Памятка PostgreSQL

# Арифметические операции:

1. Сложение (+): Этот оператор складывает два числа. Например: SELECT 3 + 5; вернёт 8.

2. Вычитание (-): Этот оператор вычитает одно число из другого. Например: SELECT 8 - 5; вернёт 3.  
  
3. Умножение (\*): Этот оператор умножает два числа. Например: SELECT 3 \* 5; вернёт 15.  
  
4. Деление (/): Этот оператор делит одно число на другое. Например: SELECT 10 / 2; вернёт 5.  
  
5. Модуль (%): Этот оператор возвращает остаток от деления одного числа на другое. Например: SELECT 10 % 3; вернёт 1.  
  
6. Квадратный корень (|/): Этот оператор возвращает квадратный корень из числа. Например: SELECT |/ 9; вернёт 3.  
  
7. Кубический корень (||/): Этот оператор возвращает кубический корень из числа. Например: SELECT ||/ 8; вернёт 2.  
  
8. Возведение в степень (^): Этот оператор возводит одно число в степень другого. Например: SELECT 2 ^ 3; вернёт 8.  
  
9. Факториал (!): Этот оператор возвращает факториал числа. Например: SELECT 5!; вернёт 120.  
  
Если применять эти операции к столбцам из таблиц, вместо чисел можно использовать имена столбцов. Например: SELECT column1 + column2 FROM table1; эта команда применит операцию сложения к каждой строке в таблице, складывая значения из column1 и column2.10:19

# Работа со строками:

1. CONCAT(): Эта функция объединяет две или более строк в одну. Например: SELECT CONCAT('Hello', ' World'); вернёт 'Hello World'.  
  
2. LENGTH(): Эта функция возвращает количество символов в строке. Например: SELECT LENGTH('Hello'); вернёт 5. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
3. SUBSTRING(): Эта функция извлекает подстроку из строки. Можно указать начальное положение и длину подстроки. Например: SELECT SUBSTRING('Hello World' FROM 1 FOR 5); вернёт 'Hello'. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
4. POSITION(): Эта функция находит позицию подстроки в строке, возвращая первое вхождение. Например: SELECT POSITION('W' IN 'Hello World'); вернёт 7.  
  
5. TRIM(): Эта функция удаляет пробелы с начала и конца строки. Например: SELECT TRIM(' Hello '); вернёт 'Hello'.  
  
6. LOWER(): Эта функция преобразует все символы строки в нижний регистр. Например: SELECT LOWER('Hello WORLD'); вернёт 'hello world'. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
7. UPPER(): Эта функция преобразует все символы строки в верхний регистр. Например: SELECT UPPER('Hello World'); вернёт 'HELLO WORLD'. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
8. INITCAP(): Эта функция преобразует первую букву каждого слова строки в верхний регистр, а остальные буквы - в нижний регистр. Например: SELECT INITCAP('hello WORLD'); вернёт 'Hello World'. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
9. REPLACE(): Эта функция заменяет все вхождения подстроки в строке на другую подстроку. Например: SELECT REPLACE('Hello World', 'World', 'PostgreSQL'); вернёт 'Hello PostgreSQL'.  
  
10. SPLIT\_PART(): Эта функция разбивает строку по указанному разделителю и возвращает указанную часть. Например: SELECT SPLIT\_PART('Hello-World', '-', 2); вернёт 'World'. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
Все эти функции работают со строковыми данными в PostgreSQL. Чтобы применить их к столбцам в таблице, вместо строковых констант можно использовать имена столбцов.

# Р**абот**а **с датами:**

1. DATE\_PART(): Эта функция извлекает определенную часть из даты или времени. Например, SELECT DATE\_PART('day', TIMESTAMP '2001-02-16 20:38:40'); вернёт 16.  
  
2. NOW(): Эта функция возвращает текущую дату и время. Например, SELECT NOW(); вернёт текущую дату и время.  
  
3. CURRENT\_DATE: Это ключевое слово возвращает текущую дату.  
  
4. CURRENT\_TIME: Это ключевое слово возвращает текущее время.  
  
5. AGE(): Эта функция возвращает разницу между двуми датами. Например, SELECT AGE(TIMESTAMP '2001-04-10', TIMESTAMP '1957-06-13'); вернёт интервал в 43 года.  
  
6. EXTRACT(): Эта функция, аналогичная DATE\_PART(), извлекает определенную составляющую даты или времени. **(МОДУЛЬ 3. ОСНОВЫ SQL)**  
  
7. DATE\_TRUNC(): Эта функция обрезает дату или время по указанной части. Например, SELECT DATE\_TRUNC('year', TIMESTAMP '2001-02-16 20:38:40'); вернёт '2001-01-01 00:00:00'.  
  
8. INTERVAL: Это ключевое слово используется для создания интервала времени, который можно добавить или вычесть из даты или времени.

9. Функция DATE используется для создания значения даты на основе указанных аргументов. Она позволяет создавать даты без временной составляющей или устанавливать только часть даты (например, год, месяц и день).

Примеры использования функции DATE:

SELECT DATE(2022, 12, 31) AS дата;

SELECT DATE(2022, 12, EXTRACT(DAY FROM CURRENT\_DATE)) AS дата;

SELECT DATE\_PART('year', CURRENT\_DATE)::integer AS текущий\_год;

Все эти функции могут использоваться для работы с датами, временем и интервалами в PostgreSQL. Для применения функций к столбцам таблицы вместо строковых констант можно использовать имена столбцов.

# Л**огические операторы:**

1. AND: Логический оператор "И". Возвращает TRUE, если все условия равны TRUE. Например: SELECT TRUE AND TRUE; вернёт TRUE.  
  
2. OR: Логический оператор "ИЛИ". Возвращает TRUE, если хотя бы одно из условий равно TRUE. Например: SELECT TRUE OR FALSE; вернёт TRUE.  
  
3. NOT: Логический оператор "НЕ". Инвертирует значение условия. Например: SELECT NOT TRUE; вернёт FALSE.  
  
4. =: Оператор равенства. Возвращает TRUE, если два значения равны. Например: SELECT 2 = 2; вернёт TRUE.  
  
5. <> или !=: Оператор неравенства. Возвращает TRUE, если два значения не равны. Например: SELECT 2 <> 3; вернёт TRUE.  
  
6. >: Оператор больше. Возвращает TRUE, если первое значение больше второго. Например: SELECT 3 > 2; вернёт TRUE.  
  
7. <: Оператор меньше. Возвращает TRUE, если первое значение меньше второго. Например: SELECT 2 < 3; вернёт TRUE.  
  
8. >=: Оператор больше или равно. Возвращает TRUE, если первое значение больше или равно второму. Например: SELECT 3 >= 3; вернёт TRUE.  
  
9. <=: Оператор меньше или равно. Возвращает TRUE, если первое значение меньше или равно второму. Например: SELECT 3 <= 3; вернёт TRUE.  
  
10. IS NULL: Проверяет, является ли значение NULL. Например: SELECT NULL IS NULL; вернёт TRUE.  
  
11. IS NOT NULL: Проверяет, не является ли значение NULL. Например: SELECT 1 IS NOT NULL; вернёт TRUE. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
12. BETWEEN: Проверяет, находится ли значение в указанном диапазоне. Например: SELECT 5 BETWEEN 1 AND 10; вернёт TRUE. **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**  
  
13. IN: Проверяет, содержится ли значение в указанном списке значений. Например: SELECT 3 IN (1, 2, 3, 4, 5); вернёт TRUE.  
  
Все эти логические операторы и функции можно использовать для создания условий в запросах SQL в PostgreSQL.

# А**грегатны**е **функций:**

1. AVG(): Эта функция возвращает среднее значение числового столбца. Например: SELECT AVG(price) FROM products; вернёт среднюю цену товаров.  
  
2. SUM(): Эта функция возвращает сумму числового столбца. Например: SELECT SUM(price) FROM products; вернёт общую стоимость всех товаров.  
  
3. MAX(): Эта функция возвращает максимальное значение столбца. Например: SELECT MAX(price) FROM products; вернёт наибольшую цену.  
  
4. MIN(): Эта функция возвращает минимальное значение столбца. Например: SELECT MIN(price) FROM products; вернёт наименьшую цену.  
  
5. COUNT(): Эта функция возвращает количество строк, соответствующих указанному критерию. Например: SELECT COUNT(\*) FROM products; вернёт количество строк в таблице products. **(МОДУЛЬ 3. ОСНОВЫ SQL)**  
6. ARRAY\_AGG(): Эта функция объединяет значения в массив. Например: SELECT ARRAY\_AGG(price) FROM products; вернёт массив со всеми ценами товаров.  
  
7. STRING\_AGG(): Эта функция объединяет значения (через определенный разделитель) в одну строку. Например: SELECT STRING\_AGG(name, ', ') FROM products; вернёт одну строку с именами всех товаров, разделенных запятой.  
  
8. GROUP\_CONCAT(): Эта функция объединяет значения в одну строку с разделителем. В PostgreSQL вместо нее применяется STRING\_AGG().  
  
9. FIRST() и LAST(): Эти функции возвращают первое и последнее значения в группе. Однако, эти функции не встроены в PostgreSQL. Чтобы их использовать, нужно использовать модульдополнения перового и последнего значения.  
  
Все эти функции можно использовать в запросах SQL в PostgreSQL для обработки групп строк и столбцов.

# О**диночны**е **функций:**

1. ABS(): Эта функция возвращает абсолютное значение числа. Например, SELECT ABS(-5); вернёт 5.  
  
2. ROUND(): Эта функция округляет число до указанного количества десятичных знаков. Например, SELECT ROUND(42.789, 2); вернёт 42.79. **(МОДУЛЬ 3. ОСНОВЫ SQL)**  
  
3. CEIL() или CEILING(): Эта функция округляет число вверх до ближайшего целого. Например, SELECT CEIL(42.1); вернёт 43.  
  
4. FLOOR(): Эта функция округляет число вниз до ближайшего целого. Например, SELECT FLOOR(42.9); вернёт 42.  
  
5. SQRT(): Эта функция возвращает квадратный корень числа. Например, SELECT SQRT(49); вернёт 7.  
  
6. CBRT(): Эта функция возвращает кубический корень числа. Например, SELECT CBRT(27); вернёт 3.  
  
7. EXP(): Эта функция возвращает экспоненту числа. Например, SELECT EXP(2); вернёт примерно 7.39.  
  
8. LN(): Эта функция возвращает натуральный логарифм числа. Например, SELECT LN(7.389); вернёт примерно 2.

# Оконные функции:

1. ROW\_NUMBER(): Эта функция возвращает номер строки в окне. Например: SELECT name, ROW\_NUMBER() OVER (ORDER BY name ASC) as rownum FROM tablename; вернёт номер каждой строки в порядке сортировки по имени.  
  
2. RANK(): Эта функция возвращает ранг строки в окне, с учетом равных значений. Она будет пропускать следующий ранг в случае равенства, поэтому два равных значения будут иметь одинаковый ранг.  
  
3. DENSE\_RANK(): Эта функция, как и RANK(), возвращает ранг строки в окне. Однако DENSE\_RANK() не пропускает следующий ранг в случае равенства, поэтому два равных значения будут иметь одинаковый ранг, и следующая строка получит следующий номер ранга.  
  
4. LEAD(): Эта функция возвращает значение следующей строки в окне. Например, SELECT name, LEAD(name) OVER (ORDER BY name ASC) as nextname FROM tablename; вернёт значение имени следующей строки для каждой строки в таблице.  
  
5. LAG(): Эта функция возвращает значение предыдущей строки в окне. Это противоположность функции LEAD().  
  
6. FIRST\_VALUE(): Эта функция возвращает значение первой строки в окне. Например, SELECT name, FIRST\_VALUE(name) OVER (ORDER BY name ASC) as firstname FROM tablename; вернёт значение первой строки в окне.  
  
7. LAST\_VALUE(): Эта функция возвращает значение последней строки в окне.

# Функций сортировки:

1. ORDER BY: Это самая базовая функция сортировки в SQL. Она позволяет упорядочить результаты запроса по одной или нескольким колонкам таблицы. Например:  
```  
SELECT \* FROM таблица ORDER BY колонка;  
```  
  
2. GROUP BY: Эта функция сортирует результаты запроса по значениям одной или нескольких колонк, и затем группирует строки с одинаковыми значениями в этих колонках. Например:  
```  
SELECT колонка1, колонка2 FROM таблица GROUP BY колонка1, колонка2;  
```  
  
3. DISTINCT: Эта функция сортирует результаты запроса по значениям одной или нескольких колонок и удаляет повторяющиеся строки. Например:  
```  
SELECT DISTINCT колонка FROM таблица; **(МОДУЛЬ 3. ОСНОВЫ SQL)**  
```  
  
4. UNION: Эта функция объединяет результаты запросов нескольких SELECT операторов и сортирует их в определенном порядке. Например:  
```  
SELECT колонка FROM таблица1 UNION SELECT колонка FROM таблица2 ORDER BY колонка;  
```  
  
5. TOP или LIMIT: Эты функции ограничивают количество строк, возвращаемых запросом, и могут использоваться для сортировки. Например:  
```  
SELECT TOP N колонка FROM таблица ORDER BY колонка;  
SELECT колонка FROM таблица ORDER BY колонка LIMIT N;  
```

# **Операторы объединения**:

1. INNER JOIN: INNER JOIN, или просто JOIN, возвращает только те строки, для которых есть совпадения в обоих таблицах, основываясь на условии связи. Например:

SELECT \* FROM таблица1 INNER JOIN таблица2 ON таблица1.колонка = таблица2.колонка;

2. LEFT JOIN (или LEFT OUTER JOIN): LEFT JOIN возвращает все строки из левой (первой) таблицы и соответствующие совпадения из правой (второй) таблицы. Если в правой таблице нет совпадений, то возвращается NULL. Например:

SELECT \* FROM таблица1 LEFT JOIN таблица2 ON таблица1.колонка = таблица2.колонка;

3. RIGHT JOIN (или RIGHT OUTER JOIN): RIGHT JOIN возвращает все строки из правой (второй) таблицы и соответствующие совпадения из левой (первой) таблицы. Если в левой таблице нет совпадений, то возвращается NULL. Например:

SELECT \* FROM таблица1 RIGHT JOIN таблица2 ON таблица1.колонка = таблица2.колонка;

4. FULL JOIN (или FULL OUTER JOIN): FULL JOIN возвращает все строки из обеих таблиц и соответствующие совпадения. Если в одной из таблиц нет совпадений, то возвращается NULL. Например: **(МОДУЛЬ 6. POSTGRESQL)**

SELECT \* FROM таблица1 FULL JOIN таблица2 ON таблица1.колонка = таблица2.колонка;

5. CROSS JOIN: CROSS JOIN производит декартово произведение между двумя таблицами. Он возвращает все возможные комбинации строк из обеих таблиц, без условия связи. Например:

SELECT \* FROM таблица1 CROSS JOIN таблица2;

Это основные операторы объединения, которые используются в PostgreSQL. Каждый из них предоставляет различные способы комбинировать данные из разных таблиц в запросах SQL.

# Функции фильтрации:

1. WHERE: Оператор WHERE позволяет фильтровать строки таблицы на основе заданного условия. Например:

SELECT \* FROM таблица WHERE условие;

2. LIKE: Оператор LIKE используется для поиска строк, соответствующих заданному шаблону. Он часто используется для фильтрации текстовых данных. Например:

SELECT \* FROM таблица WHERE колонка LIKE 'шаблон';

3. EXISTS: Функция EXISTS используется для проверки существования результатов подзапроса. Она фильтрует строки на основе результата подзапроса. Например:

SELECT \* FROM таблица WHERE EXISTS (подзапрос);

4. Условный оператор CASE в SQL используется для выполнения различных действий в зависимости от значения определенного выражения или условия..

Пример использования конструкции CASE с оператором ELSE:

SELEC колонка, CASE WHEN условие1 THEN выражение1 WHEN условие2 THEN выражение ELSE выражение\_по\_умолчани END AS новая\_кол **(МОДУЛЬ 3. ОСНОВЫ SQL)**

5. Оператор HAVING используется совместно с оператором GROUP BY и позволяет фильтровать группы данных на основе условий агрегатных функций. Он используется для ограничения результирующего набора данных, полученного после группировки данных, в соответствии с определенными критериями.

Пример использования оператора HAVING:

SELECT колонка1, агрегатная\_функция(колонка2) FROM таблица

GROUP BY колонка1

HAVING агрегатная\_функция(колонка2) > значение; **(МОДУЛЬ 3. ОСНОВЫ SQL)**

6. Функция ANY принимает набор значений и возвращает булево значение (true или false) в зависимости от выполнения условия. В случае, если хотя бы одно значение удовлетворяет условию, функция ANY возвращает true; иначе возвращается false.

Пример использования функции ANY:

SELECT \* FROM products WHERE price > ANY (SELECT price FROM competitors);

Этот запрос выбирает все продукты, чья цена выше цен, предоставленных конкурентами в подзапросе. Если хотя бы один элемент из подзапроса удовлетворяет условию (т.е. цена продукта выше), то для этого продукта вернется значение true и он будет выбран в результирующем наборе.

1. Оператор AND используется для комбинирования условий в операторе WHERE. Результатом запроса будет только те строки, которые удовлетворяют обоим условиям. Например, в запросе:

SELECT \* FROM employees

WHERE age > 30 AND department = 'Sales';

Результатом будет только те строки, где возраст сотрудника больше 30 и отдел равен 'Sales'.

2. Оператор OR также используется для комбинирования условий в операторе WHERE, но в отличие от оператора AND, результатом запроса будет любая строка, которая удовлетворяет хотя бы одному из условий. Например, в запросе:

SELECT \* FROM employees

WHERE age > 30 OR department = 'Sales';

Результатом будет любые строки, где возраст сотрудника больше 30 или отдел равен 'Sales'.

3. Оператор NOT используется для инвертирования условия в операторе WHERE. Результатом запроса будет только те строки, которые не удовлетворяют условию. Например, в запросе:

SELECT \* FROM employees

WHERE NOT age > 30;

Результатом будет только те строки, где возраст сотрудника не больше 30.

4. Оператор IN используется для проверки значения в списке значений. Результатом запроса будет только те строки, которые имеют значение, присутствующее в списке значений. Например, в запросе:

SELECT \* FROM employees

WHERE department IN ('Sales', 'Marketing', 'HR');

Результатом будет только те строки, где отдел равен 'Sales', 'Marketing' или 'HR'.

5. Оператор BETWEEN используется для проверки значения находится в заданном диапазоне. Результатом запроса будет только те строки, которые имеют значение, находящееся между заданными значениями. Например, в запросе:

SELECT \* FROM employees

WHERE age BETWEEN 25 AND 35;

Результатом будет только те строки, где возраст сотрудника находится в диапазоне от 25 до 35.

6. Оператор LIKE используется для сравнения строк с использованием шаблонов. Результатом запроса будет только те строки, которые удовлетворяют заданному шаблону. Например, в запросе:

SELECT \* FROM employees

WHERE first\_name LIKE 'J%';

Результатом будет только те строки, где имя сотрудника начинается с буквы 'J'.

7. Оператор IS NULL используется для проверки на отсутствие значения. Результатом запроса будет только те строки, которые не имеют значения в указанном столбце. Например, в запросе:

SELECT \* FROM employees

WHERE address IS NULL;

Результатом будет только те строки, где значение в столбце address равно NULL.

8. Оператор EXISTS используется для проверки наличия хотя бы одной строки в подзапросе. Результатом запроса будет только те строки, для которых подзапрос возвращает хотя бы одну строку. Например, в запросе:

SELECT \* FROM employees

WHERE EXISTS (SELECT 1 FROM orders WHERE orders.employee\_id = employees.id);

Результатом будет только те строки, для которых существует хотя бы один заказ, связанный с данным сотрудником.

9. Функция ANY используется в сочетании с операторами сравнения для сравнения значения с набором значений из подзапроса. Результатом запроса будет только те строки, для которых значение больше хотя бы одного значения из подзапроса. Например, в запросе:

SELECT \* FROM products

WHERE price > ANY (SELECT price FROM products WHERE category = 'Electronics');

Результатом будет только те строки, где цена продукта больше цены хотя бы одного продукта из категории 'Electronics'.

10. Функция SOME используется аналогично функции ANY для сравнения значения с набором значений из подзапроса. Результатом запроса будет только те строки, для которых значение больше хотя бы одного значения из подзапроса. Например, в запросе:

SELECT \* FROM products

WHERE price > SOME (SELECT price FROM products WHERE category = 'Electronics');

Результатом будет только те строки, где цена продукта больше цены хотя бы одного продукта из категории 'Electronics'.

11. Функция ALL - используется для сравнения значения со всеми значениями, возвращаемыми подзапросом. В данном случае запрос выбирает все продукты, у которых цена больше всех цен продуктов из категории "Electronics".

Запрос:

SELECT \* FROM products

WHERE price > ALL (SELECT price FROM products WHERE category = 'Electronics');

Результат запроса:

- Продукт A: цена 10

- Продукт B: цена 15

- Продукт C: цена 12

12. Оператор XOR - возвращает истину, если только одно из условий истинно. В данном случае запрос выбирает всех сотрудников, возраст которых больше 30 лет или которые работают в отделе "Sales", но не одновременно.

Запрос:

SELECT \* FROM employees

WHERE age > 30 XOR department = 'Sales';

Результат запроса:

- Сотрудник A: возраст 35, отдел "Sales"

- Сотрудник B: возраст 25, отдел "Sales"

- Сотрудник C: возраст 40, отдел "Marketing"

13. Оператор && - используется для проверки пересечения элементов массива и набора значений. В данном случае запрос выбирает все продукты, у которых теги содержат значения "sale" и "new".

Запрос:

SELECT \* FROM products

WHERE tags && ARRAY['sale', 'new'];

Результат запроса:

- Продукт A: теги ["sale", "new", "discount"]

- Продукт B: теги ["new"]

14. Оператор @> - выполняет проверку, содержит ли JSON-значение все элементы, указанные в JSON-объекте. В данном случае запрос выбирает всех сотрудников, у которых детали должности и зарплаты соответствуют указанным значениям.

Запрос:

SELECT \* FROM employees

WHERE details @> '{"position": "Manager", "salary": 5000}';

Результат запроса:

- Сотрудник A: детали {"position": "Manager", "salary": 5000}

- Сотрудник B: детали {"position": "Analyst", "salary": 3000}

- Сотрудник C: детали {"position": "Manager", "salary": 5000}

15. Оператор <@ - выполняет проверку, содержит ли JSON-значение все элементы, указанные в JSON-массиве. В данном случае запрос выбирает все продукты, у которых теги содержат все значения из массива ["sale", "new", "discount"].

Запрос:

SELECT \* FROM products

WHERE tags <@ ARRAY['sale', 'new', 'discount'];

Результат запроса:

- Продукт A: теги ["sale", "new", "discount"]

16. Оператор GROUP BY - группирует строки по указанному столбцу и применяет агрегатные функции к каждой группе. В данном случае запрос группирует сотрудников по отделам и вычисляет среднюю зарплату в каждом отделе.

Запрос:

SELECT department, AVG(salary) FROM employees

GROUP BY department;

Результат запроса:

- Отдел "Sales": средняя зарплата 4500

- Отдел "Marketing": средняя зарплата 3500

17. Оператор HAVING - используется для фильтрации групп, сформированных оператором GROUP BY. В данном случае запрос выбирает отделы, у которых средняя зарплата больше 5000. **(МОДУЛЬ 3. ОСНОВЫ SQL)**

Запрос:

SELECT department, AVG(salary) FROM employees

GROUP BY department

HAVING AVG(salary) > 5000;

Результат запроса:

- Отдел "Sales": средняя зарплата 6000

- Отдел "Finance": средняя зарплата 5500

18. Оператор CASE - позволяет выполнить различные действия в зависимости от значения столбца или выражения. В данном случае запрос выбирает все продукты и добавляет столбец "Discount" со значением, рассчитанным на основе цены продукта. **(МОДУЛЬ 3. ОСНОВЫ SQL)**

Запрос:

SELECT \*,

CASE

WHEN price > 50 THEN price \* 0.9

WHEN price > 20 THEN price \* 0.95

ELSE price

END AS discount

FROM products;

Результат запроса:

- Продукт A: цена 100, скидка 90

- Продукт B: цена 30, скидка 28.5

- Продукт C: цена 10, без скидки

20. Оператор LIMIT - ограничивает количество возвращаемых строк результата запроса.

Результат запроса:

SELECT \* FROM employees

LIMIT 10;

# Создание объектов:

1. CREATE DATABASE: Используется для создания новой базы данных.

2. CREATE TABLE: Эта функция используется для создания новой таблицы в базе данных PostgreSQL. Она определяет структуру таблицы, задает имена столбцов и их типы данных. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

3. CREATE INDEX: Используется для создания индекса на одном или нескольких столбцах таблицы. Индекс позволяет ускорить выполнение запросов путем быстрого доступа к данным в таблице.

4. CREATE VIEW: Эта функция используется для создания виртуальной таблицы, которая представляет собой результат выполнения запроса. View не хранит данные физически, он предоставляет доступ к данным других таблиц или представляет их в нужном формате.

5. CREATE FUNCTION: Используется для создания пользовательской функции в PostgreSQL. Пользовательская функция выполняет определенные операции или вычисления и может быть вызвана из SQL-запросов.

6. CREATE PROCEDURE: Позволяет создать хранимую процедуру, которая представляет собой набор инструкций SQL, объединенных в одну единицу, чтобы выполнить определенную задачу.

7. CREATE TRIGGER: Используется для создания триггера, который запускается автоматически при выполнении определенных операций (например, INSERT, UPDATE или DELETE) над таблицей. Триггеры позволяют выполнять дополнительные действия или проверки при изменении данных.

8. CREATE DOMAIN: Эта функция позволяет создавать новый пользовательский тип данных, который может использоваться для определения столбцов таблицы или аргументов функции.

Все эти функции позволяют создавать и определять различные объекты или структуры данных в PostgreSQL, в зависимости от потребностей пользователя или приложения.

9. Функция VARCHAR не относится к классу функций, так как VARCHAR является типом данных, используемым для хранения переменной длины символов в базе данных PostgreSQL. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

10. INTEGER является целочисленным типом данных, который используется для хранения целых чисел без десятичной части. Он может представлять положительные, отрицательные и нулевые значения. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

# Изменения и удаления:

1) ALTER TABLE: функция используется для изменения структуры таблицы. Она позволяет добавлять или удалять столбцы, изменять типы данных столбцов, устанавливать ограничения и другие операции.

Пример запроса:

ALTER TABLE employees

ADD COLUMN hire\_date DATE;

Результат: добавляется новый столбец "hire\_date" с типом данных DATE в таблицу "employees".

2) DROP TABLE: функция используется для удаления таблицы из базы данных. Она полностью удаляет таблицу и все связанные с ней данные.

Пример запроса:

DROP TABLE employees; **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

Результат: таблица "employees" с ее структурой и данными будет удалена из базы данных.

Это некоторые из функций, используемых в PostgreSQL для определения, изменения и удаления структур данных. Все они позволяют администраторам и разработчикам эффективно управлять структурой и объектами базы данных.

# Ограничения:

1. PRIMARY KEY: Ограничение PRIMARY KEY используется для определения первичного ключа в таблице. Он гарантирует уникальность и непустоту значений столбца (или комбинации столбцов), на которые оно накладывается. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

2. UNIQUE: Ограничение UNIQUE используется для задания уникального значения на одном или нескольких столбцах таблицы. Оно гарантирует, что каждая запись в таблице имеет уникальное значение в указанных столбцах. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

3. FOREIGN KEY: Ограничение FOREIGN KEY используется для определения ссылочной целостности между таблицами. Оно связывает столбец (или столбцы) в одной таблице с первичным ключом столбца (или столбцов) в другой таблице.

4. CHECK: Ограничение CHECK используется для задания условия или выражения, которое должно быть выполнено для всех записей в таблице. Оно позволяет определить пользовательское правило, которому должны соответствовать данные в таблице. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

5. NOT NULL: Ограничение NOT NULL используется для указания, что столбец не может содержать пустые значения (NULL). Оно гарантирует, что в указанном столбце должны быть всегда указаны значения. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

Эти функции позволяют определять ограничения целостности данных в базе данных PostgreSQL, чтобы обеспечить правильное и надежное хранение данных. Ограничения целостности устанавливают правила для значений в таблицах, гарантируя их соответствие определенным требованиям и предотвращая ошибки и некорректные данные.

# Вставка данных:

1. INSERT: Функция INSERT используется для вставки новых строк данных в таблицу. Она позволяет указать значения для каждого столбца вставляемой строки или использовать значения по умолчанию для некоторых или всех столбцов. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

2. VALUES: Функция VALUES используется для вставки одной или нескольких строк данных в таблицу или для создания набора значений в виде таблицы-значений. Она позволяет указать конкретные значения столбцов для вставляемых строк или создать набор значений для дальнейшего использования. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

3. SELECT INTO: Функция SELECT INTO используется для создания новой таблицы на основе результатов выборки данных из одной или нескольких существующих таблиц. Она позволяет выбрать определенные столбцы и строки данных и вставить их в новую таблицу.

4. COPY: Функция COPY используется для быстрой загрузки или выгрузки данных в или из таблицы. Она предоставляет механизм для копирования данных между файлами и таблицами, что может быть очень полезно для массовой вставки данных.

5. RETURNING: Ключевое слово RETURNING можно использовать с операцией INSERT, UPDATE или DELETE, чтобы вернуть результаты этих операций или получить набор значений, который был изменен или удален.

Эти функции позволяют вставлять данные в таблицы или получать наборы значений из базы данных в PostgreSQL. Они предоставляют различные способы и возможности для работы с данными и обеспечивают гибкость в управлении вставками и извлечениями в базе данных.

6. Функция INTO является частью синтаксической структуры SELECT INTO и относится к классу функций, связанных с созданием новых таблиц на основе результатов выборки данных из существующих таблиц в PostgreSQL.

Важно отметить, что операция SELECT INTO может быть использована только для создания новой таблицы. Если требуется вставить данные в существующую таблицу, следует использовать операцию INSERT или другие специфические функции для вставки данных.

# Внешний ключ:

1. REFERENCES: Определяет внешний ключ для связи с другой таблицей. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

2. ON DELETE: Определяет действие, которое происходит при удалении записи из связанной таблицы.

3. ON UPDATE: Определяет действие, которое происходит при обновлении значения в связанной таблице.

4. CASCADE: Определяет, что изменения, производимые в связанной таблице, должны отразиться на таблице с внешним ключом.

5. SET NULL: Устанавливает значение внешнего ключа в NULL при удалении записи из связанной таблицы.

6. SET DEFAULT: Устанавливает значение внешнего ключа в значение по умолчанию при удалении записи из связанной таблицы.

7. NO ACTION: Ничего не делает при удалении или обновлении записи в связанной таблице.

Это некоторые из основных функций, связанных с ключами внешнего ключа в SQL-запросах.

# Добавление данных:

1. DEFAULT: Определяет значение по умолчанию для столбца при его создании или изменении. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

2. current\_timestamp: Устанавливает текущую дату и время как значение по умолчанию.

3. nextval('sequence\_name'): Возвращает следующее значение из указанной последовательности и используется для установки автоинкрементного значения при добавлении записи.

4. NULL: Устанавливает значение NULL как значение по умолчанию для столбца.

Это некоторые из функций, используемых в PostgreSQL для установки значений по умолчанию при добавлении или обновлении данных в базе данных.

# Набор-значений:

1. UNNEST: Данная функция раскрывает многомерный массив или набор значений и возвращает его элементы в виде набора-значений. Например:

**(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

SELECT UNNEST(ARRAY[1, 2, 3]);

Результат: 1

2

3

2. GENERATE\_SERIES: Эта функция генерирует серию значений в указанном диапазоне и возвращает их в виде набора-значений. Например:

SELECT GENERATE\_SERIES(1, 5);

Результат: 1

2

3

4

5

3. jsonb\_each и jsonb\_each\_text: Эти функции позволяют разложить JSON-объект на пары ключ-значение и возвращают их в виде набора-значений. Разница между ними заключается в том, что jsonb\_each возвращает ключи и значения в JSON-формате, а jsonb\_each\_text в простом текстовом формате. Например:

SELECT \* FROM jsonb\_each('{"key1": "value1", "key2": "value2"}');

Результат:

key1 | "value1"

key2 | "value2"

4. unnest или setof: Эти функции позволяют возвращать результаты пользовательской функции в виде набора-значений. Функция может быть определена как множество значений или использовать ключевое слово unnest для раскрытия значения. Например:

CREATE FUNCTION get\_names() RETURNS SETOF text AS $$

BEGIN

RETURN QUERY SELECT name FROM users;

END;

$$

SELECT \* FROM get\_names();

Результат: Name1

Name2

Name3

# Массивы данных:

1) array\_length: функция возвращает размерность массива в указанном измерении.

Пример запроса:

SELECT array\_length('{1, 2, 3}', 1);

Результат:

3

2) array\_append: функция добавляет элемент в конец массива.

Пример запроса:

SELECT array\_append('{1, 2, 3}', 4);

Результат:

{1, 2, 3, 4}

3) array\_prepend: функция добавляет элемент в начало массива.

Пример запроса:

SELECT array\_prepend(0, '{1, 2, 3}');

Результат:

{0, 1, 2, 3}

4) array\_cat: функция объединяет несколько массивов в один.

Пример запроса:

SELECT array\_cat('{1, 2}', '{3, 4}');

Результат:

{1, 2, 3, 4}

5) unnest: функция преобразует массив в набор строк, развернув элементы массива в отдельные строки.

Пример запроса:

SELECT unnest('{1, 2, 3}');

Результат: 1 2 3

6) array\_agg: функция объединяет множество строк в один массив.

Пример запроса:

SELECT array\_agg(name) FROM employees;

Результат:

{John Doe, Jane Smith, Michael Johnson}

7) array: функция создает массив из указанных аргументов.

Пример запроса: **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

SELECT array[1, 2, 3];

Результат:

{1, 2, 3}

# Изменение данных:

1) UPDATE: функция используется для изменения данных в таблице. Она позволяет обновлять значения столбцов в соответствии с заданными условиями. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

Пример запроса:

UPDATE employees

SET salary = 5000

WHERE department = 'IT';

Результат: все записи в таблице "employees" с отделом 'IT' будут иметь значение salary равное 5000.

2) INSERT: функция используется для вставки новых записей в таблицу. Она позволяет указывать значения для конкретных столбцов или вставлять значения по умолчанию. **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

Пример запроса:

INSERT INTO employees (name, department, salary)

VALUES ('John Doe', 'HR', 4000);

Результат: в таблицу "employees" будет вставлена новая запись со значениями 'John Doe', 'HR' и 4000 для столбцов name, department и salary соответственно.

3) DELETE: функция используется для удаления записей из таблицы. Она позволяет указывать условия для определения, какие записи следует удалить.

Пример запроса: **(МОДУЛЬ 4. УГЛУБЛЕНИЕ В SQL)**

DELETE FROM employees

WHERE department = 'IT';

Результат: все записи из таблицы "employees" с отделом 'IT' будут удалены.

4) TRUNCATE: функция используется для удаления всех записей из таблицы в очень эффективном и быстром режиме. Она обычно используется для удаления всех данных перед повторным заполнением таблицы.

Пример запроса:

TRUNCATE TABLE employees;

Результат: все записи из таблицы "employees" будут удалены.

Это лишь несколько примеров функций в PostgreSQL, используемых для изменения или модификации данных в базе данных. Все они предоставляют удобные способы для выполнения операций DML (Data Manipulation Language) и обеспечения точек изменения в данных.

# 

# NULL значения:

1. IS NULL - проверяет, является ли значение NULL:

SELECT \* FROM table\_name WHERE column\_name IS NULL

2. IS NOT NULL - проверяет, что значение не является NULL:

SELECT \* FROM table\_name WHERE column\_name IS NOT NULL;

Пример результата: **(МОДУЛЬ 2. РАБОТА С БАЗАМИ ДАННЫХ)**

column\_name

-----------

value1

value2

3. COALESCE - возвращает первое не-NULL значение из списка:

SELECT column\_name, COALESCE(column\_name, 'N/A') AS result

FROM table\_name;

Пример результата: **(МОДУЛЬ 6. POSTGRESQL)**

column\_name | result

------------|--------

value1 | value1

NULL | N/A

value3 | value3

4. NULLIF - возвращает NULL, если два значения равны, иначе возвращает первое значение:

SELECT NULLIF(value1, value2) AS result FROM table\_name;

Пример результата:

result

------

value1

NULL

value3

5. NVL - возвращает второе значение, если первое значение NULL, иначе возвращает первое значение:

SELECT NVL(column\_name, 'N/A') AS result FROM table\_name;

Пример результата:

result

------

value1

N/A

value3

6. IIF - возвращает второе значение, если первое значение NULL, иначе возвращает третье значение:

SELECT IIF(value1 IS NULL, 'N/A', value1) AS result FROM table\_name;

Пример результата:

result

------

value1

N/A

value

# Представления:

Функции работы с обычными представлениями:

1. Создание обычного представления:

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

2. Обновление данных в обычном представлении:

CREATE OR REPLACE RULE view\_name AS

ON UPDATE TO view\_name

DO INSTEAD

UPDATE table\_name

SET column1 = new\_value1, column2 = new\_value2, ...

WHERE condition;

3. Вставка данных в обычное представление:

CREATE OR REPLACE RULE view\_name AS

ON INSERT TO view\_name

DO INSTEAD

INSERT INTO table\_name (column1, column2, ...)

VALUES (new\_value1, new\_value2, ...);

4. Удаление данных из обычного представления:

CREATE OR REPLACE RULE view\_name AS

ON DELETE TO view\_name

DO INSTEAD

DELETE FROM table\_name

WHERE condition;

Функции работы с материализованными представлениями:

1. Создание материализованного представления: **(МОДУЛЬ 6. POSTGRESQL)**

CREATE MATERIALIZED VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

2. Обновление данных в материализованном представлении:

REFRESH MATERIALIZED VIEW view\_name; **(МОДУЛЬ 6. POSTGRESQL)**

3. Принудительное обновление данных в материализованном представлении:

REFRESH MATERIALIZED VIEW CONCURRENTLY view\_name;

4. Удаление материализованного представления:

DROP MATERIALIZED VIEW view\_name;

В результате работы с обычными представлениями в PostgreSQL можно создавать виртуальные таблицы, а с материализованными представлениями можно создавать таблицы, которые сохраняют копию данных для оптимизации запросов.

# Анализ запроса:

1. EXPLAIN:

Функция EXPLAIN предназначена для анализа плана выполнения запроса. Она позволяет получить информацию о том, как PostgreSQL выполняет запрос, какие индексы используются, какие операции выполняются и другие детали выполнения запроса.

Пример использования:

EXPLAIN SELECT \* FROM users WHERE age > 30;

Результат:

QUERY PLAN

-------------------------

Seq Scan on users

Filter: (age > 30)

2. EXPLAIN ANALYZE:

Функция EXPLAIN ANALYZE выполняет анализ плана выполнения запроса и замеряет время выполнения. Она предоставляет детальную статистику о выполнении запроса, включая время выполнения каждой операции и общее время выполнения.

Пример использования:

EXPLAIN ANALYZE SELECT \* FROM users WHERE age > 30;

Результат:

QUERY PLAN

---------------------------------------------------------------------------------------------------------------------

Seq Scan on users

Filter: (age > 30)

(2 rows)

(actual time=0.012..0.017 rows=10 loops=1)

3. SET ENABLE\_SEQSCAN:

Команда SET ENABLE\_SEQSCAN позволяет включить или отключить последовательное сканирование таблиц. Последовательное сканирование является одним из способов выполнения запросов, и иногда может быть неэффективным, особенно для больших таблиц.

Пример использования:

SET ENABLE\_SEQSCAN TO OFF;

SELECT \* FROM users WHERE age > 30;

Результат:

-- В результате будет использован другой план выполнения запроса без последовательного сканирования таблиц.

4. SET ENABLE\_HASHJOIN:

Команда SET ENABLE\_HASHJOIN позволяет включить или отключить объединение таблиц с использованием хеш-таблицы. Хеш-таблицы могут быть эффективным способом выполнения запросов, особенно для больших объединений таблиц.

Пример использования:

SET ENABLE\_HASHJOIN TO OFF;

SELECT \* FROM users JOIN orders ON users.id = orders.user\_id;

Результат:

-- В результате будет использован другой план выполнения запроса без использования хеш-таблицы для объединения таблиц.

5. ANALYZE:

Команда ANALYZE используется для обновления статистики таблицы. Статистика таблицы является важным элементом для оптимизации запросов, и ее актуализация может улучшить производительность выполнения запросов.

Пример использования:

ANALYZE users;

Результат:

-- Статистика таблицы users будет обновлена.

Это лишь некоторые функции SQL-команд управления выполнением запросов в PostgreSQL. С помощью этих функций, а также других возможностей PostgreSQL, можно оптимизировать и управлять выполнением запросов для достижения наилучшей производительности.

# JSON-значения:

1. Функции для работы с JSON-значениями:

а) jsonb\_extract\_path(jsonb\_expression, VARIADIC string[]): Извлекает значение из JSON-значения по указанному пути.

Пример:

SELECT jsonb\_extract\_path('{"name": {"first": "John", "last": "Doe"}, "age": 30}', 'name', 'first');

Результат:

jsonb\_extract\_path

--------------------

"John"

б) jsonb\_agg(expression): Преобразует результаты запроса в JSON-массив.

Пример:

SELECT jsonb\_agg(name) FROM users;

Результат:

jsonb\_agg

---------------------------------

["John", "Alice", "Bob"]

в) jsonb\_set(target jsonb, path text[], new\_value jsonb, create\_missing boolean): Заменяет значение по указанному пути в JSON-значении на новое значение.

Пример:

SELECT jsonb\_set('{"name": {"first": "John", "last": "Doe"}, "age": 30}', '{name, last}', '"Smith"');

Результат:

jsonb\_set

---------------------------------------------------------

{"name": {"first": "John", "last": "Smith"}, "age": 30}

2. Операторы для работы с JSON-значениями:

а) ->: Получает значение по ключу из JSON-значения.

Пример:

SELECT '{"name": {"first": "John", "last": "Doe"}, "age": 30}' -> 'name' ->> 'first';

Результат:

?column?

----------

John

б) ->>: Получает значение по ключу из JSON-значения и преобразует его в строку.

Пример:

SELECT '{"name": {"first": "John", "last": "Doe"}, "age": 30}' ->> 'age';

Результат:

?column?

----------

30

в) @>: Проверяет, содержит ли JSON-значение все ключи и значения другого JSON-значения.

Пример:

SELECT '{"name": "John", "age": 30, "city": "New York"}' @> '{"name": "John", "age": 30}';

Результат:

?column?

----------

t

Такие функции и операторы помогают упростить работу с JSON-значениями в PostgreSQL и позволяют выполнять различные операции с данными в формате JSON.

# Временные таблицы

Оператор WITH в PostgreSQL также позволяет использовать общие табличные выражения (Common Table Expressions, CTE), которые представляют именованные подзапросы. CTE используются для создания временных таблиц или подзапросов, которые могут быть ссылками в других запросах в рамках одного оператора WITH. **(МОДУЛЬ 6. POSTGRESQL)**

Пример использования CTE с оператором WITH:

WITH

cte\_name (column1, column2) AS (

SELECT column1, column2

FROM table1

WHERE condition

)

SELECT \*

FROM cte\_name

WHERE column2 = value;

Обратите внимание, что cte\_name - это имя, которое вы назначаете вашей CTE для дальнейшего использования в запросе. Список столбцов (column1, column2) в скобках определяет столбцы, которые будут доступны в CTE. Затем в самом операторе SELECT можно использовать эту CTE как временную таблицу для выборки данных.

CTE предлагает более читаемый и модульный подход к созданию и использованию именованных подзапросов и облегчает понимание и сопровождение сложных запросов.