**Индексы**

Индекс представляет собой средство, помогающее ускорить поиск необходимых данных за счет физического или логического их упорядочивания. Индекс представляет собой набор ссылок, упорядоченных по определенному столбцу таблицы, который в данном случае будет называться индексированным столбцом. Индексы - это наборы уникальных значений для некоторой таблицы с соответствующими ссылками на данные. Они расположены в самой таблице и являются удобным внутренним механизмом системы SQL-сервера, с помощью которого осуществляется доступ к данным оптимальным способом.

Когда мы создаем индекс, мы добавляем в таблицу столбец, в котором в порядке возрастания хранятся ссыки на элементы для выбранного столбца. Мы добавляем дополнительный слой при поиске, но он уже отсортирован.

Индексы используются неявно

* При выполнении запроса БД сама определяет какие индексы нужно использовать
* В рамках одного и того же запроса БД может не использовать индекс

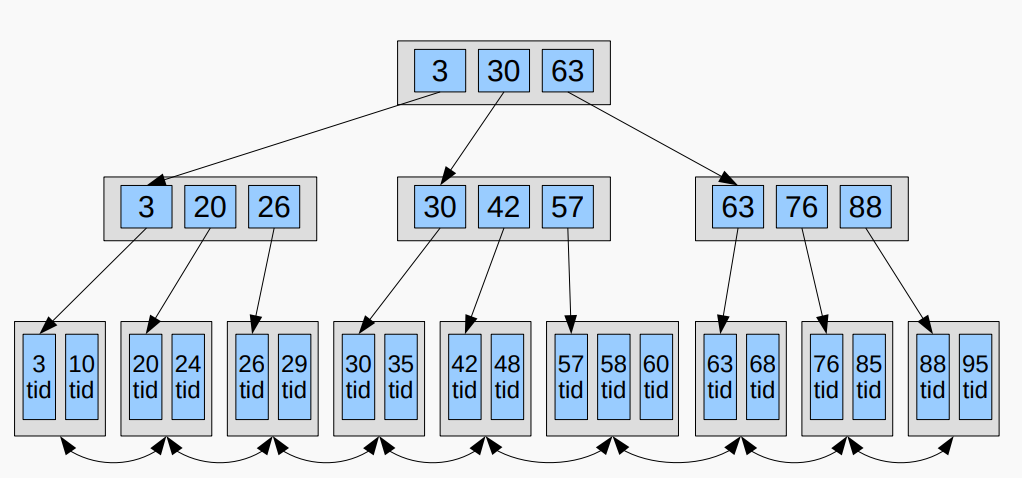
Индексы полезны, если в запросе присутствует:

* Предложения where join
* Min(), max()
* Сортировка или группировка

Минусы индексы

* Индексы занимают место в памяти
* Необходимо обновлять индекс, если что то произошло с индексированным столбцом
* Если таблиция маленького размера, то слой абстракции будет только замедлять обработку запроса.
* При выборе больших объемов данных

Типы индексов

* Btree (сбалансированное дерево) – создается по умолчанию. Представляет собой сбалансированное дерево, каждая вершина или ссылается на набор подвершин, или является листом, который содержит id нужной записи в таблице. Отсортировано по возрастанию.
  + Поддерживает операторы: >, <, >=, <=, =
  + Поддерживает LIKE
  + Индексирует NULL
  + Время поиска Olog(n) – бинарный поиск
* Хэш-индекс (Хэш-индекс) использует хэш функцию для данных
  + Поддерживает только оператор =
  + Время поиска O(1)
* Gist (обобщенное дерево поиска) используется для индексации геометрических типов данных и текстов.
* Gin (обобщенный обратный) – для индексации массивов
* SP-GiST (GIST с двоичным разбиением пространства) - используется для данных которые уже упорядочены, но не сбалансированы (номера телефонов)
* BRIN (блочно-диапазонный) полезен на огромных наборах данных, которые подразумевают естественную упорядоченность (почтовые индексы)

Чем меньшему количеству строк соответствует значение атрибута, тем выше селективность. Такие атрибы следует использовать в начале индекса.

Атрибуты из ORDER BY нужно добавлять в конец индексов т.к. там мы имеем отсортированный список.

EXPLAIN - показать какие индексы будут использоваться для этого запроса.

EXPLAIN SELECT \* FROM products WHERE price = 9990 AND category = 'Телевизоры'

**Оптимизация запросов**

EXPLAIN – вывод примерного плана выполнения запроса на основе статистических данных

ANALYZE – протокол выполнения плана реального запроса

**Выбор плана выполнения запросов**

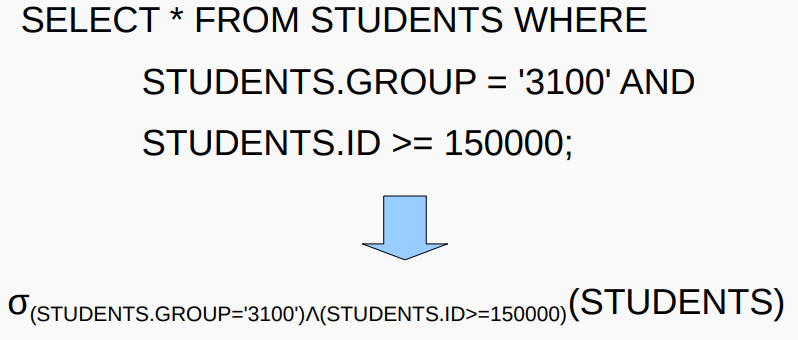
Так как SQL декларативен, то в начале он должен понять как надо обрабатывать запрос. Для этого

1. Разбор запроса (parser) строится дерево
2. Преобразование запроса
3. Планировщик + оптимизатор
4. Выполнение плана

План выполнения запроса

СУБД должна

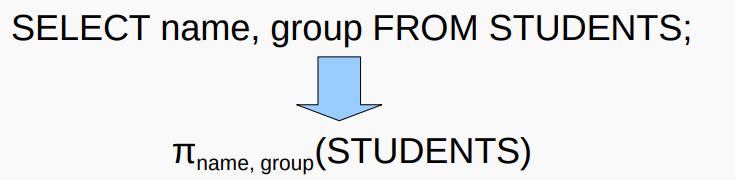
* Уметь построить возможные программы, результаты которых соответствуют заданному SQL запросу
* Выбрать программу, выполнение которой будет наиболее эффективной

При выборе плана выполнения запроса учитываются компонеты:

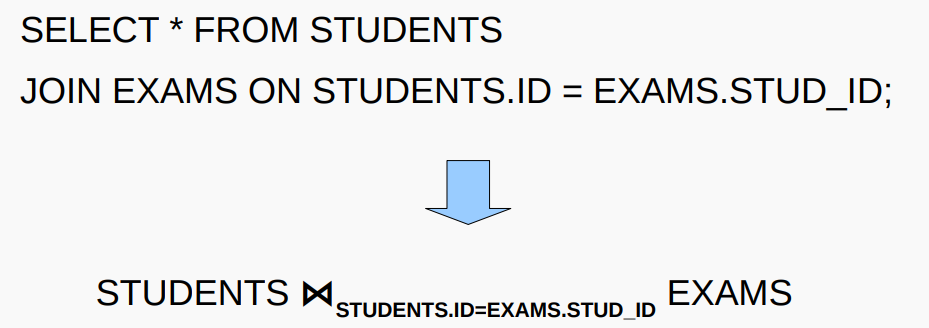
* Число обменом с устройствами внешней памяти
* Среднее время обмена

R, S – отношения (таблицы)

ϕ – предикат (условие)

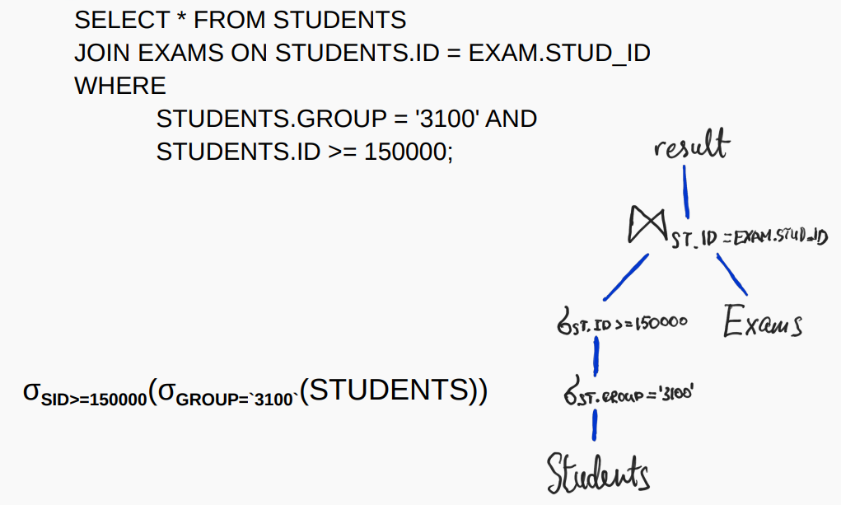
Α(R) – операция выборки

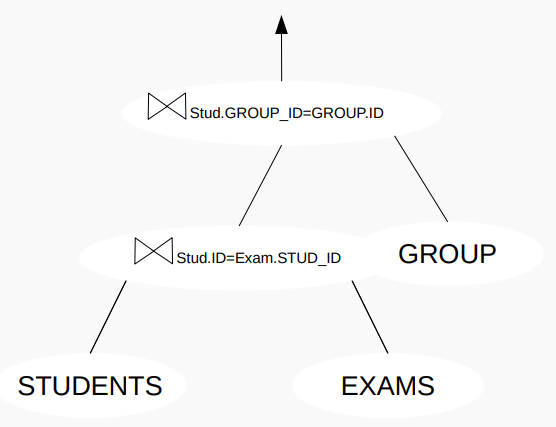
π(R) – проекция результат выборки



R∞S – соединение таблиц (join)

Материализация данных – позволяет нам сохранять результаты промежуточных вычислений, но увеличивает время выполнение запроса.

* Конв. обр. данных — передача результатов одной операции на обработку другой без создания временных отношений (для хранения промежуточных результатов).
* Для планов, в которых большинство операций происходят в конвейере:
  + при расчете стоимости плана выполнения запроса отпадает необходимость в учете стоимости записи и последующего чтения промежуточных таблиц.

Левостороннее дерево — результат соединения - в левой части дерева, представляющего план. Обычно оптимизаторы запросов в субд применяют их

* Необходимо сократить число планов для анализа
* Такие планы позволяют избежать материализацию, используя конвейерную обработку данных

При построении плана нужно учитывать:

* Использовать конвейерную обработку (левосторонние планы, избегать блокирующих операций).
* Делать выборку как можно раньше.
* Делать проекции раньше.
* Грамотно планировать соединения.
* Цель: уменьшение размеров промежуточных данных => уменьшение числа операций чтения записи во внешнюю память

Расчет стоимости плана выполнения запроса

* Чтение входных или промежуточных таблиц.
* Запись промежуточных данных (материализация — результат сохраняется во временных отношениях после выполнения одной операции для обработки следующей операцией).
* Сортировка результата (DISTINCT)

Оптимизация запросов

Для того, чтобы снять план используется EXPLAIN – показывает что происходит при запросе.

Обработка запроса в psql

* parser – синтаксический анализатор
* planner – выбор оптимального пути выполнения запроса EXPLAIN
* executor – выполнение согласно плану EXPLAIN ANALYZE

EXPLAIN дает нам план запроса в текстовом представлении, каждый узел – одна из операций

* Получение данных - … scan
* Построение битовых карт – bitmap …
* Обработка данных – sort, aggregate, append, limit…
* Операции над множеством – intersect/except …
* Соединение – nested loop, join, …
* Вложенный запрос – InitPlan, Subplan

Выполнение плана – обход дерева в глубину.

Вывод EXPLAIN ANALYZE

* Стоимость операции - абстрактная величина в операция ввода вывода и тактов CPU. В выводе 2 значения startup – стоимость извлечения первой, total – стоимость извлечения последней записи. Но это всего лишь статическая оценка.
  + cost=#...#
* Количество строк - количество записей результата
  + rows=#
* Ширина данных
  + width=#
* Реальные показатели
  + actual time – среднее время в миллисекундах до извлечения первой/последней записей
  + rows – реальное количество возвращаемых записей
  + loops – количества повторов узла

Получение данных

* Seq Scan – последовательный просмотр таблицы (оптимален для малых таблиц)
* Index Scan – просмотр по индексу
* Index only scan – просмотр по индексу без использования таблицы
* Tid scan – просмотр по физическому идентификатору записи – тык в конкретную запись
* CTE Scan/Worktable scan – просмотр набора записей, возвращаемых временной таблицей
* Values Scan – просмотр константного набора записей
* Result – запрос, не обращающийся к таблице.
* Function scan – просмотр набора записей возвращаемых функцией

Обработка данных

* Sort – сортитровка при явном задании ORDER BY в запросе
* Group, Aggregate, hashAggregate, GroupAggregate, Windowagg – группировка одинаковых значений, при наличии GROUP BY в запросе
* Unique – удаление повторяющихся строк, при использовании DISTINCT, UNION
* Hash – построение хэш таблицы
* Materialize – помещение результата во временную таблицу
* Limit – ограничение количества возвращаемых записей, при наличии LIMIT

Операции над множествами

* Subquery scan -просмотр подзапроса
* SetOp{…} – операция пересечения
* Append – присоединение
* Marge Append – присоединение с сортировкой
* Recursive Union – рекурсивное присоединение

Соединения

* Nested loop – пробегаем по всем записям одной из таблиц
  + С вложенным Seq scan – вложенный цикл, т.е. для каждой записи слева мы пробегаем все записи справа.
  + С вложенным Index scan – быстрее так как использует индекс
* Merge join – соединение слиянием, одновременно проходим по двум наборам записей (предварительно отсортированных) оптимально для больших таблиц
* Hash join - строим хэш таблицу, ищем совпадения
* Semi join / Anti join при использовании EXISTS/NOT EXISTS

Bitmap Index Scan

BitMap Heap Scan

Bitmap Scans всегда состоят, минимум, из двух узлов. Сначала (на нижнем уровне) идет Bitmap Index Scan, а затем – Bitmap Heap Scan.

Index cond

BitmapAnd

Nested loop Seq scan - вложенный цикл, т.е. для каждой записи слева мы пробегаем все записи справа.