1. **SERIAL**

четырёхбайтное целое с автоувеличением

Типы данных smallserial, serial и bigserial не являются настоящими типами, а представляют собой просто удобное средство для создания колонок с уникальными идентификаторами (подобное свойству AUTO\_INCREMENT в некоторых СУБД).

PostgreSQL имеется особый тип для таких номеров — SERIAL. В основе этого типа лежит тип INTEGER, однако значением по умолчанию для величин этого типа является не NULL, а следующее целое число. Таким образом, если добавлять записи в таблицу, не указывая для полей типа SERIAL, эти значения будут присваиваться автоматически как порядковые целые числа. На значения типа SERIAL накладывается ещё одно ограничение — они не могут быть NULL.

Тип SERIAL в PostgreSQL реализуется с помощью специального механизма — последовательностей. Для каждого поля этого типа создаётся таблица, которая содержит, помимо прочего, текущее число из последовательности. При извлечении очередного номера из последовательности текущее число увеличивается.

1. **Sequence – генератор последовательности**

CREATE SEQUENCE создаёт генератор последовательности. Эта операция включает создание и инициализацию специальной таблицы ***имя***, содержащей одну строку. Владельцем генератора будет пользователь, выполняющий эту команду.

После создания последовательности работать с ней можно, вызывая функции:

nextval- Продвигает последовательность к следующему значению и возвращает его

lastval- Выдаёт значение любой последовательности, которое было возвращено при последнем вызове функции nextval

currval- Выдаёт значение заданной последовательности, которое было возвращено при последнем вызове функции nextval

setval- Устанавливает текущее значение последовательности

1. **Запросы**

WHERE ***условие\_ограничения***

где ***условие\_ограничения*** — любое выражение значения, выдающее результат типа boolean.

Строки порождённой входной таблицы, прошедшие фильтр WHERE, можно сгруппировать с помощью предложения GROUP BY, а затем оставить в результате только нужные группы строк, используя предложение HAVING.

GROUP BY собирает в одну строку все выбранные строки, выдающие одинаковые значения для выражений группировки. В качестве ***выражения*** внутри ***элемента\_группирования*** может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов SELECT), либо произвольное значение, вычисляемое по значениям входных столбцов. В случае неоднозначности имя в GROUP BY будет восприниматься как имя входного, а не выходного столбца.

HAVING исключает из результата строки групп, не удовлетворяющих условию. HAVING отличается от WHERE: WHERE фильтрует отдельные строки до применения GROUP BY, а HAVING фильтрует строки групп, созданных предложением GROUP BY. Каждый столбец, фигурирующий в ***условии***, должен однозначно ссылаться на группируемый столбец, за исключением случаев, когда эта ссылка находится внутри агрегатной функции или негруппируемый столбец функционально зависит от группируемых.

В присутствие HAVING запрос превращается в группируемый, даже если GROUP BY отсутствует.

1. **Коррелированные подзапросы и чем отличаются от обычных**

Результат такого подзапроса зависит от строк, которые извлекаются в основном запросе.

Изображение выглядит как текст, снимок экрана, дисплей, Шрифт

Автоматически созданное описание

1. **Подзапросы**

EXISTS

|  |
| --- |
| SELECT \*  FROM products  WHERE EXISTS (SELECT 1                FROM inventory                WHERE products.product\_id = inventory.product\_id); |

В этом PostgreSQL примере условия EXISTS будут возвращены все записи из таблицы products, где в таблице inventory есть хотя бы одна запись с совпадающим product\_id. Мы использовали SELECT 1 в подзапросе для повышения производительности, поскольку результирующий набор столбцов не имеет отношения к условию EXISTS (имеет значение только наличие возвращенной строки).

|  |
| --- |
| SELECT \*  FROM suppliers  WHERE supplier\_name IN ('Apple', 'Samsung', 'Asus'); |

IN

В этом PostgreSQL примере условия IN будут возвращены все строки из таблицы suppliers, где supplier\_name равно «Apple», «Samsung» или «Asus». Поскольку в SELECT используется \*, то все поля из таблицы suppliers будут отображаться в результирующем наборе.  
Приведенный выше пример IN эквивалентен следующему оператору SELECT:

1. **Рекурсия**

В общем виде рекурсивный запрос WITH всегда записывается как не рекурсивная часть, потом UNION (или UNION ALL), а затем рекурсивная часть, где только в рекурсивной части можно обратиться к результату запроса. Такой запрос выполняется следующим образом:

**Вычисление рекурсивного запроса**

1. Вычисляется не рекурсивная часть. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную рабочую таблицу.
2. Пока рабочая таблица не пуста, повторяются следующие действия:
   1. Вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную промежуточную таблицу.
   2. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

Рекурсивные запросы обычно применяются для работы с иерархическими или древовидными структурами данных.

1. **виды джоинов и их особенности минимальное и максимально количество строк в джоинах**

CROSS JOIN - Соединённую таблицу образуют все возможные сочетания строк из ***T1*** и ***T2*** (т. е. их декартово произведение), а набор её столбцов объединяет в себе столбцы ***T1*** со следующими за ними столбцами ***T2***. Если таблицы содержат N и M строк, соединённая таблица будет содержать N \* M строк.

INNER JOIN - Для каждой строки R1 из T1 в результирующей таблице содержится строка для каждой строки в T2, удовлетворяющей условию соединения с R1.

LEFT OUTER JOIN - Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. Таким образом, в результирующей таблице всегда будет минимум одна строка для каждой строки из T1.

RIGHT OUTER JOIN - Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL. Это соединение является обратным к левому (LEFT JOIN): в результирующей таблице всегда будет минимум одна строка для каждой строки из T2.

FULL OUTER JOIN - Сначала выполняется внутреннее соединение. Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. И наконец, в результат включаются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL.

1. **Представление**

CREATE VIEW создаёт представление запроса. Создаваемое представление лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению.

Команда CREATE OR REPLACE VIEW действует подобным образом, но если представление с этим именем уже существует, оно заменяется. Новый запрос должен выдавать те же столбцы, что выдавал запрос, ранее определённый для этого представления (то есть, столбцы с такими же именами должны иметь те же типы данных и следовать в том же порядке), но может добавить несколько новых столбцов в конце списка. Вычисления, в результате которых формируются столбцы представления, могут быть совершенно другими.

1. **материализованные представления и отличие от обычного представления**

CREATE MATERIALIZED VIEW определяет материализованное представление запроса. Заданный запрос выполняется и наполняет представление в момент вызова команды (если только не указано WITH NO DATA). Обновить представление позже можно, выполнив REFRESH MATERIALIZED VIEW.

Команда CREATE MATERIALIZED VIEW подобна CREATE TABLE AS, за исключением того, что она запоминает запрос, порождающий представление, так что это представление можно позже обновить по требованию. Материализованные представления сходны с таблицами во многом, но не во всём; например, не поддерживаются временные материализованные представления и автоматическая генерация OID.

Материализованные представления (materialized views) в PostgreSQL – это представления, которые фактически хранят результаты запросов, как если бы это были реальные таблицы. Это означает, что при обращении к материализованному представлению не требуется повторно выполнять исходный запрос; вместо этого PostgreSQL может просто вернуть сохранённые результаты.

Одно из основных отличий заключается в том, что материализованные представления не обновляются автоматически при изменении исходных данных. Для обновления данных в материализованном представлении требуется явно выполнить операцию REFRESH MATERIALIZED VIEW.

Таблицы в PostgreSQL, с другой стороны, отражают текущее состояние хранимых данных в реальном времени.

Кроме того, таблицы обеспечивают больше возможностей для операций обновления и вставки

Материализованные представления обычно предпочтительнее, когда запросы к базе данных занимают много времени и ресурсов, и нет необходимости в наиболее актуальных данных.

Обычные представления обычно лучше подходят, когда нужны самые актуальные данные, и нет ограничений по производительности или времени выполнения запросов.

1. **агрегатные функции**

Агрегатные функции получают единственный результат из набора входных значений.

Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее.

Большинство агрегатных функций игнорируют значения NULL, так что строки, для которых выражения выдают одно или несколько значений NULL, отбрасываются

Агрегатное выражение может фигурировать только в списке результатов или в предложении HAVING команды SELECT. Во всех остальных предложениях, например WHERE, они запрещены, так как эти предложения логически вычисляются до того, как формируются результаты агрегатных функций.

AVG: находит среднее значение. Входной параметр должен представлять один из следующих типов: smallint, int, bigint, real, double precision, numeric, interval. Для целочисленнных параметров результатом будет значение типа numeric, для параметров, которые представляют число с плавающей точкой, - значение типа double precision.

BIT\_AND: выполняет операцию побитового умножения (операции логического И) для чисел следующих типов: smallint, int, bigint, bit. Если параметр содержит значение NULL, то возвращается NULL.

BIT\_OR: выполняет операцию побитового сложения (операции логического ИЛИ) для чисел следующих типов: smallint, int, bigint, bit. Если параметр содержит значение NULL, то возвращается NULL.

BOOL\_AND: выполняет операцию логического умножения для значений типа bool. Если входные все значения равны true, то возвращается true, иначе возвращается false.

BOOL\_OR: выполняет операцию логического сложения для значений типа bool. Если входные хотя бы одно из значений равно true, то возвращается true, иначе возвращается false.

COUNT(\*): находит количество строк в запросе

COUNT(expression): находит количество строк в запросе, для которых expression не содержит значение NULL.

SUM: находит сумму значений

MIN: находит наименьшее значение

MAX: находит наибольшее значение

STRING\_AGG(expression, delimiter): соединяет с помощью delimiter все текстовые значения из expression в одну строку.