**Объекты БД**

**Индексы** – структура данных, позволяющая эффективнее находить и извлекать данные (быстрая выборка).

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

Эффективность поиска достигается за счет создания списка всех значений данного 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 и строками исходной таблицы (индекс не хранит данные!).

**Тип индексов**:

*B-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.

Хранит отсортированный список ключей. Сложность поиска O(LogN).

Индекс в данной структуре хранится в виде последовательности страниц, данные внутри страниц отсортированы. Корень и ветви содержат множества ссылок на страницы индекса, а листья сами страницы, которые содержат указатели на данные в таблице.

Каждый уровень дерева образует двусвязный список. B-tree является сбалансированным, сортированным небинарным деревом. [Видео по структуре.](https://www.youtube.com/watch?v=y-Wtyvme4gE&ab_channel=JPoint%2CJoker%D0%B8JUGru)

*Hash - индекс* – для нахождения точного совпадения (поддерживает только оператор ‘=’ ), не поддерживает сортировки и поиски диапазонов. Сложность поиска O(1), но он неуниверсален.

*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:

Различают *простой индекс* – содержит один аргумент, *составной индекс* – включает несколько аргументов, *индекс по выражению* – в качестве индексов выступает результат вычисления какой-либо функции по определенному аргументу, *частичный индекс –* индекс для некоторого подмножества данных, генерится для строк, которые соответствуют заданному условию, *уникальный индекс* – по умолчанию создается для ограничений Unique и Primary Key, для обеспечения уникальности, *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).

На эффективность использования индекса влияет его селективность, чем меньше кол-во индексов, которые соответствуют условию, тем выше селективность индекса в данном запросе. На селективность индекса влияет запрос и данные в таблице. Например, запрос по условию ‘=’ даст большую селективность нежели > или <, так как, вероятнее всего, под это условие попадет меньше индексов. В запросах с > или < , чтобы оценить селективность, нужно понимать распределение данных в таблице и насколько велика будет селективность для этого запроса.

Так же следует учитывать, что составной индекс не дает существенного прироста при поиске только по второму аргументу из-за особенности организации в нем.

Так же нужно учитывать, что индекс занимает достаточно много места на диске и ускорение на незначительный промежуток времени может не стоить того кол-ва места, которое занимает индекс.

Типы индексов – 1) кластеризованные и 2) декластеризованные

Такой индекс задает физический порядок хранения данных в таблице (хранит строки таблицы в листьях), по сути сортирует таблицу. Данные хранятся вместе с индексом. Такой индекс может быть только один. То есть, в зависимости от того, есть ли кластеризованный индекс, таблицы подразделяются на кластеризованные (отсортированные) или heap-таблицы.

По сути это индекс в «стандартном» понимании (не хранит данные, является указателем, можно создавать больше 1).

*Минусы индексов:*

1) Ускоряют получение данных, но замедляют обновление данных, так как новые данные нужно добавить в исходную таблицу, затем добавить в нужное место в индексе. 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.

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.

Оптимизатор самостоятельно определяет способ сканирования (последовательное, индексное, по битовой карте), в зависимости от эффективности того или иного способа, он представляет все методы решения задачи и выбирает наилучший по производительности.

Перебор по таблице читает данные сразу, индексу нужно перейти в таблицу и вычитать данные, для 1 отдельной строки это дольше усиливается при фрагментации

[**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.

Используется для сокрытия реализации запроса (инкапсуляция) – используется как интерфейс для доступа к БД, но реализация невидима для пользователя. Выступает как дополнительный слой абстракции для обеспечения безопасности данных, если запретить пользователям обращаться к таблице напрямую и использовать представления.

После создания представления есть возможность только добавлять новые аргументы (только в конец таблицы т.к меняется порядок), в уже созданном представлении нельзя переименовывать столбцы, удалять, менять порядок следования. В случае необходимости таких изменений в представлении нужно создать новое представление [Create Or Replace View] – опция or replace удалит старое представление с таким же именем и создаст новое с нужными изменениями. Если необходимо применить недоступные операции, то сначала нужно явно удалить представление, а затем создать новое с изменениями.

Обновляемые/Необновляемые представления – Предоставляют или не предоставляют возможность изменения данных в таблице, используя представление в качестве источника.

Чтобы представление было обновимо, СУБД должна быть способна для каждой строки представления найти соответствующую строку в исходной таблице, а для каждого обновляемого столбца представления - соответствующий столбец в исходной таблице (восстановить исходную таблицу по представлению). Поэтому, чтобы представление было обновляемым, оно не должно содержать: from с несколькими таблицами, having, group by, distinct, limit, агрегационные функции, оконные функции, where не может иметь подзапрос)

With check option – при обновлении данных, они будут проверяться на соответствие условию представления. Например:

CREATE VIEW adults\_view AS

SELECT id, name, age

FROM persons

WHERE age >= 18

WITH CHECK OPTION;

Запрос представления будет отображать age>=18, при отсутствии with check option можно вставлять новые значения, где age<18, хоть они и не будут видны в выборке запроса, с этой опцией в таблицу можно вставить только данные удовлетворяющие условию age>=18.

**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.

|  |  |
| --- | --- |
| CREATE FUNCTION name(parametrs)  RETURNS data\_type AS $$  --тело функции  $$ LANGUAGE name\_of\_language  Language можно писать после returns | CREATE FUNCTION name(parametrs)  RETURNS data\_tipe AS $$  BEGIN  RETURN - -используется вместо SELECT  END  $$ LANGUAGE PLPGSQL |
| Аргументы функции могут быть in/out/inout/variadic/default | Можно использовать процедуры – переменные, циклы, if, исключения  Присвоение через = или := |
| Возвращаемые значения:  Returns data\_tipe – возврат через out параметры  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  [Практика](https://www.youtube.com/watch?v=c4nadBeCHSQ&list=PLBheEHDcG7-k1Y_Uy04Dj2ylWhcfSfqoF&index=9&ab_channel=EngineerSpock-IT%26%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) | Используются те же команды для возвращения, если возвращается множество, то return query  Return используется, если нужно явно вернуть значение  DECLARE для объявления новых переменных  Если декларируешь переменные в аргументах, то нужно явно присваивать им значения  Select sum(\*) into  Либо через присвоение аргумент = sum(\*) from table |
|  | **If** выражение **then** действие  **Elsif** выражение **then** действие  **Else** действие  **End if;** |
|  | **While** Выражение = true  **Loop**  Действие  **End loop;**  **Loop**  **Exit when** Выражение = true; - тоже самое что if true then return;  Действие  **End loop;**  **For** счетчик **in[Reverse]** num1**..**num2 **[By** шаг**]**  **Loop**  Действие  **End loop;**  **Continue when** выражение – прерывает текущую итерацию  **Exit / Exit when** – выход из цикла  **Return next** Выражение – используется с множеством, чтобы выводить его промежуточные результаты |

**Exception – RAISE [уровень] ‘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** – Тип хранимой процедуры, вызывающийся при наступлении некоторых событий в БД (изменении данных)

Хранимые процедуры выполняются при их явном вызове, Триггеры же вызываются при попытке изменения данных (до или после, в зависимости от опции AFTER UPDATE/ BEFORE UPDATE)

Единственное его предназначение – вызывать функцию при обнаружении определенных изменений в БД.

Тригеры и функции выполняются в той же транзакции, в которой содержится запрос – не могут начать или закоммитить транзакцию, может только откатить.

На plpgsql создаются с помощью функции без аргументов с возвращаемым значением trigger.

**Оконные функции –** функция, которая работает с выделенным набором строк (партицией). Так же, как и group by необходимы, когда нужно агрегировать данные внутри определенного контекста (Отличие в том, что group by сокращает кол-во строк в результирующей выборке, а функция нет).



В случае использования в запросе одновременно group by и оконной функции сначала отработает group by, а затем к данной выборке будет применена оконная функция.

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

Filter – условие для включения кортежа в оконную функцию.

Если over без указания partition by то вся таблица будет считаться как одна партиция.

Окно и партиция – Партиция представляет собой агрегированные данные по одному или нескольким атрибутам, окно – диапазон строк внутри партиции, которые будут использоваться для вычисления оконной функции для каждой строки.

**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

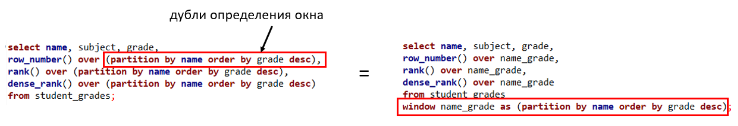
Range – использует значения в качестве разделителя диапазонов, его формирует не одна строка, а совокупность неуникальных значений, сгруппированных в один диапазон. Результатом вычисления для такого диапазона будет агрегированное значение из всех включенных в него кортежей.

[Примеры.](https://habr.com/ru/companies/otus/articles/490296/)

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.

[Пример в конце](https://sql-academy.org/ru/guide/windows-functions-frames)

При использовании order by автоматически выставляется диапазон **range between unbounded preceding and current row**.



**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

[Примеры функций.](https://sql-academy.org/ru/guide/types-of-windows-functions)

**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**