**Объекты БД**

**Индексы** – структура данных, позволяющая эффективнее находить и извлекать данные (быстрая выборка). Посредством установления связи между собственным ключом и строками таблицы.

**Create** **Index** <Имя> **On** <Таблица (аргумент, <Порядок сортировки>) >

Эффективность поиска достигается за счет создания списка всех значений для заданного аргумента, организованного определенным образом (зависит от вида индекса), который будет содержать указатель на данные в таблице. То есть любой вид индекса устанавливает соответствие между собственным ключом, которые организованы определенным образом и строками исходной таблицы (индекс не хранит данные!).

**Тип индексов**:

*B-tree* (balanced tree) – тип индекса по умолчанию, так как является наиболее универсальным типом. Поддерживает операции > = < и др. схожие типа Like (поддерживает ‘abc%’ но не ‘%abc’ так как неясно, что именно ищем и все-равно придется проверять все строки) или In.

Характеристика B-tree структуры:

1. Узел хранит более одного значения
2. Не является бинарным деревом (кол-во ребер коррелирует с кол-вом значений, хранимых в узле)
3. Жестко сбалансированное (естественная балансировка благодаря особенностям его построения)

Материалы о структуре B-tree [1](https://www.youtube.com/watch?v=K1a2Bk8NrYQ&ab_channel=SpanningTree), [2](https://www.youtube.com/watch?v=WXXetwePSRk&ab_channel=VolodyaMozhenkov), [3](https://www.youtube.com/watch?v=H01NmrEFATI&ab_channel=%D0%9F%D0%B0%D0%B2%D0%B5%D0%BB%D0%A1%D0%BE%D1%80%D0%BE%D0%BA%D0%B8%D0%BD).

B-tree индексы берут за основу структуру B-tree дерева, в качестве значений выступают страницы индексов, а каждый уровень дерева образует двусвязный список.

*Hash - индекс* – для нахождения точного совпадения (поддерживает только оператор ‘=’ ), не поддерживает сортировки и поиски диапазонов. Сложность поиска O(1), но он неуниверсален, индексы не пишутся в журнал транзакций, активно используются СУБД, но не рекомендуются к использованию за исключением временных таблиц.

*Специализированные индексы:* GiST(обобщенное дерево поиска) – для геометрических типов данных и для организации полнотекстового поиска. GIN (обобщенный обратный) – для массивов и наборов значений. SP-GIST (GiST с двоичным разбиением пространства) – для данных, которые подразумевают естественную упорядоченность. BRIN (блочно-диапазонный) – для больших объемах данных, которые подразумевают естественную упорядоченность.

Виды индексов:

1. *простой индекс* – содержит один аргумент
2. *составной индекс* – включает несколько аргументов
3. *индекс по выражению* – в качестве индексов выступает результат вычисления какой-либо функции по определенному аргументу
4. *частичный индекс –* индекс для некоторого подмножества данных, генерируется для строк, которые соответствуют заданному условию
5. *уникальный индекс* – по умолчанию создается для ограничений Unique и Primary Key, для обеспечения уникальности
6. *покрывающий индекс* – индекс, которому не нужно обращаться к таблице для удовлетворения запроса, то есть он уже содержит все нужные данные.

При объявлении индекса по нескольким аргументам его сортировка происходит сначала по первому аргументу, а для повторяющихся значений первого аргумента будут отсортированы по второму (работает как order by по нескольким аргументам). В составном индексе наиболее селективные индексы необходимо устанавливать на первое место (отбрасывает больше значений, следовательно, меньше проверок для последующих условий).

Типы индексов:

1) Кластеризованный - такой индекс задает физический порядок хранения данных в таблице (сортирует таблицу по индексу). Такой индекс может быть только один. То есть, в зависимости от того, есть ли кластеризованный индекс, таблицы подразделяются на кластеризованные (отсортированные) или heap-таблицы.

2) декластеризованные - индекс в «стандартном» понимании (не хранит данные, является указателем, можно создавать больше 1).

*Минусы индексов:*

1) Ускоряют получение данных, но замедляют обновление/вставку данных, так как новые данные нужно добавить в исходную таблицу, затем на их основании достроить индекс. Исходя из этого индексы лучше использовать для БД в хранилищах данных, получающих плановые обновления, а не для тех, которые постоянно обновляются, поскольку индексы могут не успевать обновляться за новыми данными и станут бесполезны. Для оптимизации работы индекса существуют HOT-обновления – Если изменяется атрибут строки, который не входит в индекс, то создается версия строки, а новая версия индекса не создается, при этом, для получения актуальных изменений индекс пойдет по цепочке версий строк до нужной. Этот механизм возможен при условии, что новая версия строки помещается в одну страницу.

2) Занимают место на диске (достаточно много).

Рекомендации: не использовать на малых таблицах, делать индексы под запросы, не использовать индексы на аргументах с частым null (низкая селективность и доп. затраты на хранение), не использовать индексы с частыми обновлениями данных.

В общем решение использовать индексы или нет зависит от множества факторов и эффективность их использования будет варьироваться в разных сценариях.

**Особенности и оптимизация запроса**

Жизненный цикл запроса состоит из этапов разбора (синтаксического и семантического), планирования и исполнения.

На этапе планирования оптимизатором запросов выбирается оптимальный путь исполнения запроса на основании имеющейся статистики, в частности на выбор способа сканирования таблицы (последовательное, индексное, по битовой карте и др.)

Выбор стратегии сканирования таблицы: Оптимизатор опирается на имеющиеся данные в статистике о частоте запрашиваемых данных и затратах, которые СУБД понесет в результате исполнения.

Наличие индекса не гарантирует использование индексного сканирования. Индексы имеют накладные расходы в виде необходимости «заглядывать» в таблицу для получения данных и оптимизатор на основании статистики может посчитать, что дешевле произвести последовательное сканирование.

Пример: Есть 10.000 записей, строим индекс по age (распределение значений 0-100), хотим получить все имена, age которых > 10, то есть примерно 70% всех строк из таблицы. При таких условиях есть вероятность, что оптимизатор не будет обращаться к индексу, а просто переберет все 10.000 записей таблицы, т.к. для использования индекса ему нужно для каждого значения age сходить в таблицу и посмотреть имя, которое соответствует этому индексу (а если оно еще и не закэшировано и лежит на диске, то вообще труба).

\*Покрывающие индексы не требуют «заглядывать» в таблицу, т.к. они уже имеют всю необходимую информацию.

\*В PostgreSQL нет хинтов (невозможно явно сказать СУБД как строить план запроса, какой индекс выбрать).

**Селективность индекса** – метрика, которая описывает насколько эффективно индекс может фильтровать данные в таблице. Высокая селективность – использование индекса дает возможность отфильтровать бо’льшее кол-во значений, т.е. результатом выборки будет наименьшее кол-во значений.

Селективность = кол-во уникальных значений / общее кол-во строк.

Пример низкой селективности: индекс имеет 2 уникальных значения (М/Ж), общее кол-во строк 10.000, поиск по этому индексу отсечет только половину строк, остальные 5.000 придется пробегать последовательным поиском – пример актуален в случае равного распределения полов.

На селективность влияет распределение данных в таблице, то есть заведомо высокоселективный индекс может иметь низкую селективность за счет неравномерного распределения данных.

Пример: индекс по age (ожидаемые значения от 0 до 100 лет), то есть при поиске точного значения возраста должен оставаться 1% от записей в таблице, которые нужно будет проходить последовательно. Но в таблице данные распределены неравномерно и, допустим, на значение 25 мы имеем 85% строк таблицы, что делает необходимым последовательно пройтись по всем этим строкам последовательно.

То есть при оптимизации нужно обращать внимание на кол-во уникальных значений, которые может принимать индекс, область поиска т.е. поиск по age = 38 даст меньше записей, чем age >38 и общее распределение данных в таблице, т.е. индекс может быть потенциально селективным, но данные в таблице распределены неравномерно и большинство данных, например, приходится именно на age = 38.

Индексы по выражениям – для того, чтобы оптимизатор использовал индексы в выражениях типа where lower(column\_name) недостаточно построить индекс по column\_name из-за разницы хранимых значений индекса и запрашиваемых значений, необходимо построить индекс именно для этого выражения.

Составной индекс – индекс, построенный по нескольким атрибутам, что позволяет более эффективно использовать фильтрацию по нескольким атрибутам одновременно. При построении такого индекса нужно учитывать последовательность атрибутов, а именно, наиболее селективные атрибуты необходимо устанавливать в начале.

Общий план оптимизации запросов:

1. Анализ запрашиваемых данных – определить необходимость запрашиваемых атрибутов, наличие лишних join и подзапросов.
2. Если есть подзапросы, возможно, заменить на JOIN (тем более коррелирующие)
3. Анализ плана запроса с помощью explain/explain analyze (общий план и способ сканирования).
4. Рассмотреть построение индекса с учетом особенностей выше
5. Рассмотреть возможность оптимизации структуры таблиц (нармализация/денормализация).

[**Представления / View**](https://habr.com/ru/articles/47031/)

Объект БД, виртуальная таблица, основанная на результате выполнения запроса к одной или нескольким таблицам. Является псевдонимом для запросов select, хранит в себе логику запроса, но не содержит данных (кроме материализованных представлений), может использоваться в качестве источника данных, как обычная таблица - по сути это сохраненный запрос в виде объекта БД, который используется как источник данных.

- позволяет сокращать сложные запросы  
- позволяет подменить реальную таблицу (скрыть логику агрегации или исключить некоторые атрибуты из выборки), также выступает в качестве дополнительного слоя абстракции, к которому также можно применять правила безопасности.

**Редактирование представления**:

Стандартный синтаксис представлен [Create Or Replace View], но такая конструкция позволяет осуществлять не все операции редактирования, такой конструкцией невозможно:

Удаление атрибута  
 Редактирование имени атрибута  
 Изменение порядка следования атрибутов

Конструкция [Create or replace view] позволяет только добавлять новые атрибуты в конец, на остальные операции ограничений для View нет. В случае необходимости внесения изменений из категории «невозможные» необходимо явно удалить представление и пересоздать его.

**Обновляемые и необновляемые представления:**

Обновляемым является представление, в которые можно вносить изменения так, как если бы изменения вносились в обычную таблицу. Попытка вставки в неизменяемое представление приведет к ошибке.

Обновляемым считается в представление, в котором не присутствуют:

Базирование запроса на нескольких таблицах (join, подзапросы, unionи др.)  
 distinct, group by, having, union, intersect, except, limit  
 Нет оконных и агрегирующих функций

Общий смысл таких ограничений – возможность однозначно сопоставить данные с исходной таблицей (т.е. восстановить исходную таблицу).

Пример простого view:

**CREATE VIEW** adults\_view AS  
**SELECT** id, name, age  
**FROM** persons  
**WHERE** age >= 18

Данное представление будет отображать данные из таблицы person, которые соответствуют условию age>=18, но при этом позволяет вставлять данные независимо от этого условия (то есть данные с любым возрастом), при этом в последующих выборках они будут не видны.

Для того, чтобы задать ограничение на вставку данных (чтобы они соответствовали ограничению представления) необходимо использовать **With [ lockal| Cascade ] check option –** позволяет вставлять или обновлять данные в таблице, которые будут видимы в этом представлении (local – для текущего представления, cascade – для всех представлений, включая те, которые основаны на текущем представлении).

**Материализованное представление –** физически сохраняет результат запроса исполнения запроса на диске, что позволяет ускорить его выполнение за счет отсутствия необходимости исполнения запроса для получения данных.

**Хранимые процедуры** – это фрагмент кода, который сохраняется в базе данных и может быть вызван и выполнен по запросу. Могут содержать элементы императивного стиля (циклы, условия, блоки кода и переменные) – для написания могут быть использованы языки C, Python, plpgsql, SQL и др.

Плюсы:

Все преимущества инкапсуляции (сокрытие реализации, интерфейс для взаимодействия, повторное использование кода, безопасность данных, легкость изменения реализации).

Клиент обращается не к базе, а к функции, что дает возможность скрыть от пользователя структуру таблицы.

Обеспечивают механизмы защиты – можно ограничить права на доступ к функции или к таблице, чтобы клиент пользовался только функцией и не касался данных.

Исполнение логики на стороне СУБД – нужный результат сразу получается клиентским приложением, не нужно гнать данные по сети и обрабатывать локально – будет большая нагрузка на ядро сервера, но уменьшит нагрузку на сеть.

Компилируемы и хранятся на стороне БД – запрос нужно каждый раз прогонять через оптимизатор, а процедура скомпилируется (то есть будет выбран оптимальный план выполнения в виде кода) и он будет использоваться при каждом вызове.

Минусы:

Грузит сервер, т.к. логика исполняется там

Трудно мигрировать с одной субд на другую (у всех разный синтаксис, придется переписывать).

**Функции –** частный случай хранимой процедуры (т.е. обладает всеми ее свойствами, но имеет иные задачи).

*Различия:*

Функция вызывается в операторе SQL, то есть используется только в контексте запроса, процедура с помощью call и не привязана к запросу.

Функция всегда возвращает значение, процедура может не возвращать ничего.

Функция используется для вычисления и возвращения результатов, процедуры используются для более обширных действий с бд: изменение данных в таблице, управление вызовом операторов, координация транзакций.

Функции не могут содержать TCL-команды и vacuum, но при этом используются в контексте транзакций.

|  |  |  |
| --- | --- | --- |
|  | **Синтаксис функций на SQL** | **Синтаксис функций на PLPGSQL** |
| **Синтаксис** | **CREATE** **FUNCTION** name(parametrs)  **RETURNS** data\_type **AS** **$$**  --тело функции  **$$** **LANGUAGE** SQL  Language можно писать после returns | **CREATE** **FUNCTION** name(parametrs)  **RETURNS** data\_tipe AS $$  **BEGIN**  --тело функции  **RETURN** - -используется для явного возвращения значения.  **END**  **$$** **LANGUAGE** **PLPGSQL** |
|  | Типы аргументов для функции:  in/out/inout/variadic(массив)/default  Возвращаемые значения явно задаются с помощью **returns,** его можно не указывать, в случае, если явно заданы переменные out, в таком случае агрументы будут возвращены в том же порядке.  **Returns** data\_tipe – возврат через out параметры одно значение  **Returns** **setoff** data\_tipe – вернет множество значений указанного типа  **Returns** **setoff** record – при возвращении значений нескольких типов  **Returns** **setoff** table\_name – вернет все столбцы из указанной таблицы  **Returns** **table** (column\_names) – возвращает указанные столбцы | Типы аргументов и задание возвращаемых значений полностью соответствуют функции на языке SQL.  **PLPGSQL** позволяет предоставляет использовать возможности процедурных языков: создавать переменные, использовать ветвления, циклы и др.  **Return** используется как оператор, возвращающий значение, для возвращения множества используется **Return query** (сразу возвращает результирующий набор) **или return next** (последовательно добавляет строки в результирующий набор)  Пример:  **Create** **function** avg\_age() **returns** int **as** **$$**  **Begin**  **Return** **avg**(age) **from** persons;  **End;**  **$$** **language** **plpgSQL**  **Декларирование переменных:**  Осуществляется в блоке **declare**  (Присвоение значений переменным с помощью = или :=)  Если переменные декларируются с помощью аргументов, то им должны быть явно присвоены значения, иначе они будут равны null.  **CREATE** **FUNCTION** name(parametrs)  **RETURNS** data\_tipe AS $$  **Declare**  Name type;  **BEGIN**  --тело функции  **RETURN** - -используется для явного возвращения значения.  **END**  **$$** **LANGUAGE** **PLPGSQL**  **Ветвление:**  **If** expression **then** action  **Elsif** expression **then** action  **Else** action  **End if;**  **Циклы:**  **While** expression  **Loop**  -- тело цикла  **End loop;**  **Loop**  **Exit when** expression  -- тело цикла  **End loop;**  **For** счетчик **in[Reverse]** num1**..**num2 **[By** шаг**]**  **Loop**  -- тело цикла  **End loop;**  **Continue when** выражение – прерывает текущую итерацию  **Exit / Exit when** – выход из цикла (работает как break)  **Return next** Выражение – используется с множеством, чтобы выводить его промежуточные результаты  Исключения:  **RAISE [level] ‘Сообщение’, имя аргумента**  Уровень серьезности ошибки – DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION (абортирует транзакцию), warning и exception пишутся в логи сервера по умолчанию.  **EXCEPTION WHEN** условие **THEN** логика – перехват исключения |

**Триггеры** – объект бд, с помощью которого назначаются действия в ответ на изменения в БД. Объект триггера всегда связан с функцией, т.к. именно она будет отвечать за «действие».

Общий синтаксис:

**Create** **Trigger** trigger\_name [**Condition**]   
**on** table\_name  
**for** **each** [**row/statement**]  
**execute** function\_name()

**Row/statement –** row сработает для каждой измененной строки, statement сработает для операции. Statement требует указания таблицы с помощью referencing.

**Condition –** указывает на то, когда триггер должен сработать по отношению к операции, примененной к таблице. Конфигурируется [Before| After | Instead of] и указанием операции [Insert | Update | Delete].

Триггеры требуют специально созданные функции с Returns trigger, языком plpgsql и их реализация часто сопровождается обращением к переменным типа [new | old | Tg\_op] и др.

**Оконные функции –** функция, выполняющая вычисления по набору строк. Позволяют обрабатывать группы строк без образования группировок в результирующем наборе, т.е. оставляют все строки исходной таблицы и добавляют к ней результат вычислений для каждой строки.

Для каждой строки вычисляется «партиция» - набор строк, которые используются для исполнения вычислений (по принципу группировки), затем для каждой такой группы применяется указанная функция.

  
Оконные функции исполняются после всех операторов, кроме order by.

Синтаксис:

**Select** функция [**filter** (**where** условие)] **over** (**partition by** атрибут [**order by** атрибут] [**rows**|**range** диапазон строк])

Filter – фильтрует строки в каждом окне, для которых будет выполнена агрегатная функция.

Partition by – атрибут, по которому будет производиться группировка для агрегирующей функции (eсли over без указания partition by, то вся таблица будет считаться как одна партиция)

Order by – упорядочивает строки внутри окна, автоматически выставляется диапазон **range between unbounded preceding and current row**, если он явно не задан.

**Rows/range** – параметры, задающие диапазон окна для расчётов в текущей партиции, всегда используются с order by.

Общий вид: **range between** <1 граница> **and** <2 граница>

Row – использует физическое положение строки, то есть каждая строка является уникальным диапазоном, независимо от уникальности ее значения (уникальность определяется по параметрам сортировки).

Опции для границ:

**UNBOUNDED PRECEDING** - все строки, предшествующие текущей

**N PRECEDING** - N строк до текущей строки

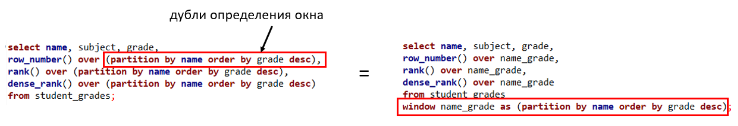
**CURRENT ROW** - текущая строка

**N FOLLOWING** - N строк после текущей строки

**UNBOUNDED FOLLOWING** - все последующие строки

Range – использует значения в качестве разделителя диапазонов, его формирует не одна строка, а совокупность неуникальных значений, сгруппированных в один диапазон. Результатом вычисления для такого диапазона будет агрегированное значение из всех включенных в него кортежей.

N preceding и N following не работают при сортировке по тексту, с остальными типами задают диапазоны значений, а не строк, которые будут учитываться при расчётах исходя из текущего значения.



**window** – для сокращения синтаксиса, в случае, когда код после over повторяется.

*Агрегирующие* – выполняют арифметические операции. Такие же, как обычные агрегирующие функции.

*Ранжирующие* – Присваивает «ранг» каждому кортежу внутри партиции по определенному критерию. Обязательно условие - order by.

Row\_number() – нумерует кортежи в рамках партиции.

Rank() – Присваивает ранг каждой строке кортежа исходя из значения условия сортировки. Для одинаковых значений в строке сортировки будет присвоен одинаковый ранг, следующий ранг увеличивается на кол-во строк с одинаковым рангом.

Dense\_rank() – возвращает ранг каждой строки, одинаковые нумеруются одним рангом, следующий идет по порядку.

*Функции смещения* – Получение значений из предыдущих или последующих строк.

Lag(атрибут, шаг, значение, если null) – обращается к данным предыдущих строк окна.

Lead(атрибут, шаг, значение, если null) - обращается к данным последующих строк окна.

First\_value(атрибут) – Первое значение в окне

Last\_value(атрибут) – Последнее значение в окне

Материалы [1](https://habr.com/ru/companies/otus/articles/490296/), [2](https://sql-academy.org/ru/guide/partitions).

**Sequences – последовательности**

Использование типа данных serial приводит к автоматическому созданию последовательности и привязки ее к определенному атрибуту. То есть атрибуту присваивается int и привязывается nextval значение созданной последовательности в качестве default constraint.

Если делать это вручную, то нужно при создании таблицы навесить constraint DEFAULT nextval(‘имя последовательнсоти’) – для того, чтобы он автоматически вставлялся, если значение не установлено.

*Создание последовательности:*

CREATE SEQUENCE [имя\_последовательности]  
INCREMENT BY [шаг]  
MINVALUE [минимальное\_значение]  
MAXVALUE [максимальное\_значение]  
START WITH [начальное\_значение]  
OWNED BY [связанный\_атрибут]  
CACHE [количество\_значений\_в\_кэше]  
[CYCLE | NO CYCLE];

select nextval('seq') – возвращает следующее значение последовательности

select currval('seq') – возвращает текущее значение последовательности

select lastval() – возвращает последнее значение любой последовательности в текущей сессии

select setval('seq', 1, true) – установка текущего значения последовательности (true – установленное значение становится текущим, false – сначала применить замененное, затем установленное).

Тип данных serial – это сгенерированная БД последовательность, проблема serial в том, что можно руками вставить значение и когда значение последовательности доберется до этого значения, то произойдет ошибка вставки, так же при ограничении прав доступа на последовательность так же произойдет ошибка вставки из-за невозможности получить следующее ее значение.

Начиная с 10 версии автоинкрементируемый тип данных можно задать с помощью **generated always/by default as identity [Опции как при создании последовательности] –** рекомендуется к использованию (входит в стандарт SQL).

**Always –** не дает вручную вставить данные в последовательность, **default –** позволяет ручную вставку.

**Insert into** table () **overriding system value values** ()– позволяет вставлять собственные значения в обход **always**