1.В SQL порядок исполнения запроса обычно происходит следующим образом:

1. **FROM:** Выбор данных из таблицы или представления.
2. **WHERE:** Применение условий фильтрации для выбора конкретных строк данных.
3. **GROUP BY:** Группировка данных по определенным столбцам.
4. **HAVING:** Применение условий фильтрации для групп данных, аналогично WHERE, но применяется к результатам GROUP BY.
5. **SELECT:** Выбор столбцов для вывода.
6. **ORDER BY:** Сортировка результатов по указанным столбцам и направлению (ASC для по возрастанию, DESC для по убыванию).
7. **LIMIT/OFFSET:** Ограничение числа возвращаемых строк (опционально).

Пример SQL-запроса с использованием всех этих операторов:

SELECT column1, column2, COUNT(\*)

FROM table\_name

WHERE condition

GROUP BY column1, column2

HAVING condition

ORDER BY column1 ASC, column2 DESC

LIMIT 10 OFFSET 5;

Этот запрос выбирает определенные столбцы из таблицы, фильтрует строки с использованием условия WHERE, группирует результаты по столбцам column1 и column2, применяет дополнительное условие к группам с использованием HAVING, сортирует результаты по столбцам column1 по возрастанию и column2 по убыванию, и, наконец, ограничивает количество возвращаемых строк и определяет смещение с использованием LIMIT и OFFSET.

2. Materialized view (материализованный вид) в SQL - это объект базы данных, который хранит результаты запроса в виде таблицы. В отличие от обычных видов (views), которые являются виртуальными и не хранят данные сами по себе, материализованные виды физически сохраняют данные, что позволяет повысить производительность запросов за счет увеличения скорости доступа к данным, но может потребоваться больше места для их хранения и больше усилий для обслуживания.

Пример создания материализованного вида в SQL:

-- Создание материализованного вида

CREATE MATERIALIZED VIEW mv\_example

AS

SELECT column1, column2, COUNT(\*)

FROM table\_name

WHERE condition

GROUP BY column1, column2;

-- Обновление материализованного вида (это обновляет данные)

REFRESH MATERIALIZED VIEW mv\_example;

-- Запрос материализованного вида

SELECT \* FROM mv\_example;

В этом примере **mv\_example** - это имя материализованного вида. Запрос **CREATE MATERIALIZED VIEW** определяет структуру и начальные данные материализованного вида на основе указанного запроса. Запрос **REFRESH MATERIALIZED VIEW** используется для обновления данных в материализованном виде, и его можно запросить так же, как и обычную таблицу.

Стоит отметить, что синтаксис и возможности материализованных видов могут различаться в зависимости от системы управления базой данных (например, PostgreSQL, Oracle, SQL Server), поэтому важно ознакомиться с документацией конкретной СУБД для получения точной и подробной информации.

4. В SQL "вид" (view) представляет собой виртуальную таблицу, основанную на результатах выполнения запроса. Вместо того чтобы физически хранить данные, вид предоставляет удобный способ абстрагирования сложных запросов и представления данных. Вот пример использования и создания вида:

-- Создание вида

CREATE VIEW my\_view AS

SELECT column1, column2

FROM my\_table

WHERE condition;

-- Запрос данных с использованием вида

SELECT \* FROM my\_view;

**Объяснение:**

1. **Создание вида (CREATE VIEW):** В этом примере создается вид с именем **my\_view**, который основан на результате запроса **SELECT column1, column2 FROM my\_table WHERE condition;**. Важно отметить, что при создании вида не создается новая таблица с данными. Вид сохраняет только определение запроса.
2. **Запрос данных с использованием вида (SELECT \* FROM my\_view):** После создания вида его можно использовать в запросах, как если бы это была обычная таблица. Запрос **SELECT \* FROM my\_view** возвращает результаты, соответствующие условиям, определенным в запросе создания вида.

Виды облегчают работу с базой данных, позволяя абстрагироваться от сложных запросов и упрощать повторное использование запросов.

Важно отметить, что виды могут быть обновлены, чтобы отражать изменения в базовых данных, но они не хранят собственные данные. Вместо этого они предоставляют виртуальное представление данных из одного или нескольких исходных запросов.

4. В SQL оператор JOIN используется для объединения строк из двух или более таблиц на основе определенного условия. Вот несколько типов JOIN-операторов:

1. **INNER JOIN (Внутреннее объединение):**
   * **Синтаксис:**

SELECT \*

FROM table1

INNER JOIN table2 ON table1.column = table2.column;

* + Возвращает строки, у которых есть соответствующие значения в обеих таблицах по указанным столбцам.

1. **LEFT JOIN (Левое объединение):**
   * **Синтаксис:**

SELECT \*

FROM table1

LEFT JOIN table2 ON table1.column = table2.column;

* + Возвращает все строки из левой таблицы (**table1**) и соответствующие строки из правой таблицы (**table2**). Если нет соответствия, значения из правой таблицы будут NULL.

1. **RIGHT JOIN (Правое объединение):**
   * **Синтаксис:**

SELECT \*

FROM table1

RIGHT JOIN table2 ON table1.column = table2.column;

* + Возвращает все строки из правой таблицы (**table2**) и соответствующие строки из левой таблицы (**table1**). Если нет соответствия, значения из левой таблицы будут NULL.

1. **FULL JOIN (Полное объединение):**
   * **Синтаксис:**

SELECT \*

FROM table1

FULL JOIN table2 ON table1.column = table2.column;

* + Возвращает все строки из обеих таблиц. Если нет соответствия, соответствующие значения будут NULL.

1. **CROSS JOIN (Кросс-объединение):**
   * **Синтаксис:**

SELECT \*

FROM table1

CROSS JOIN table2;

* + Возвращает декартово произведение строк из обеих таблиц, то есть каждая строка из первой таблицы соединяется со всеми строками из второй таблицы.

5.   
Use Case в SQL (Structured Query Language) представляет собой выражение, которое используется для оценки условия и выполнения действий в зависимости от того, соответствует ли это условие истине или лжи. Оно часто применяется в операторе **CASE**, который может использоваться в различных частях SQL-запроса.

Пример использования **CASE** в SQL:

SELECT column1,

column2,

CASE

WHEN condition1 THEN 'Value1'

WHEN condition2 THEN 'Value2'

ELSE 'DefaultValue'

END AS custom\_column

FROM table\_name;

Объяснение:

1. **CASE оператор:** Он начинает блок **CASE** и определяет, какие значения будут возвращены в зависимости от выполнения условий.
2. **WHEN condition THEN result:** Определение условий и соответствующих значений для возвращения. Если **condition1** истинно, будет возвращено **'Value1'**, если **condition2** истинно, то **'Value2'**. Если ни одно из условий не истинно, будет возвращено **'DefaultValue'** из блока **ELSE**.
3. **END AS custom\_column:** Завершает блок **CASE** и определяет, как будет называться созданная колонка.

Это может быть использовано в различных контекстах, например, для создания новых категорий на основе значений в столбце, присваивания статуса в зависимости от условий и т.д.

Пример:

SELECT product\_name,

quantity,

CASE

WHEN quantity > 10 THEN 'High Demand'

WHEN quantity > 5 THEN 'Moderate Demand'

ELSE 'Low Demand'

END AS demand\_level

FROM products;

Этот запрос создаст колонку **demand\_level**, которая будет указывать уровень спроса на продукт в зависимости от его количества.

1. В SQL существует несколько агрегатных функций, таких как AVG (среднее значение), MIN (минимальное значение) и MAX (максимальное значение). Агрегатная функция выполняет вычисление на наборе значений и возвращает одиночное значение. Вот как они используются:

**AVG (Среднее значение):**

* Используется для вычисления среднего значения числового столбца.

SELECT AVG(column\_name) AS average\_value

FROM table\_name;

**MIN (Минимальное значение):**

* Используется для нахождения минимального значения числового или текстового столбца.

SELECT MIN(column\_name) AS min\_value

FROM table\_name;

**MAX (Максимальное значение):**

* Используется для нахождения максимального значения числового или текстового столбца.

SELECT MAX(column\_name) AS max\_value

FROM table\_name;

Примеры с использованием этих функций:

-- Пример среднего значения

SELECT AVG(salary) AS average\_salary

FROM employees;

-- Пример минимального значения

SELECT MIN(order\_date) AS earliest\_order\_date

FROM orders;

-- Пример максимального значения

SELECT MAX(product\_price) AS highest\_price

FROM products;

В этих примерах агрегатные функции используются для вычисления среднего значения зарплаты сотрудников, нахождения самой ранней даты заказа и определения самой высокой цены продукта соответственно.

Ключевые слова **WITH** и **WITH RECURSIVE** в SQL используются для создания временных таблиц (common table expressions, CTE) в рамках запроса. **WITH** предоставляет именованные подзапросы, которые могут использоваться внутри основного запроса, тогда как **WITH RECURSIVE** предоставляет тот же функционал, но позволяет рекурсивно ссылаться на результаты самого себя.

### Пример с WITH:

### WITH temp\_table AS (

### SELECT column1, column2

### FROM table1

### WHERE condition

### )

### SELECT \*

### FROM temp\_table

### WHERE column2 > 10;

### В этом примере temp\_table - это временная таблица, которая создается для использования внутри основного запроса. Она содержит результаты SELECT column1, column2 FROM table1 WHERE condition, и затем мы выбираем строки из этой временной таблицы, где column2 > 10.

### Пример с WITH RECURSIVE:

### WITH RECURSIVE recursive\_table AS (

### SELECT start\_value

### FROM table1

### WHERE condition

### UNION

### SELECT rt.start\_value + 1

### FROM recursive\_table rt

### WHERE rt.start\_value < 10

### )

### SELECT \*

### FROM recursive\_table;

В этом примере **recursive\_table** - это рекурсивная временная таблица. Она начинается с выбора **start\_value** из **table1**, а затем рекурсивно добавляет **start\_value + 1** до тех пор, пока **start\_value** меньше 10. Результат представляет собой последовательность значений от начального значения до 10.

Использование **WITH RECURSIVE** полезно при работе с иерархическими или рекурсивными структурами данных, например, при работе с деревьями или графами.

### В SQL коррелированный подзапрос выполняется для каждой строки внешнего запроса. Это означает, что количество выполнений коррелированного подзапроса будет равно количеству строк, полученных внешним запросом.

### Напротив, простой подзапрос выполняется только один раз и возвращает результат, который затем можно использовать внешнему запросу.

Давайте разберем это на примере. Предположим, у нас есть две таблицы: "Employees" (Сотрудники) и "Departments" (Отделы). Мы хотим получить список отделов, в которых работает более двух сотрудников.

В SQL для обновления нематериализованных представлений (называемых также виртуальными представлениями или просто представлениями) необходимо изменить их определение (тело запроса) с помощью оператора CREATE OR REPLACE VIEW.

Процесс обновления нематериализованного представления включает в себя следующие шаги:

1. Определите новое тело запроса представления, которое соответствует ваши актуальным требованиям. В новом теле запроса вы можете изменять, добавлять или удалять столбцы, указывать другие условия отбора, задавать новые вычисляемые столбцы и т. д.

2. Используя оператор CREATE OR REPLACE VIEW, создайте представление с новым определением. Этот оператор заменяет существующее представление новым определением.

Вот пример синтаксиса обновления нематериализованного представления:

CREATE OR REPLACE VIEW Имя\_представления AS

<новое\_тело\_запроса>;

Ключевое слово **JOIN** в SQL используется для объединения строк из двух или более таблиц на основе связанного столбца между ними. Операция **JOIN** обычно используется с ключевым словом **ON** для указания условия объединения. Однако существует также возможность использования ключевого слова **USING** для более удобного объединения по столбцу с одинаковым именем в обеих таблицах.

**NATURAL JOIN** в SQL представляет собой операцию объединения таблиц, при которой столбцы для сопоставления автоматически выбираются на основе их одинаковых имен. Это означает, что при использовании **NATURAL JOIN** не требуется явно указывать столбцы для сравнения, так как SQL автоматически определяет соответствующие столбцы по их именам.