**Тема 4. Язык запросов SQL. Индексы и Представления.**

**Индексы SQL**

Если хочется быстрее найти какую-то главу или тему, то мы открываем содержание и по указателю на номер страницы быстро находим то, что искали. Любой словарь построен по такому же принципу – слова в словаре уже отсортированы в алфавитном порядке. Это значительно упрощает поиск нужного нам термина.

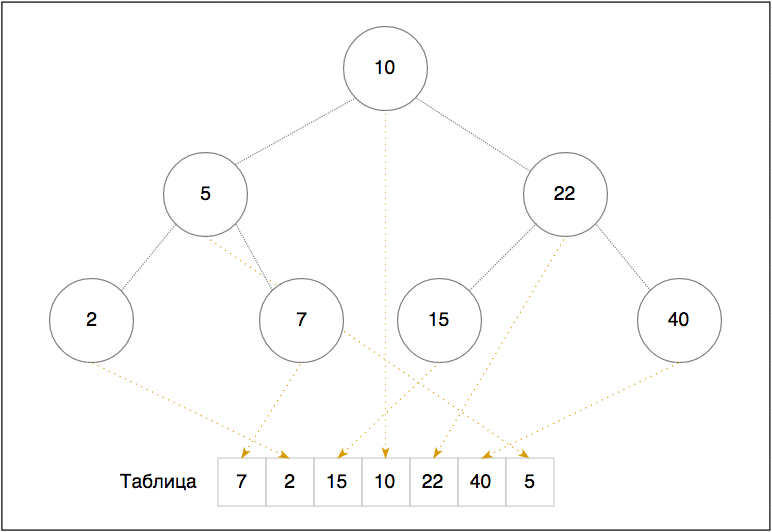
Суть индексов в базах данных такая же – упростить и ускорить поиск нужной нам строки или строк. Без индекса при поиске нужного значения будет проверяться каждая строчка из таблицы. Если значений очень много, то поиск занимает большое количество времени.

Индекс помогает ускорить запрос. Если запросы в БД начинают тормозить, то первым делом думают о создании индекса. Но нужно учитывать, это не всегда правильно. Нельзя воспринимать создание индексов как панацею - несмотря на достоинства, индексы имеют и ряд недостатков. Первый из них – индексы занимают дополнительное место на диске и в оперативной памяти. Каждый раз когда вы создаете индекс, вы сохраняете ключи в порядке убывания или возрастания, которые могут иметь многоуровневую структуру. И чем больше/длиннее ключ, тем больше размер индекса. Второй недостаток – замедляются операции вставки, обновления и удаления записей. Однако алгоритмы построения индексов разработаны таким образом чтобы иметь как можно меньший негативный эффект для указанных операций и даже позволяет выполнять их быстрее.

Помимо этого индекс используется для ограничения уникальных значений в таблице, сортировки, группировки и соединения таблиц.

В этой лекции представлено общее описание об индексах и также затрагиваются их реализация в PostgreSQL.

Индекс представляет из себя отдельную таблицу с отсортированными значениями и ссылками на запись в основной таблице. Сам индекс можно представить как дерево:



Индекс можно создать для любого столбца или представления (view), за исключением столбцов с типами данных для хранения больших объектов: text, image или varchar(max).

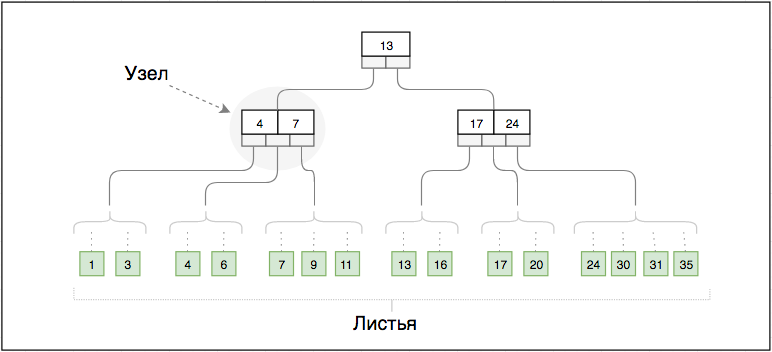
Создается индекс следующей командой:

CREATE INDEX name ON table (column);

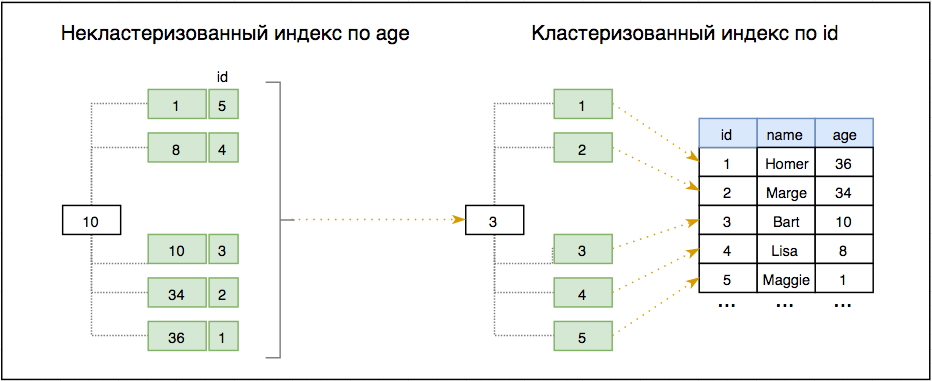
К примеру, у нас есть таблица пользователей и мы создали индекс по колонке age.

CREATE INDEX users\_age\_idx ON users (age);

Теперь мы хотим найти пользователя, которому 15 лет. Что происходит внутри базы данных? Она знает, что на колонку age создан индекс и начнет поиск сначала по индексу, начиная с корня и спускаясь вниз по узлам до тех пор, пока не найдет искомое значение. В итоге поиска мы получаем указатель на строку со всеми данными из таблицы. Вместо того, что-бы проверять каждое значение, которое могло бы занять 7 шагов, мы нашли его за 3 шага.



Дерево состоит из узлов и листьев. Указатели на исходную таблицу хранятся в листьях или могут уже содержать в себе все данные, все зависит от вида индексов. Их бывает двух видов: кластеризованный и некластеризованный.



Кластеризованный индекс уже хранит данные в своих листьях. Он находится в отсортированном виде и создается только один на всю таблицу. Обычно на колонку id, которая является первичным ключом (primary key), по умолчанию создается кластеризованный индекс.

Некластеризованный индекс хранит в своих листьях ссылки на записи кластеризованного индекса или на записи из кучи, если кластеризованного индекса нет.

Куча (heap) это просто неотсортированные данные таблицы.

На самом деле, кластеризованный индекс это не отдельная таблица, а просто отсортированная таблица по выбранной колонке.

При создании индекса можно указать ряд параметров. Давайте рассмотрим только основные и часто используемые параметры (на самом деле, их немного больше).

CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table\_name [ USING method ] ({ column | ( expression ) } [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]) [ WHERE predicate ]

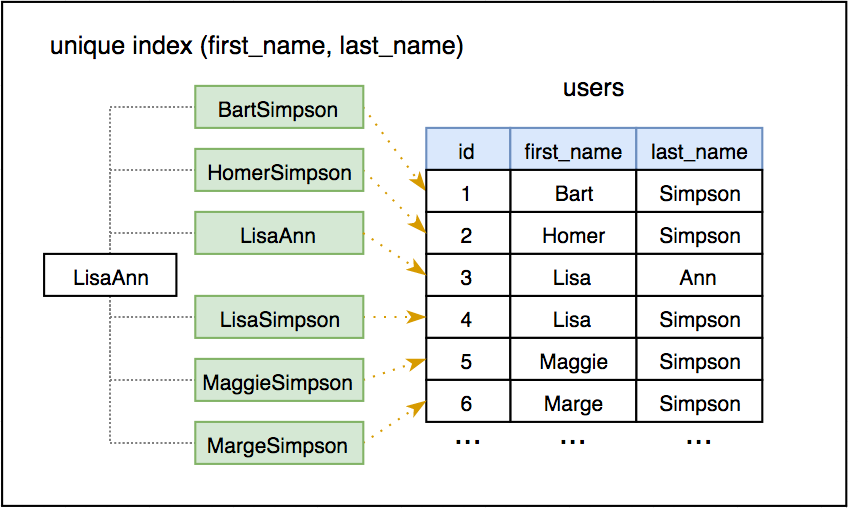
**Уникальный индекс**

CREATE UNIQUE INDEX users\_uid\_idx ON users (uid);

Такой индекс обеспечивает уникальность значений в индексируемой колонке. Также есть возможность создать составной уникальный индекс.

CREATE UNIQUE INDEX users\_full\_name\_idx ON users (first\_name, last\_name);

В этом случае обеспечивается уникальность значений на все колонки, но не на каждую отдельно. Т.е., если вы создадите составной уникальный индекс на поля first\_name и last\_name, то это означает, что повторяющихся имен + фамилий не будет, но разрешается использовать одинаковые либо имя, либо фамилию по отдельности. При создании первичного ключа уникальный индекс создается автоматически.



**Блокировка**

При обычном создании индекса БД блокирует вставку, изменение и удаление в таблице. В некоторых случаях бывает, что индекс создается не быстро, а таблица обновляется очень часто и не хотелось бы блокировать ее изменение. Для этого есть параметр – CONCURRENTLY.

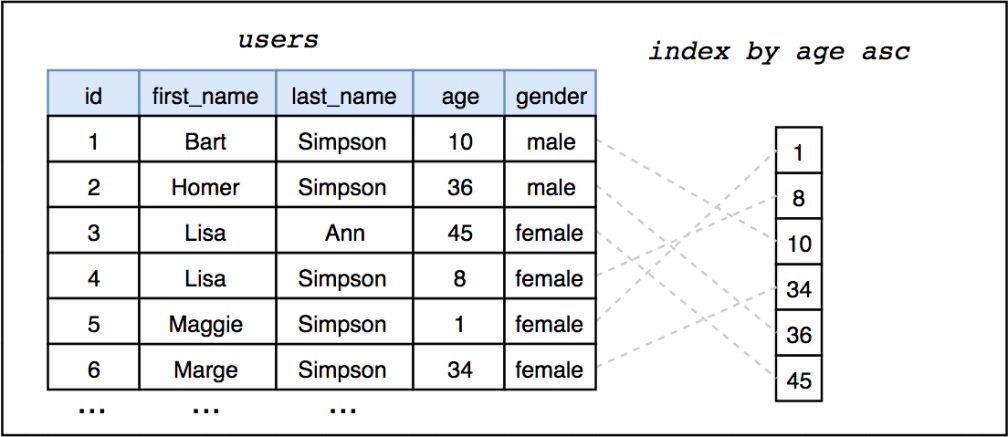
CREATE INDEX users\_age\_\_idx CONCURRENTLY ON users (age);

В таком случае создание индекса не будет блокировать изменение таблицы, но время на само создание увеличится. Некоторое время индекс использоваться не будет.

**Сортировка**

Индекс бывает полезен при сортировке выборки. При создании индекса мы можем указать ему направление сортировки ASC или DESC.

CREATE INDEX users\_age\_\_idx ON users (age) ASC;



После выборки всех записей и указания сортировки по колонке age, то БД воспользуется отсортированным индексом по колонке age и после соберет все данные по связанным записям.

SELECT \* FROM users WHERE gender = 'female' ORDER BY age;

Также при создании индекса мы можем указать куда вставлять записи со значением NULL, добавив параметры NULL FIRST или NULLS LAST.

CREATE INDEX users\_age\_\_idx ON users (age) ASC NULL FIRST;

**Функциональный индекс**

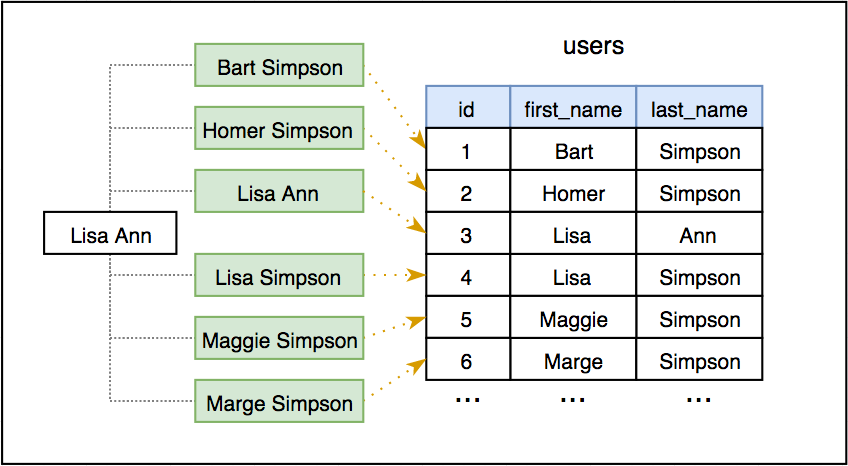
Еще бывают случаи, когда нам индекс требуется не по самому полю, а по результату выражения. Допустим, у нас есть следующий запрос:

SELECT \* FROM users WHERE (first\_name || ' ' || last\_name) = 'Glenn Quagmire';

Здесь функциональный индекс может сильно помочь. Создается он следующим образом:

CREATE INDEX users\_full\_name\_idx ON users (first\_name || ' ' || last\_name);

Функциональный индекс не хранит выражение, а наоборот – результат выражения. Благодаря этому сильно ускоряется выборка, т.к. отпадает необходимость высчитывать выражение для каждой записи. Но есть обратная сторона – сильно падает скорость создания и обновления записи, т.к. рассчитывается новое значение.

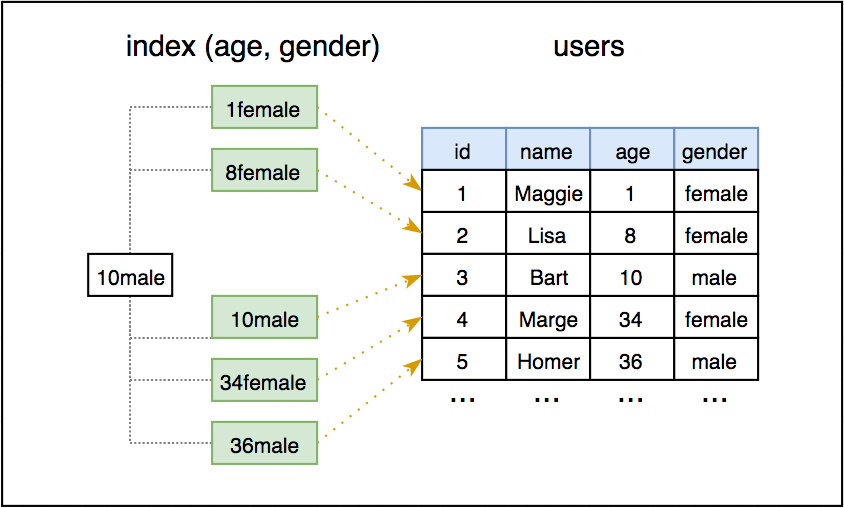


**Составной индекс**

Составной индекс – означает индекс, созданный по одной или более колонок. Т.е. если та же таблица пользователей имеет колонки age и gender, то можно создать индекс на эти две колонки:

CREATE INDEX users\_age\_idx ON users (age, gender);

Тут важно понимать, что порядок важен. Т.е. если мы используем в индексе две колонки (age + gender) то ключи индекса будут выглядеть как: 16male, 20female, 21male и т.д. Это означает, что при поиске по параметрам возраст + пол подсистема склеит их и попытается найти запись по индексу с соответствующим ключом.



**Частичный индекс**

Бывают случаи, что не нужно строить индекс по всей таблице. Например, есть таблица users и в ней есть колонка is\_active. Т.к., обычно, все запросы работают только с активными пользователям, то тогда есть возможность создать индекс только по активным записям:

CREATE INDEX users\_name\_part\_idx ON users (name)

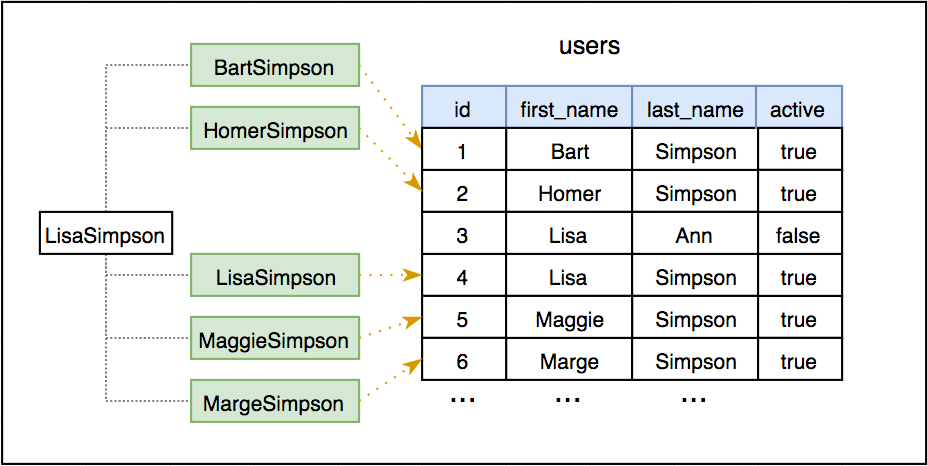
WHERE is\_active = true;

Составные и уникальные индексы тоже могут быть частичными.

CREATE UNIQUE INDEX users\_full\_name\_part\_idx

ON users (first\_name, last\_name)

WHERE is\_active = true;



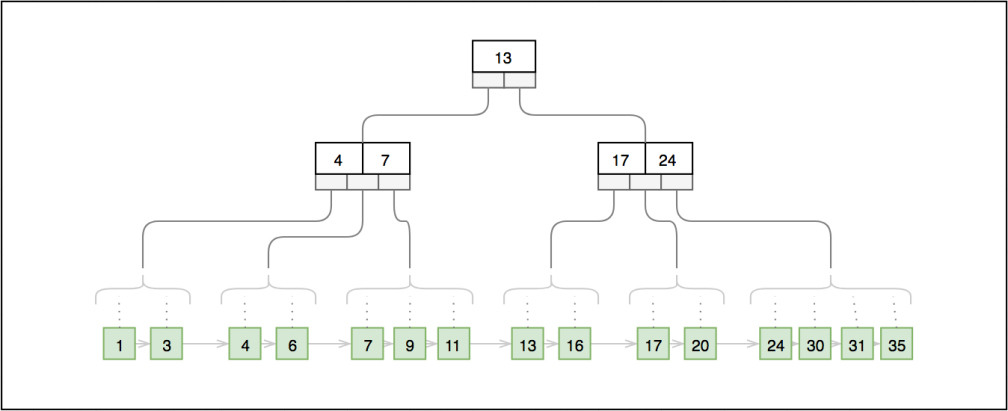
**Индексы PostgreSQL**

Если при создании индекса не указывать его тип, то по умолчанию будет создан индекс типа B-Tree. Однако PostreSQL позволяет создавать индексы таких типов, как B-Tree, Hash, GiST, SP-GiST, GIN и BRIN.

CREATE INDEX name ON table USING **index\_type** (column);

**B-Tree**

B-tree (Balanced Tree) строит индексы используя реализацию сбалансированного дерева. Он может быть использован в условиях сравнения или проверке в диапазоне.

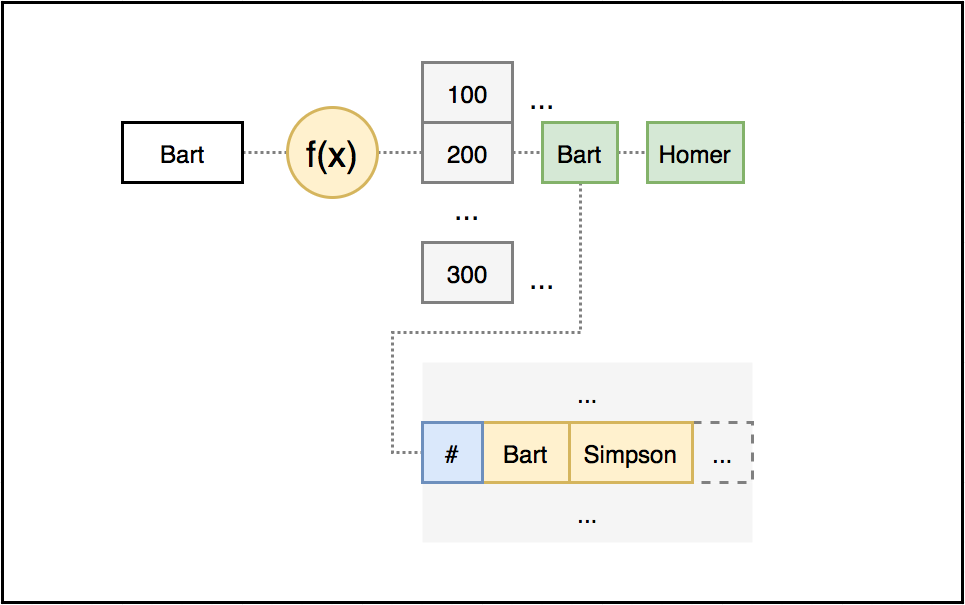


Полезен в следующих случаях:

* операторы сравнения >, <, =, >=, <=, BETWEEN и IN;
* условия пустоты IS NULL и IS NOT NULL;
* операторы поиска подстроки LIKE и ~, если искомая строка закреплена в начале шаблона (например name LIKE 'Lisa%');
* регистронезависимые операторы поиска подстроки ILIKE и ~\*. Но только в том случае, если искомая строка начинается с символа, который одинаков и в верхнем и в нижнем регистре (например числа)`.
* B-деревья могут также применяться для получения данных, отсортированных по порядку.

**Hash**

Hash индексы используются только при условии равенства (name = 'Bart'). При построении hash индекса участвует hash функция, которая принимает значение ('Bart') и на выходе, вычисляя hash (200), распределяет их по секциям. При корреляции объекты внутри секции выстраиваются в цепочку.



Создается следующее командой:

CREATE INDEX name ON table USING hash (column);

**GiST**

GiST (Generalized Search Tree) для построения индексов использует один из нескольких алгоритмов, наиболее подходящих под тип индексируемого поля. Поэтому набор операторов при работе с которыми может быть задействован этот индекс зависит от типа поля. По умолчанию PostgreSQL предоставляет индексы для некоторых типов данных, таких как геометрические типы, сетевые адреса, диапазоны и т.д. Так же этот список можно расширить, установив соответствующие модули.

На базе GiST могут быть реализованы B-деревья, R-деревья и многие другие схемы индексации.

Полезен в следующих случаях:

* типы box, circle и polygon – операторы &&, &>, &<, &<|, >>, <<, <<|, <@, @>, @, |&>, |>>, ~, ~=;
* типы inet и cidr – операторы &&, >>, >>=, >, >=, <>, <<, <<=, <, <=, =;
* тип point – операторы >>, >^, <<, <@, <@, <@, <^, ~=;
* тип tsquery – операторы <@, @>;
* тип tsvector – оператор @@;
* все типы range – операторы &&, &>, &<, >>, <<, <@, -|-, =, @>, @>.

**SP-GiST**

SP-GiST (Space-Partitioned GiST) поддерживает деревья поиска с разбиением, что облегчает разработку широкого спектра различных несбалансированных структур данных, в том числе деревьев квадрантов, а также k-мерных и префиксных деревьев. Общей характеристикой этих структур является то, что они последовательно разбивают пространство поиска на сегменты, которые не обязательно должны быть равного размера. При этом поиск, хорошо соответствующий правилу разбиения, с таким индексом может быть очень быстрым.

Полезен в следующих случаях:

* тип point – операторы >>, >^, <<, <@, <@, <@, <^, ~=;
* типы box – операторы &&, &>, &<, &<|, >>, <<, <<|, <@, @>, @, |&>, |>>, ~, ~=;
* все типы range – операторы &&, &>, &<, >>, <<, <@, -|-, =, @>, @>;
* все типы text – операторы <, <=, =, =>, ~<=~, ~<~, ~>=~, ~>~.

**GIN**

GIN (Generalized Inverted Index) индексы применимы к составным типам, работа с которыми осуществляется с помощью ключей. Это массивы, jsonb и tsvector. Как и GiST индексы, они могут реализовать один из нескольких алгоритмов. И стандартный набор можно также расширить, установив модели.

Полезен в следующих случаях:

* массивы – операторы &&, <@, =, @>;
* тип jsonb – операторы ?, ?&, ?|, @>;
* тип tsvector – операторы @@ и @@@.

GIN индекс может быть создан только для определенных полей jsonb поля.

**BRIN**

BRIN (Block Range Index) предназначается для обработки очень больших таблиц, в которых определенные столбцы некоторым естественным образом коррелируют с их физическим расположением в таблице. Зоной блоков называется группа страниц, физически расположенных в таблице рядом; для каждой зоны в индексе сохраняется некоторая сводная информация. Например, в таблице заказов магазина может содержаться поле с датой добавления заказа, и практически всегда записи более ранних заказов и в таблице будут размещены ближе к началу; в таблице, содержащей столбец с почтовым индексом, также естественным образом могут группироваться записи по городам.

Полезен в следующих случаях:

* типы box – операторы &&, &>, &<, &<|, >>, <<, <<|, <@, @>, @, |&>, |>>, ~, ~=;
* все типы range – операторы &&, &>, &<, >>, <<, <@, -|-, =, @>, @>;
* операторы сравнения >, <, =, >=, <=.

**Представления SQL**

Представление (VIEW) - объект данных, который не содержит никаких данных его владельца. Это - тип таблицы, чье содержание выбирается из других таблиц с помощью выполнения запроса. В этой лекции, вы узнаете что такое представления, как они создаются, и немного об их возможностях. Использование представлений основанных на улучшенных средствах запросов, таких как объединение и подзапрос, разработанных очень тщательно, в некоторых случаях даст больший выигрыш по сравнению с запросами.

Типы таблиц, с которыми вы имели дело до сих пор, назывались - базовыми таблицами. Это - таблицы, которые содержат данные. Однако имеется другой вид таблиц: - представления. Представления - это таблицы чье содержание выбирается или получается из других таблиц. Они работают в запросах и операторах DML точно также как и основные таблицы, но не содержат никаких собственных данных. Представления - подобны окнам, через которые вы просматриваете информацию, которая фактически хранится в базовой таблице. Представление - это фактически запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

Создается представление командой CREATE VIEW. Она состоит из слов CREATE VIEW (СОЗДАТЬ ПРЕДСТАВЛЕНИЕ), имени представления которое нужно создать, слова AS (КАК), и далее запроса, как в следующем примере:

CREATE VIEW Londonstaff AS SELECT \* FROM Salespeople WHERE city = 'London';

Теперь мы имеем представление, называемое Londonstaff. Можно использовать это представление точно так же как и любую другую таблицу. Она может быть запрошена, модифицирована, вставлена в, удалена из, и соединена с, другими таблицами и представлениями. Чтобы сделать запрос такого представления, достаточно выполнить следующую команду:

SELECT \* FROM Londonstaff;

Когда вы указываете БД выбрать (SELECT) все строки ( \* ) из представления, он выполняет запрос содержащий в определении - Loncfonstaff, и возвращает все из его вывода. Имея предикат в запросе представления, можно вывести только те строки из представления, которые будут удовлетворять этому предикату. Преимущество использования представления, по сравнению с основной таблицы, в том, что представление будет модифицировано автоматически всякий раз, когда таблица лежащая в его основе изменяется. Содержание представления не фиксировано, и переназначается каждый раз когда вы ссылаетесь на представление в команде. Если вы добавите завтра другого, живущего в Лондоне продавца, он автоматически появится в представлении. Представления значительно расширяют управление вашими данными. Это - превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице. Если вы хотите чтобы ваш продавец был показан в таблице Продавцов, но при этом не были показаны комиссии других продавцов, вы могли бы создать представление с использованием следующего оператора:

CREATE VIEW Salesown AS SELECT snum, sname, city FROM Salespeople

Другими словами, это представление - такое же как для таблицы Продавцов, за исключением того, что поле comm, не упоминалось в запросе, и следовательно не было включено в представление.

Представление может теперь изменяться командами модификации DML, но модификация не будет воздействовать на само представление. Команды будут на самом деле перенаправлены к базовой таблице:

UPDATE Salesown SET city = 'Palo Alto' WHERE snum = 1004;

Его действие идентично выполнению той же команды в таблице Продавцов. Однако, если значение комиссионных продавца будет обработано командой UPDATE:

UPDATE Salesown SET comm = .20 WHERE snum = 1004;

она не будет выполнена, так как поле comm отсутствует в представлении Salesown. Это важное замечание, показывающее что не все представления могут быть модифицированы.

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

DROP VIEW <view name>

Помните, вы должны являться владельцем представления чтобы иметь возможность удалить его.