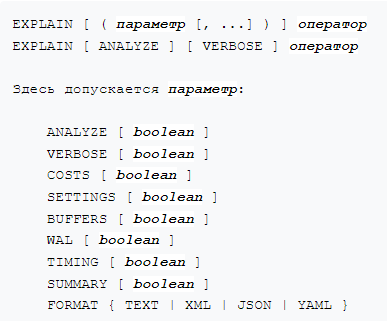
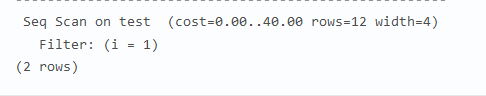
**ПЛАН ЗАПРОСА**

Выполняя полученный запрос, Postgres разрабатывает для него план запроса. В системе работает сложный планировщик, задача которого — подобрать хороший план.

Получить план запроса можно с помощью команды **explain** после которой следует сам запрос.



Пример вывода



Вначале указывается каким образом postgres будет искать данные. Затем указывается таблица.

**Cost –** стоимость операции, это число оценивает временные и ресурсные затраты на выполнение операции. Первое число – стоимость запуска до первой строки, второе – общая стоимость выдачи всех строк.

**Rows –** приблизительное количество строк, которое эта операция способна вернуть. Это именно количество строк, которые запрос может вернуть. Количество просканированных строк может быть и больше, например при использовании WHERE.

**Width –** приблизительно столько байт содержится в одной строке, возвращенной в рамках данной операции.

Для большинства операций важна общая стоимость, но в таких запросах как EXISTS, важно только нахождение первой строки, поэтому postgres пытается минимизировать стоимость запуска, а не общую.

Можно вывести также фактическое время, которое потребовалось на выполнение операции, используя **explain analyze.** В этом случае оператор будет выполнен на самом деле, а не только запланирован.

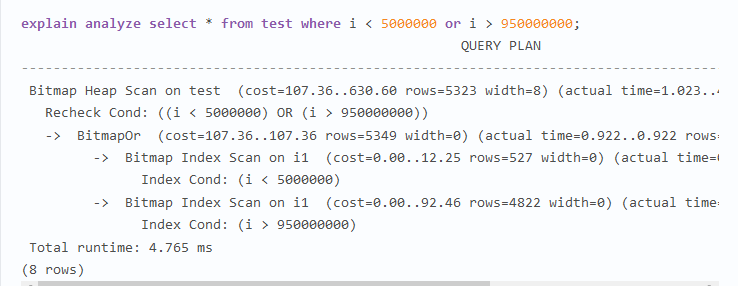
Но примерная стоимость запроса может варьироваться, так как статистика команды **analyze** рассчитывается по случайной выборке.

Некоторые узлы в плане запроса могут выполняться несколько раз, в таких случаях значение **loops** показывает, сколько всего раз выполнялся этот узел, а фактическое время и число строк вычисляется как среднее по всем итерациям.

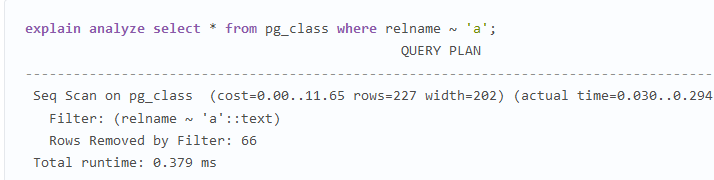
Виды операций в плане запроса:

* **Seq Scan on table\_name** – последовательное сканирование.
* **Index Scan using index\_name on table –** поиск по индексу. Postgres открывает индекс; в индексе находит, где в таблице находятся строки, соответствующие условию; открывает таблицу и получает строки, указанные индексом.  
  Сканирование по индексу также может использоваться при сортировке по проиндексированному полю.
* **Index Scan Backward –** сканирование по индексу в порядке убывания. Выполняется например в таком случае:  
  
* **Index Only Scan –** Выполняется когда мы выбираем колонки только из индекса, и нет необходимости лезть в саму таблицу. Данные будут возвращаться прямо из индекса.
* **Limit** – соответствует sql команде LIMIT  
  
* **Bitmap Heap Scan** в комбинации с **Bitmap Index Scan –** IndexScan Использует сканирование по индексу, но перед тем как обращаться к таблице, Heap Scan сортирует адреса взятые из индекса, чтобы уменьшить издержки на чтение**.**

Вначале Bitmap Index Scan создает битовую карту, где каждой странице из таблицы будет соответствовать 1 бит. Затем он установит биты в единички, если на этой странице может находится строка, которую надо вернуть. После того как будут найдены все страницы, Bitmap Heap Scan выполняет последовательный поиск в них, проверяя условие Rexheck Cond. Несколько условий создают несколько Bitman Index Scan, которые затем объединяются с помощью **Bitmap Or, And или Not.**

Сканирование по битовой карте **позволяет избежать повторных обращений к одной и той же странице данных**.

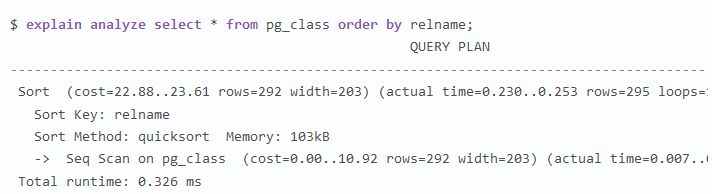
В операциях с условием WHERE план запроса содержит строку **Filter**, в которой указано, по каким критериям осуществлялась фильтрация и сколько строк было отброшено.



В операциях с ORDER BY план запроса содержит операцию **Sort**. В качестве доп информации указаны поле для сортировки и ее тип.

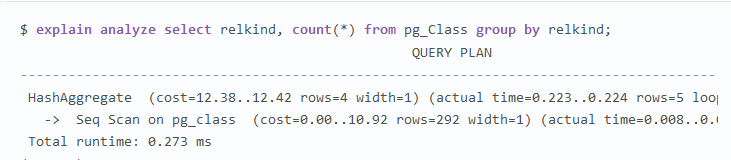
Если тип сортировки **Memory,** используется оперативная память.

Ели тип сортировки **Disk –** postgres использует временные файлы.



**HashAggregate –** эта операция в основном применяется при использовании GROUP BY и каких-нибудь агрегатных функций.

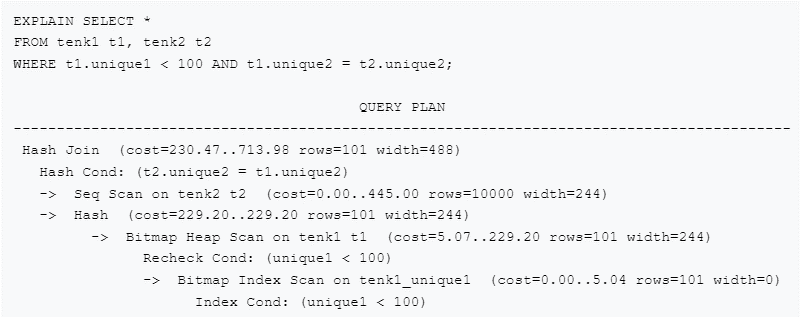
Перед выполнением этой операции сканируются все строки. Для каждой строки находит «ключ» по которому происходит группировка. Затем в хэше (ассоциативном массиве) добавляет выбранную строку в нужную корзину. После обработки всех строк, хэш сканируется и выполняется расчет агрегатных функций.



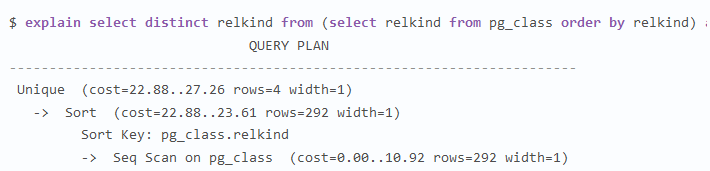
**HashJoin –** используется при объединении таблиц с помощью join.

Сначала HashJoin вызывает субоперацию **Hash,** которая в свою очередь вызывает обход таблицы. Hash создает ассоциативный массив, ключом которого является столбец по которому происходит объединение.

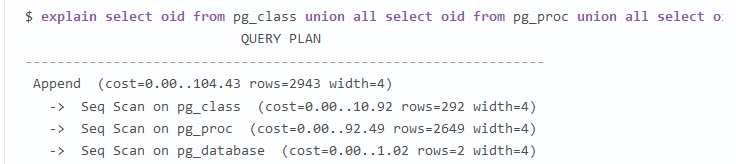
Потом Hash Join запускает вторую субоперацию и для каждой строки из нее ищет совпадение в хэше. Если совпадений нет, строка игнорируется. Если есть – то формируются совпадающие строки.



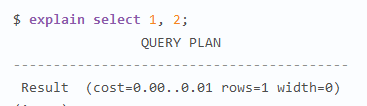
**Unique –** отбор уникальных строк, используется при вызове SQL оператора DISTINCT, но только когда строки отсортированы. Она сравнивает текущее значение с предыдущим, и если они одинаковые, отбрасывает текущее.



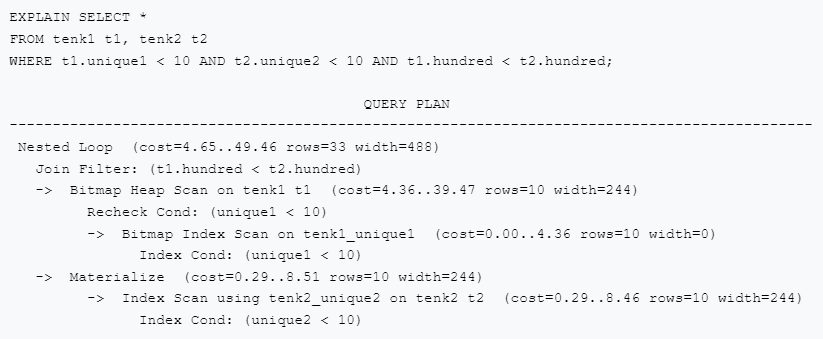
**Append –** запускает множество субопераций, и возвращает все возвращенные ими строки в виде общего результата. Используется в запросах с UNION и UNION ALL



**Result** – появляется в очень простых запросах, как правило, когда выбирается постоянное значение.



**Materialize –** сохраняет считанные данные в памяти, если к ним нужно будет обращаться несколько раз. Например выбрав с помощью индекса строки, мы сохраняем их, чтобы затем применить в условии WHERE. Иначе пришлось бы 10 раз лазить в индекс и выбирать строки.



Планировщик PosthreSQL выбирает операции основываясь на статистике.

Например представим такой запрос:



Если у всех строк в таблице одинаковое значение some\_value, тогда применение к столбцу индекса не имеет смысла.

С другой стороны, если значения в столбце уникальны (или почти уникальны), использование индекса – отличная идея.

При выборе метода доступа планировщик заглядывает в специальную статистику, где содержится такая информация как:

* Степень упорядоченности данных в столбцах
* Cardinality – степень уникальности того или иного значения.

Некоторые базы данных позволяют использовать **хинты**, чтобы указать базе данных на **необходимость использования индекса либо последовательного сканирования**. Однако это не более как рекомендация, и оптимизатор не обязательно ее исполнит.

**ИНДЕКСЫ**

**Индекс –** объект базы данных, создаваемый с целью ускорения поиска записей в бд. Набор ссылок, упорядоченных по определенному столбцу таблицы.

Индексы служат также для поддержки некоторых ограничений целостности.

Физически индекс – упорядоченный набор значений из индексированного столбца с указателями на места физического размещения исходных строк в структуре базы данных. Указатель состоит из номера блока файла и позиции строки внутри блока

Обычно ссылки хранятся в виде какой-либо структуры данных, оптимизированной для поиска. Без индекса будет выполняться полный просмотр (сканирование) всех строк таблицы, что может значительно повлиять на производительность.

**Кластерные индексы -** При кластеризованном индексе строки физически хранятся на диске в том же порядке, что и индекс.Таблица может иметь только 1 кластерный индекс.

Кластерный индекс автоматически создается при создании первичного ключа.

Он всегда должен быть уникальным. Если столбец содержит дублируещиеся значения, то СУБД принудительно обеспечивает уникальность, добавляя идентификатор к строкам, содержащим дубликаты значений.

При этом все значения будут отсортированы в определенном порядке либо возрастания, либо убывания.

**Некластерные индексы** – они не перестраивают физическую структуру таблицы, а лишь организуют ссылки на соответствующие строки. Поэтому таблица может иметь несколько некластеризованных индексов.

Листья такого индекса содержат только ключевые столбцы, по которым определен индекс.

Если в бд есть кластеризованный индекс, то некластерный обычно ссылается на соответсвующий элемент кластерного. Это позволяет не перестраивать структуру некластерного индекса всякий раз, когда кластерный индекс меняет физический порядок строк в таблице.

Если кластеризованного индекса нет, то некластеризованный ссылается непосредственно на строку с данными.

**Составной индекс –** может содержать более одного столбца. Могут быть как кластеризованными так и нет.

В составном индексе имеет значение, какой столбец был указан первый. Такой индекс может быть использован как для поиска по обоим столбцам, так и только по первому. А вот по второму не может.  
Порядок столбцов влияет также и на сортировку в индексе. Сначала он будет отсортирован по первому полю, затем по второму.

**Уникальный индекс –** обеспечивает уникальность каждого значения в индексируемом столбце, либо нескольких столбцов в связке, если индекс составной. В настоящее время уникальными могут быть только индексы B-деревья. При этом NULL считаются не равными друг другу.

Уникальный индекс автоматически создается:

* когда вы определяете столбец как **первичный ключ.** Этот индекс также будет и кластеризованным, если в таблице нет кластеризованного индекса.
* когда вы определяете столбец как **unique.**

**Покрывающий индекс -**  позволяет конкретному запросу получить все необходимые данные с листьев индекса без дополнительных обращений к записям самой таблицы. Такой индекс хранит не только проиндексированные столбцы, но и дополнительные. При этом все необходимые данные получаются из индекса, и нет нужды в обращении к куче. Поэтому покрывающие индексы немного быстрее.

**INCLUDE (**postgres**) –** позволяет добавлять в индекс дополнительные столбцы, которые не будут участвовать в поиске, но будут хранится рядом с индексом.



**Индексы по выражениям**

Индексы можно создать не только по столбцу, но и по функции или скалярному выражению с одним или несколькими столбцами таблицы. Это позволяет быстро находить данные в таблице по результатам вычислений.

Например, для сравнений без учёта регистра символов часто используется функция lower:



Этот запрос сможет использовать индекс, определённый для результата функции lower(col1) так:



Если же мы объявим этот индекс уникальным, он не даст добавить в таблицу строки которые различаются только регистром. Т. о. индексы по выражениям можно использовать еще и для нетривиальных ограничений уникальности.

**Частичный индекс**  - индекс, который строится по подмножеству строк таблицы, определяемому условным выражением. Такой индекс содержит записи только для некоторых строк. Лучше всего использовать, когда определенное значение поля встречается редко в таблице.

**Разряженный индекс –** характеризуется тем, что каждый ключ ассоциируется с определенным указателем на блок в сортированном файле данных. Обычно базы данных слишком большие, и делятся на блоки чтобы уместиться в памяти.

**Плотный индекс** в свою очередь отличается тем, что каждый ключ ассоциируется с определённым указателем на запись в сортированном файле данных.

Список всех индексов находится в таблице pg\_index.

**ТИПЫ ИНДЕКСОВ**

Указать тип индекса можно с помощью директивы **USING <тип\_индекса>** перед списком индексируемых полей.

**B-tree –** тип индекса по умолчанию на основе b-дерева.

Он пригоден для данных которые можно отсортировать.

При создании индекса можно явно указывать порядок сортировки. Причем на составном индексе можно указывать разный порядок сортировки для разных полей.

Индексные записи упакованы в страницы. Листья содержат индексируемые ключи и ссылки на строки таблицы. Во внутренних страницах (вершинах) каждая запись ссылается на дочернюю страницу индекса и содержит минимальное значение ключа в этой странице.

Они позволяют ускорить практически любой запрос, условие которого является выражением, состоящим из полей, входящих в индекс, логических операторов и операторов сравнения.

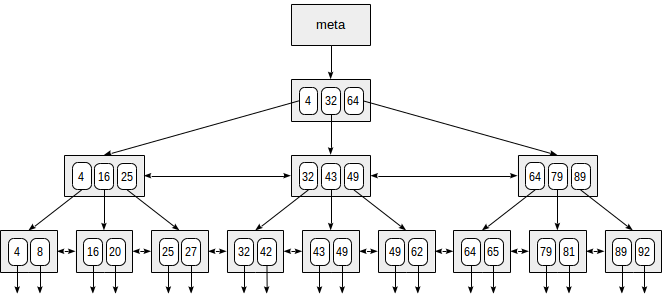
B-tree индексирует также null и поддерживает поиск по условиям is null и is not null. Неопределенные значения располагаются с одного или другого края листовых узлов в зависимости от того, как был создан индекс (nulls first или nulls last).

Так же ускоряет ORDER BY по проиндексированному полю.

Так же может ускорять запросы с LIKE для поиска строк, начинающихся с заданной подстроки.

**Свойства** B-деревьев:

* Они сбалансированы, любую страницу отделяет от корня одно и то же число вершин, поэтому поиск любого значения занимает одинаковое время.
* Они сильно ветвисты. За счет этого глубина деревьев получается небольшой.
* Данные в индексе упорядочены, а страницы одного уровня связаны между собой двунаправленным списком. Поэтому получить данные в промежутке мы можем проходя по списку в одну ил в другую сторону, не возвращаясь каждый раз к корню.



**Hash индекс**



Может быть построен **по столбцам любого типа**, не обязательно имеющим линейный порядок. Такие индексы хранят только хеш-значение индексируемых данных.

Предназначены в первую очередь для нагрузки с большим количеством операций SELECT и UPDATE, которые выполняют сканирование с проверкой равенства для больших таблиц.

Они **могут строиться только по одному столбцу** и **не позволяют проверять уникальность.**

Поддерживают **только оператор равно**, поэтому для поиска интервалов будут бесполезны. Могут участвовать в сканировании индекса по битовой карте и обратном сканировании.  
Не позволяет сортировать и искать значения в промежутке. Не может использоваться в INDEX \_ONLY\_SCAN, так как не хранит значения ключей, а только хеш.

**Bitmap**

Наиболее эффективны для частоповторяющихся данных.

Создается битовая маска и накладывается на таблицу.

заключается в создании отдельных битовых карт (последовательность 0 и 1) для каждого возможного значения столбца, где каждому биту соответствует строка с индексируемым значением, а его значение равное 1 означает, что запись, соответствующая позиции бита содержит индексируемое значение для данного столбца или свойства.

**Реверсивный индекс**

Аналог B-tree индекса, используемый в основном для монотонно возрастающих значений. Биты реверсируются, за счет чего достигается большой разброс данных и хорошая балансировка дерева. Такие индексы не могут быть использованы для поиска по диапазону.

**GiST и SP-GiST**

GiST расшифровывается как «Generalized Search Tree» (Обобщённое поисковое дерево).

Это сбалансированный иерархический метод доступа, который представляет собой базовый шаблон, на основе которого могут реализовываться произвольные схемы индексации.

Он позволяет распределить данные любого типа (Например те, для которых операция сравнения не имеет смысла) по сбалансированному дереву и использовать это дерево для поиска по самым разным условиям.

При построении B-дерева мы сортируем все множество объектов и делим его на части по принципу больше-меньше, при построении GiST индексов можно реализовать любой принцип разбиения любого множества объектов.

Индекс b-tree жестко привязан к семантике сравнения: поддержка операторов «больше», «меньше», «равно» — это все, на что он способен (зато способен очень хорошо!). Но в современных базах хранятся и такие типы данных, для которых эти операторы просто не имеют смысла: геоданные, текстовые документы, картинки…

Позволяет описывать собственные методы индексирования, беря на себя низкоуровневые проблемы (блокировки, страничные структуры и т.д.) и предоставляет свой собственный интерфейс: несколько функций, относящихся не к технической сфере, а к прикладной области.

**GIN**

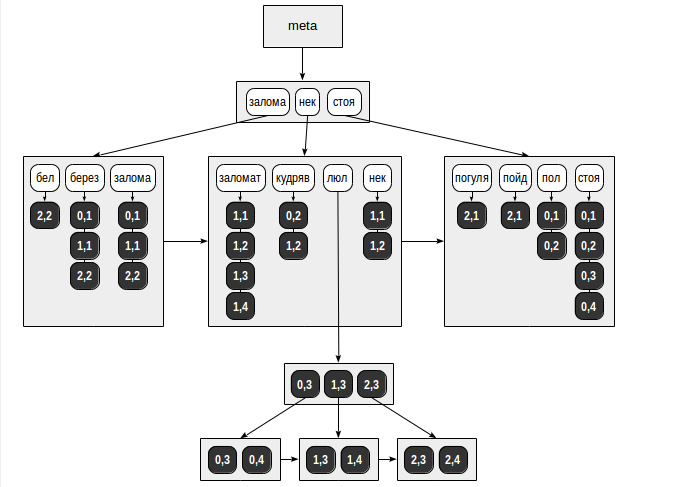
Generalized Inverted Index — это так называемый **обратный** **индекс.**

Он работает с типами данных, значения которых не являются атомарными, а состоят из элементов. При этом индексируются не сами значения, а их составные элементы; каждый элемент ссылается на те значения, в которых он встречается.

**Аналогия**: алфавитный указатель в конце книги, где для каждого термина приведен список страниц, где этот термин упоминается.

Обновление такого индекса как правило дорого обходится.

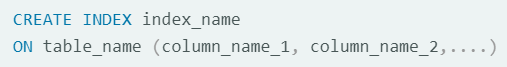
В данном случае строки текста содержат ссылки на строки в бд, в которых они встречаются.

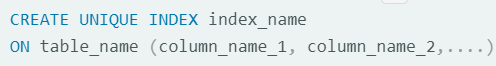


Другой пример использования метода gin — **индексирование массивов**. В этом случае в индекс попадают элементы массивов, что позволяет ускорять ряд операций над ними:

**СОЗДАНИЕ ИНДЕКСОВ**

1. Когда колонки учавствуют в условии where



1. Когда столбец должен иметь уникальные значения.  
   
2. Когда в условии where используется поиск по функции или скалярному выражению с одной или несколькими колонками таблицы.  
   