bowtie_\varphi T) \equiv (R \bowtie_\theta S) \bowtie_\varphi T$ (ассоциативность)

$\sigma_{\theta \wedge \varphi}(R) \equiv \sigma_\theta(\sigma_\varphi(R))$

$\sigma_\varphi(R \bowtie_\theta S) \equiv (\sigma_\varphi(R) \bowtie_\theta S)$, если φ относится к атрибутам R

$\pi_A(R \bowtie_\theta S) \equiv \pi_A(\pi_{(A \cup B) \cap attrs(R)}(R) \bowtie_\theta S)$, B — атрибуты из условия θ

Материализация данных:
Сохранение результатов промежуточных операций.
Увеличивает время выполнения запроса т.к. добавляет две доп операции в план - запись промежуточных данных и затем их чтение.
Конвейерная обработка данных:

передача результатов одной обработки другой без создания промежуточных/буферных/временных отношений (таблиц).

Левостороннее дерево:
Дерево в котором внешнее отношение всегда находиться слева.
- Сокращает число планов для анализа
- Использует конв. обработку данных

Типы деревьев
1. Левостороннее (Внешнее отношение всегда на левой ветке)
2. Смешанное линейное (Внешнее отношение на левой ветке XOR на правой ветке)
3. Нелинейное (Внешнее отношение на левой или на правой ветке)
Советы:
1. Использовать конв. обработку (левосторонние планы)
2. Делать выборку как можно раньше
3. Делать проекции раньше
4. Грамотно планировать соединения
Выполнение соединений (JOIN)

1. Block Nested Loop Join

Block Nested Loop Join (блочное вложенное соединение) — это улучшенная версия обычного Nested Loop Join. Основное отличие заключается в том, что он работает с блоками данных, что позволяет сократить количество обращений к диску.

- **Принцип работы**:
  1. Разбивает одну из таблиц (обычно меньшую) на блоки.
  2. Каждому блоку из первой таблицы сопоставляется вся вторая таблица.
  3. Для каждого блока выполняется соединение с использованием Nested Loop Join.
- **Преимущества**:
  - Улучшенная производительность за счет уменьшения числа обращений к диску.
  - Подходит для больших наборов данных.
- **Недостатки**:
  - По-прежнему может быть медленным для очень больших таблиц, если нет индексов.

2. Nested Loop Join

Nested Loop Join (вложенное соединение) — это базовый метод соединения, который подходит для небольших таблиц или когда вторая таблица (inner table) имеет индекс по ключу соединения.

- **Принцип работы**:

1. Для каждой строки из внешней таблицы (outer table) выполняется цикл по всем строкам внутренней таблицы (inner table).
2. Каждая пара строк проверяется на выполнение условия соединения.

- **Преимущества**:
  - Прост в реализации.
  - Эффективен для небольших таблиц или если на внутреннюю таблицу есть индекс.
- **Недостатки**:
  - Может быть очень медленным для больших таблиц из-за большого количества итераций.

Hash Join

Hash Join — это метод соединения, который используется для больших наборов данных, особенно когда нет индексов по ключам соединения. Этот метод часто оказывается наиболее эффективным для соединений с большими таблицами.

Принцип работы:

1. **Построение хеша (Build Phase)**:
   - Хеширование одной из таблиц (обычно меньшей) по ключу соединения.
   - Создание хеш-таблицы в памяти, где ключом является значение из колонки соединения, а значением — вся строка.
2. **Пробег по второй таблице (Probe Phase)**:
   - Для каждой строки из второй таблицы вычисляется хеш по ключу соединения.
   - Выполняется поиск соответствующих значений в хеш-таблице.
   - Если соответствие найдено, строки соединяются.

Преимущества:

- Эффективен для соединения больших таблиц.
- Хорошо работает с неиндексированными таблицами.

Недостатки:

- Требует значительных объемов оперативной памяти для хранения хеш-таблиц.
- Производительность может снизиться, если хеш-таблицы не помещаются в памяти.

Sort-Merge Join

Sort-Merge Join — это метод соединения, который используется для отсортированных таблиц. Он также эффективен для больших наборов данных, особенно если таблицы

уже отсортированы по ключу соединения.

Принцип работы:

1. **Сортировка**:
   - Обе таблицы сортируются по ключу соединения, если они еще не отсортированы.
2. **Слияние**:
   - Одновременный проход по обеим отсортированным таблицам.
   - Сравнение текущих строк из обеих таблиц.
   - Если ключи совпадают, строки соединяются.
   - Если ключи не совпадают, указатель смещается на следующую строку в таблице с меньшим значением ключа.

Преимущества:

- Эффективен для больших таблиц, если они уже отсортированы по ключу соединения.
- Поддерживает устойчивость к внешним данным, если сортировка уже выполнена.

Недостатки:

- Требует сортировки, если таблицы не отсортированы, что может быть дорогостоящей операцией.
- Требует значительного объема дискового ввода-вывода для сортировки больших таблиц.

Index Nested Loop Join

Index Nested Loop Join — это оптимизированная версия Nested Loop Join, которая использует индексы для ускорения поиска соответствующих строк. Этот метод подходит для случаев, когда хотя бы одна из таблиц индексирована по ключу соединения.

Принцип работы:

1. **Внешний цикл (Outer Loop)**:
   - Для каждой строки из внешней таблицы выполняется поиск соответствующих строк в внутренней таблице.
2. **Поиск по индексу (Index Lookup)**:
   - Внутренняя таблица (inner table) ищется с использованием индекса.
   - Для каждой строки из внешней таблицы производится поиск соответствующих строк в индексе внутренней таблицы.

Преимущества:

- Быстр и эффективен, если на внутреннюю таблицу есть индекс по ключу соединения.
- Уменьшает количество операций поиска за счет использования индекса.

Недостатки:

- Менее эффективен, если нет индексов на внутреннюю таблицу.
- Производительность может снизиться для очень больших таблиц при отсутствии индексов.
  EXPLAIN — позволяет посмотреть план выполнения запроса, отобранный PostgreSQL.
  Сам план **не выполняется**.
  Для выполнения запроса используется **EXPLAIN ANALYZE**

Выбор правильного плана выполнения

- **Nested Loop Join**:
  - Применяется для небольших таблиц или если внутренние таблицы индексированы по ключу соединения.
  - Меньшее количество операций, если используются индексы.
- **Hash Join**:
  - Эффективен для больших таблиц, особенно если нет индексов.
  - Использует хеш-таблицу для быстрого поиска соответствий.
- **Sort-Merge Join**:
  - Применяется для отсортированных таблиц.
  - Быстрое соединение после сортировки данных.
- **Index Nested Loop Join**:
  - Оптимизированный вариант для индексированных таблиц.
  - Быстрое соединение, если внутренние таблицы индексированы по ключу соединения.