BigQuery

Use Cloud BigQuery to run super-fast, SQL-like queries against append-only tables. BigQuery makes it easy to:

- Control who can view and query your data.
- Use a variety of third-party tools to access data on BigQuery, such as tools that load or visualize your data.
- Use datasets to organize and control access to tables, and construct jobs for BigQuery to execute (load, export, query, or copy data).

Q 1. explain types of join in detail

SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables.

We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with **WHERE clause** to filter data retrieval.

When we need to show the columns in the output, from more than one table, then use Joins,

Joins are of 3 types.

Cross join, inner join (natural join), outer join

In joins if n tables are there, then minimum n-1 join conditions will be there

Cross join		When you want to combine every row of one table with every row of other table, then we use cross join If in one table m rows are there and in other table n rows are there, then output will contain m*n rows
Inner Join	Equi join	While joining multiple tables, if we add condition, and if condition is based on = operator, then it is called as equi join
	Non equi join	While joining multiple tables, if we add condition, and if the condition is based on operator which is other than =, then it is called as non equi join
	Self join	While joining multiple tables, if we join the table with itself then it is called as self join
Outer join	Right outer join	If we want to retrieve matching as well as nonmatching rows from the table on the right side in from clause, then use right outer join
	Left outer join	If we want to retrieve matching as well as nonmatching rows from the table on the left side in from clause, then use left outer join
	Full outer join	If we want to retrieve matching as well as nonmatching rows from both the tables in the from clause, then use full outer join

Syntax:

The syntax for SQL INNER JOIN is:

SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;

SQL LEFT JOIN

- LEFT JOIN returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.
- **Syntax**
- The syntax of LEFT JOIN in SQL is**:**
- **SELECT** table1.column1,table1.column2,table2.column1,....
  FROM table1
  LEFT JOIN table2
  ON table1.matching_column = table2.matching_column;

SQL RIGHT JOIN

RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join.It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

The syntax of RIGHT JOIN in SQL is:

SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;

Q 2. Write an SQL query to join three tables.

There may occur some situations sometimes where data needs to be fetched from three or more tables. This article deals with two approaches to achieve it.

Example:
Creating three tables:

1. student
2. marks
3. details

Two approaches to join three or more tables:
1. Using joins in sql to join the table:
The same logic is applied which is done to join 2 tables i.e. **minimum** number of join statements to join **n** tables are **(n-1)**.
Query:

```
select s_name, score, status, address_city, email_id,

accomplishments from student s inner join marks m on

s.s_id = m.s_id inner join details d on

d.school_id = m.school_id;
```

2. Using parent-child relationship:
This is rather an interesting approach. Create column **X** as primary key in one table and as foreign key in another table (i.e creating a parent-child relationship).
Let's look in the tables created:
s_id is the **primary key** in student table and is **foreign key** in marks table. **(student (parent) – marks(child))**.
school_id is the **primary key** in marks table and **foreign key** in details table. **(marks(parent) – details(child))**.

Query:

```
select s_name, score, status, address_city,

email_id, accomplishments from student s,

marks m, details d where s.s_id = m.s_id and

m.school_id = d.school_id;
```

Q 3. explain group by clause

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e. if a particular column has the same values in different rows then it will arrange these rows in a group.

Features

- GROUP BY clause is used with the SELECT statement.
- In the query, the GROUP BY clause is placed after the WHERE clause.

- In the query, the GROUP BY clause is placed before the ORDER BY clause if used.
- In the query, the Group BY clause is placed before the Having clause.

  SELECT column1, function_name(column2)
  FROM table_name
  WHERE condition
  GROUP BY column1, column2
  ORDER BY column1, column2;

Group By single column

Group By single column means, placing all the rows with the same value of only that particular column in one group. Consider the query as shown below:

Query:

SELECT name, SUM(sal) FROM emp

GROUP BY name;

Group By Multiple Columns

Group by multiple columns is say, for example, **GROUP BY column1, column2**. This means placing all the rows with the same values of columns **column 1** and **column 2** in one group. Consider the below query:

Query:

SELECT SUBJECT, YEAR, Count(*)

FROM Student

GROUP BY SUBJECT, YEAR;

HAVING Clause in GROUP BY Clause

We know that the WHERE clause is used to place conditions on columns but what if we want to place conditions on groups? This is where the HAVING clause comes into use. We can use the HAVING clause to place conditions to decide which group will be part of the final result set. Also, we can not use aggregate functions like SUM(), COUNT(), etc. with the WHERE clause. So we have to use the HAVING clause if we want to use any of these functions in the conditions.

Syntax:

SELECT column1, function_name(column2)

FROM table_name

WHERE condition

GROUP BY column1, column2

HAVING condition

ORDER BY column1, column2;

Example:

SELECT NAME, SUM(sal) FROM Emp

GROUP BY name

HAVING SUM(sal)>50000;

Q 4. what is subquery

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query. Important rules for Subqueries:

- You can place the Subquery in a number of SQL clauses: WHERE clause, HAVING clause, FROM clause. Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.
- A subquery is a query within another query. The outer query is called as main query and inner query is called as subquery.
- The subquery generally executes first when the subquery doesn't have any co-relation with the main query, when there is a co-relation the parser takes the decision on the fly on which query to execute on precedence and uses the output of the subquery accordingly.
- Subquery must be enclosed in parentheses.
- Subqueries are on the right side of the comparison operator.
- ORDER BY command cannot be used in a Subquery. GROUPBY command can be used to perform same function as ORDER BY command.
- Use single-row operators with singlerow Subqueries. Use multiple-row operators with multiple-row Subqueries.

Syntax

SELECT column_name

FROM table_name

WHERE column_name expression operator

  (SELECT column_name FROM table_name WHERE ...);

Q.5 explain PL/SQL stored procedure, function

Procedures and functions are PL/SQL blocks that have been defined with a specified name.

Syntax

CREATE PROCEDURE syntax is**:**

```
CREATE PROCEDURE procedure_name
@Parameter1 INT,
@Parameter2 VARCHAR(50) = NULL,
@ReturnValue INT OUTPUT
AS
BEGIN
END
GO
```

Example

In this example, we will create a procedure in PL/SQL

```
CREATE PROCEDURE GetStudentDetails
    @StudentID int = 0
AS
BEGIN
    SET NOCOUNT ON;
    SELECT FirstName, LastName, BirthDate, City, Country
    FROM Students WHERE StudentID=@StudentID
END
GO
```

Parameters in Procedures

In PL/SQL, parameters are used to pass values into procedures. There are three types of parameters used in procedures:

IN parameters

- Used to pass values into the procedure
- Read-only inside the procedure
- Can be a variable, literal value, or expression in the calling statement.

OUT parameters

- Used to return values from the procedure to the calling program
- Read-write inside the procedure

- Must be a variable in the calling statement to hold the returned value

IN OUT parameters

- Used for both passing values into and returning values from the procedure
- Read-write inside the procedure
- Must be a variable in the calling statement

Drop Procedure in PL/SQL

To drop a procedure in PL/SQL use the **DROP PROCEDURE** command
Syntax
DROP PROCEDURE syntax is:
DROP PROCEDURE procedure_name

Functions:

PL/SQL functions are reusable blocks of code that can be used to perform specific tasks. They are similar to **procedures** but must always return a value.

A function in PL/SQL contains:

- **Function Header**: The function header includes the function name and an optional parameter list. It is the first part of the function and specifies the name and parameters.
- **Function Body**: The function body contains the executable statements that implement the specific logic. It can include declarative statements, executable statements, and exception-handling statements.

Create Function in PL/SQL

To create a procedure in PL/SQL, use the **CREATE FUNCTION statement.**

Syntax

The syntax to create a function in PL/SQL is given below:

CREATE [OR REPLACE] FUNCTION function_name
(parameter_name type [, …])

— This statement is must for functions
RETURN return_datatype

{IS | AS}

BEGIN
— program code

[EXCEPTION
exception_section;

END [function_name];

Q.6 What are cursors, how are they used in real world databases?

A cursor is a temporary work area allocated by the database server when performing Data Manipulation Language (DML) operations on a table. It's essentially a pointer that allows you to navigate through a result set one row at a time.

Types of Cursors

There are two primary types of cursors:

1. **Implicit Cursors:**
   - Automatically created by the database system when executing DML statements (INSERT, UPDATE, DELETE).
   - Used internally by the database to manage changes.
   - Programmers don't have direct control over them.
2. **Explicit Cursors:**
   - Created by the programmer to process data row by row.
   - Offer more control over data manipulation.
   - Commonly used for complex data processing tasks.

How Cursors are Used in Real-World Databases

Cursors are employed in various scenarios where granular control over data is required. Here are some common use cases:

1. Data Validation:

- Iterating through each record to check for data consistency and integrity.
- Identifying and correcting invalid data before processing.

2. Complex Data Transformations:

- Manipulating data based on specific conditions or calculations.
- Updating multiple tables or columns based on the results of processing each row.

3. Reporting and Analysis:

- Generating custom reports or analyses that require row-by-row processing.
- Calculating aggregates or statistics based on specific criteria.

4. Batch Processing:

- Handling large datasets in smaller chunks for efficient processing.
- Updating or deleting data in batches to improve performance.

Example of Cursor Usage

Imagine a scenario where you need to update the salary of employees based on their performance rating. You could use a cursor to iterate through each employee's record, check their rating, calculate the new salary, and update the database accordingly.

SQL

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, performance_rating FROM employees;
  emp_rec emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_rec;
    EXIT WHEN emp_cursor%NOTFOUND;

    -- Calculate new salary based on performance_rating
    UPDATE employees SET salary = new_salary
    WHERE employee_id = emp_rec.employee_id;
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

Q. 7 Explain Window functions in sql

All aggregate functions can be used as window function	Sum, avg, count, min, max
Row_number()	To assign numbers sequentially within window
Rank()	To assign numbers sequentially, but if 2 values are same, then assign same rank, but then it skips next numbers

Dense_rank()	To assign numbers sequentially, but if 2 values are same, then assign same dense rank, but it does not skips next numbers Hence most of the places useful for finding nth topmost
First_value(field)	To find the first value , within window
Last_value(field)	To find the last value within window
Lag(field,n)	To find next n th value use lag
Lead(field,2)	To find previous n th value use lead

When to use window functions

1. When we want to display value of aggregate function, but along with, we also want to show the column which is not used in group by clause
2. To find nth topmost value, it should also display if the value is repeating
3. If we want to find nth topmost sal within each department

To find the next highest or previous highest

Key Components of a Window Function:

- **OVER clause:** Defines the set of rows (window) to be considered for the calculation.
- **PARTITION BY clause:** Divides the result set into partitions based on specified columns.
- **ORDER BY clause:** Specifies the order of rows within each partition.
- **FRAME clause:** Defines the specific range of rows within a partition (optional).

Types of Window Functions:

1. **Aggregate Window Functions:**
    - Apply aggregate functions (SUM, AVG, COUNT, MIN, MAX) over a window.
    - Examples:
        - SUM(column_name) OVER (PARTITION BY department_id): Calculates the total for each department.
        - AVG(salary) OVER (): Calculates the average salary across the entire dataset.
2. **Ranking Window Functions:**
    - Assign a rank or row number to each row within a partition.
    - Examples:
        - ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC): Assigns a sequential number to each employee within a department based on descending salary.

        - RANK() OVER (PARTITION BY department_id ORDER BY salary DESC): Assigns a rank to each employee within a department based on salary, with gaps for ties.
        - DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC): Assigns a rank to each employee within a department based on salary, without gaps for ties.

3. **Distribution Window Functions:**
    - Divide the result set into groups based on specified criteria.
    - Examples:
        - NTILE(4) OVER (ORDER BY salary): Divides the result set into four groups based on salary quartiles.
        - PERCENT_RANK() OVER (): Calculates the percentile rank of each row within the entire result set.

Example:

Consider a table named employees with columns employee_id, department_id, and salary.

SQL

```sql
SELECT
   employee_id,
   department_id,
   salary,
   AVG(salary) OVER (PARTITION BY department_id) AS avg_department_salary,
   ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS row_num
FROM
   employees;
```

This query calculates the average salary for each department and assigns a row number to each employee within their department based on descending salary.

Common Use Cases:

- Calculating running totals or moving averages.
- Ranking and percentile calculations.
- Data analysis and exploration.
- Complex reporting.

Q.8 How to Delete duplicate records in SQL?

1. Identify Duplicate Records: Determine the columns that define uniqueness.
2. Choose a Record to Keep: Decide which record to retain in case of duplicates (e.g., the latest, oldest, or one with a specific condition).
3. Delete Extra Records: Construct a SQL query to eliminate the unwanted duplicates.

Methods

1. Using ROW_NUMBER() or RANK() (SQL Server, Oracle, PostgreSQL)

This method assigns a sequential number to each row within a partition. Duplicate rows will have the same number. You can delete those with a rank greater than 1.

```
WITH DuplicateRows AS (
  SELECT *,
       ROW_NUMBER() OVER (PARTITION BY Column1, Column2 ORDER BY Column3) AS RowNum
  FROM YourTable
)
DELETE FROM DuplicateRows
WHERE RowNum > 1;
```

2. Using GROUP BY and HAVING (Most SQL dialects)

Identify duplicate groups based on specific columns and delete records that aren't part of the first occurrence.

```
DELETE FROM YourTable
WHERE ID NOT IN (
  SELECT MIN(ID)
  FROM YourTable
  GROUP BY Column1, Column2
);
```

3. Using a Temporary Table (Most SQL dialects)

Create a temporary table with unique records, then delete and repopulate the original table.

```
CREATE TABLE #TempTable (
   -- Columns
);
INSERT INTO #TempTable (Column1, Column2, ...)
SELECT DISTINCT Column1, Column2, ...
FROM YourTable;
TRUNCATE TABLE YourTable;
INSERT INTO YourTable (Column1, Column2, ...)
SELECT * FROM #TempTable;
DROP TABLE #TempTable;
```

4. Using Self-Join (Most SQL dialects)

Find duplicate records by joining the table with itself.

SQL

```
DELETE a
FROM YourTable a
INNER JOIN YourTable b
ON a.ID > b.ID AND a.Column1 = b.Column1 AND a.Column2 = b.Column2;
```

Q.9 How to find duplicate records in SQL?

Method 1: Using GROUP BY and HAVING

This is a common method to identify duplicate records based on specific columns.

```
SELECT column1, column2, COUNT(*) AS duplicate_count
FROM your_table
GROUP BY column1, column2
HAVING COUNT(*) > 1;
```

- Replace your_table with the actual name of your table.
- Replace column1 and column2 with the columns you want to check for duplicates.

Explanation:

- The GROUP BY clause groups rows based on column1 and column2.
- The COUNT(*) function counts the number of rows in each group.
- The HAVING clause filters groups where the count is greater than 1, indicating duplicates.

Method 2: Self-Join

This method can be used to find duplicate records and display all columns of the duplicated rows.

```
SELECT a.*

FROM your_table a

INNER JOIN your_table b

ON a.column1 = b.column1

AND a.column2 = b.column2

AND a.id <> b.id;
```

- Replace your_table with the actual name of your table.
- Replace column1, column2, and id with appropriate column names.

Explanation:

- The table is joined with itself based on the specified columns.
- The WHERE clause ensures that the joined rows have different id values, indicating duplicates.

Method 3: Using ROW_NUMBER() (SQL Server, Oracle, PostgreSQL)

This method assigns a sequential number to each row within a partition. Duplicate rows will have the same number.

```
WITH DuplicateRows AS (

  SELECT *,

         ROW_NUMBER() OVER (PARTITION BY column1, column2 ORDER BY id) AS RowNum

  FROM your_table

)

SELECT *

FROM DuplicateRows

WHERE RowNum > 1;
```

- Replace your_table, column1, column2, and id with appropriate values.

Explanation:

- The ROW_NUMBER() function assigns a sequential number to each row within the partition defined by column1 and column2.
- The WHERE clause filters rows with a RowNum greater than 1, indicating duplicates.

Q.10 what is difference between Delete and truncate?

Both DELETE and TRUNCATE commands are used to remove data from a table in SQL, but they have distinct characteristics:

DELETE

- **Removes specific rows:** You can specify conditions using the WHERE clause to delete particular rows.
- **DML command:** Data Manipulation Language, allowing for granular control over data modification.
- **Slower than TRUNCATE:** Deletes rows one by one, which can be time-consuming for large tables.
- **Can be rolled back:** Changes made with DELETE can be undone using ROLLBACK.
- **Triggers execution:** Triggers associated with the DELETE operation will be fired.

TRUNCATE

- **Removes all rows:** Deletes all data from a table at once.
- **DDL command:** Data Definition Language, used for structural changes to the table.

- **Faster than DELETE:** Removes all data in a single operation.
- **Cannot be rolled back:** Changes made with TRUNCATE are permanent.
- **No trigger execution:** Triggers are not fired when using TRUNCATE.

Q.11 Explain Outer join, full outer vs left outer vs right outer

Outer Joins: Full, Left, and Right

Outer joins are used in SQL to combine rows from two tables based on a related column, but unlike inner joins, they also include rows from one or both tables that don't have corresponding matches in the other table.

Full Outer Join

- **Returns all rows when there is a match in either left or right table.**
- Includes unmatched rows from both tables.
- Often represented by FULL OUTER JOIN or simply FULL JOIN.

Example:

SQL

```
SELECT *

FROM table1

FULL OUTER JOIN table2

ON table1.column = table2.column;
```

Left Outer Join

- **Returns all rows from the left table, and the matched rows from the right table.**
- Includes unmatched rows from the left table.
- Often represented by LEFT OUTER JOIN or simply LEFT JOIN.

Example:

SQL

```
SELECT *

FROM table1

LEFT OUTER JOIN table2

ON table1.column = table2.column;
```

Right Outer Join

- **Returns all rows from the right table, and the matched rows from the left table.**
- Includes unmatched rows from the right table.
- Often represented by RIGHT OUTER JOIN or simply RIGHT JOIN.

Example:

SQL

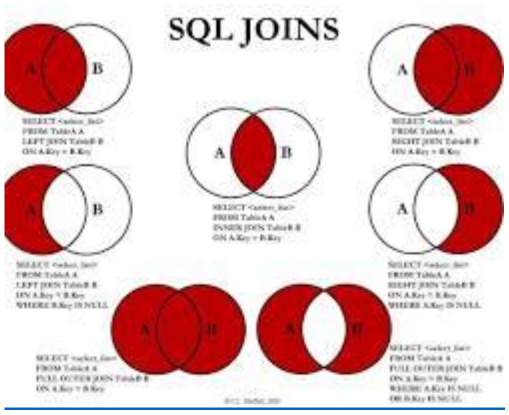
```
SELECT *

FROM table1

RIGHT OUTER JOIN table2

ON table1.column = table2.column;
```

Visual Representation



Venn diagram showing full outer join, left outer join, and right outer join

When to Use Which Join

- **Full Outer Join:** When you need to see all data from both tables, regardless of whether there's a match.
- **Left Outer Join:** When you want to retain all records from the left table, even if there's no match in the right table.
- **Right Outer Join:** When you want to retain all records from the right table, even if there's no match in the left table.

Q.12 Explain In outer join what will happen if there is null value in right side table and valid value in left side table, explain with example

OR

Explain Null safe operator (joins)

In MySQL, the null-safe equality operator <=> is extremely useful when you're joining tables on columns that may contain NULL values.

Example

Table: employees

emp_id	name	dept_id
1	Alice	10
2	Bob	NULL

emp_id	**name**	**dept_id**
3	Carol	30

Table: departments

dept_id	**dept_name**
10	HR
NULL	No Dept
30	Marketing

Join with <=>:

```
SELECT e.emp_id, e.name, d.dept_name
FROM employees e
LEFT JOIN departments d
  ON e.dept_id <=> d.dept_id;
```

Result:

emp_id	**name**	**dept_name**
1	Alice	HR
2	Bob	No Dept
3	Carol	Marketing

Bob matched with No Dept because NULL <=> NULL is TRUE.

Join with =:

```
SELECT e.emp_id, e.name, d.dept_name
FROM employees e
LEFT JOIN departments d
  ON e.dept_id = d.dept_id;
```

Result:

emp_id	**name**	**dept_name**
1	Alice	HR
2	Bob	Null
3	Carol	Marketing

This won't match Bob, result will have NULL dept_name for him

Q.13 Explain In outer join what will happen is there is null value in left side table and valid value in right side table, explain with example

In a right outer join, the focus is on preserving all rows from the *right* table, even if there's no corresponding match in the left table. When considering a null value in the right-side table and a valid value in the left-side table, the behavior is straightforward.

Behavior of Null Values in Right Outer Join

- **Null value in the right side table:** This scenario doesn't affect the inclusion of the row in the result set.
- **Valid value in the left side table:** If there's a corresponding match in the left table, the values from the left table will be included in the result set. Otherwise, null values will appear for the left table's columns.

Example: Let's use the same tables as before:

- **Customers** (CustomerID, CustomerName)
- **Orders** (OrderID, CustomerID, OrderDate)

Imagine a scenario where there's an order with a null CustomerID (perhaps due to data inconsistency).

SQL

```
SELECT *

FROM Customers c

RIGHT JOIN Orders o

ON c.CustomerID = o.CustomerID;
```

In this case, the row with the null CustomerID in the Orders table will still be included in the result set, but the corresponding CustomerID and CustomerName from the Customers table will be null.

Result:

CustomerID	**CustomerName**	**OrderID**	**OrderDate**
NULL	NULL	1003	2023-11-20
1	John Doe	1001	2023-11-15
3	Bob Johnson	1002	2023-12-01

Export to Sheets

Summary

In a right outer join, a null value in the right-side table doesn't prevent the row from being included in the result set. The focus is on preserving all rows from the right table, and null values in the left table's columns will appear when there's no corresponding match.

Q.14 Explain inner join, if the column on which we are taking join has null values then what will happen?

Impact of Null Values on Inner Join

When the column used for the join contains null values, those rows will not be included in the result set.

- Reason: Null values represent an unknown or missing value. SQL cannot compare a null value to any other value, including another null value.
- Result: Rows with null values in the join column will be excluded from the final result.

Example:

Consider two tables:

- Customers (CustomerID, CustomerName)
- Orders (OrderID, CustomerID, OrderDate)

If the CustomerID column in the Orders table has some null values, these orders will not be included in the result set when joining with the Customers table.

SQL

```
SELECT *

FROM Customers c

INNER JOIN Orders o

ON c.CustomerID = o.CustomerID;
```

Only orders with a valid CustomerID will be included in the result.

Q.15 explain Having clause , upsert in sql

HAVING Clause

The **HAVING** clause is used in SQL to filter the results of a GROUP BY clause. It's similar to the WHERE clause, but it operates on groups of rows rather than individual rows.

Syntax:

SQL

```
SELECT column_name, aggregate_function(column_name)

FROM table_name

WHERE condition  // Optional

GROUP BY column_name

HAVING condition;
```

Key points:

- It always follows the GROUP BY clause.
- It filters groups of rows based on aggregate functions (like SUM, AVG, COUNT, MIN, MAX).
- You cannot use column names in the HAVING clause that are not included in the GROUP BY clause or an aggregate function.

Example:

Let's say we have an orders table with columns customer_id, order_amount. We want to find customers who have placed orders totaling more than $1000.

SQL

```
SELECT customer_id, SUM(order_amount) AS total_order_amount

FROM orders

GROUP BY customer_id

HAVING SUM(order_amount) > 1000;
```

Explanation:

1. GROUP BY customer_id: Groups the orders by customer.
2. SUM(order_amount): Calculates the total order amount for each customer.
3. HAVING SUM(order_amount) > 1000: Filters the groups where the total order amount is greater than $1000.

UPSERT in SQL

UPSERT is a term used to describe an operation that combines an INSERT and an UPDATE in a single statement. It's a way to insert new data into a table or update existing data based on a specific condition.

The syntax and availability of UPSERT vary across different SQL databases.

Common Approaches

1. **MERGE statement (SQL Server, Oracle):**
   - Combines INSERT and UPDATE into a single statement.
   - Provides flexibility in specifying conditions for both operations.
2. **INSERT ... ON DUPLICATE KEY UPDATE (MySQL):**
   - Inserts a new row if it doesn't exist.
   - Updates an existing row if a duplicate key is found.
3. **Using triggers:**
   - Creates a trigger to check for duplicate data and perform an update if necessary.

Example (MySQL)

```
INSERT INTO customers (customer_id, name, email)

VALUES (101, 'John Doe', 'johndoe@example.com')

ON DUPLICATE KEY UPDATE name = VALUES(name), email = VALUES(email);
```

Note: The specific syntax and capabilities of UPSERT vary across different databases.

Key points:

- UPSERT operations are often used for data synchronization or to avoid duplicate records.
- They can improve performance by reducing the number of round trips to the database.
- The choice of UPSERT method depends on the specific database system and the desired behavior.

Q.16 What is primary key in a table?

A primary key is a column (or a combination of columns) in a database table that uniquely identifies each row.

It ensures that every record in the table can be distinguished from every other record.

Key Characteristics of a Primary Key:

- Unique: Each value in the primary key must be different.
- Not Null: A primary key cannot contain null values.
- Index: The primary key is automatically indexed for efficient data retrieval.

Purpose of a Primary Key:

- Unique Identification: It serves as a unique identifier for each record.
- Data Integrity: It maintains data consistency and accuracy.
- Relationships: It's used to establish relationships between tables (foreign keys).
- Indexing: Improves query performance due to automatic indexing.

Example:

Consider a customers table with columns customer_id, name, and address. The customer_id column could be the primary key as it uniquely identifies each customer.

Q.17 What is foreign key in a table?

A foreign key is a column (or a set of columns) in one table that refers to the primary key of another table.

It establishes a link between the two tables, ensuring data integrity and relationships between them.

Key Characteristics of a Foreign Key:

- Referential Integrity: It maintains consistency between the data in two tables.
- Relationship: It defines a relationship between two tables (parent and child).
- Not Null (Optional): Unlike primary keys, foreign keys can be null.
- Multiple Foreign Keys: A table can have multiple foreign keys.

Purpose of a Foreign Key:

- Data Consistency: Ensures that data in the related tables is accurate and consistent.
- Referential Integrity: Prevents accidental deletion of data from the parent table that is referenced by the child table.
- Relationships: Defines how tables are connected and related.

Example:

Consider two tables:

- Customers (CustomerID, CustomerName)
- Orders (OrderID, CustomerID, OrderDate)

The CustomerID in the Orders table is a foreign key that references the CustomerID (primary key) in the Customers table. This ensures that every order is associated with a valid customer.

Q.18 what are types of Relationships supported by sql ?

SQL supports three primary types of relationships between tables:

1. One-to-One Relationship

- Definition: A single record in one table is associated with exactly one record in another table.

Example: A person and their passport.

- Implementation: Often achieved by making the primary key of one table a foreign key in the other.

2. One-to-Many Relationship

- Definition: One record in one table can be associated with multiple records in another table, but a record in the second table can only be associated with one record in the first table.
- Example: A customer and their orders. One customer can have many orders, but an order belongs to only one customer.
- Implementation: The primary key of the "one" side becomes a foreign key in the "many" side.

3. Many-to-Many Relationship

- Definition: Multiple records in one table can be associated with multiple records in another table.
- Example: Students and courses. A student can take many courses, and a course can have many students.
- Implementation: This relationship is typically implemented by creating a third table called a junction table. This table contains foreign keys from both original tables.

Q.19 Explain MongoDB Architecture in details

MongoDB, a NoSQL document-oriented database, stands out for its flexibility, scalability, and performance. Its architecture is designed to handle vast amounts of data efficiently while maintaining high availability. Let's delve into its key components:

Fundamental Concepts

- Documents: The basic unit of data in MongoDB, similar to rows in relational databases. They are JSON-like structures with key-value pairs.
- Collections: A group of documents, analogous to tables.
- Databases: Contain multiple collections.
- BSON: The binary-encoded format used for storing documents, offering faster processing than JSON.
- Indexes: Used to optimize query performance, similar to indexes in relational databases.

Architecture Overview

MongoDB's architecture comprises several layers:

Storage Engine

- MMAPv1: The original storage engine, storing BSON documents directly on disk.
- WiredTiger: The default and recommended storage engine, offering improved performance, compression, and features like ACID compliance.
- Other Storage Engines: MongoDB supports additional storage engines for specific use cases.

Indexing

MongoDB supports various index types:

- Single-field indexes: Create indexes on a single field for efficient querying.
- Compound indexes: Create indexes on multiple fields for complex queries.
- Text indexes: Enable full-text search capabilities.
- Geospatial indexes: Support efficient queries based on geographic location.

Query Processing

MongoDB's query processor optimizes query execution by:

- Using indexes to efficiently locate data.
- Executing query operations on the retrieved data.
- Returning results to the client.

Replication

MongoDB ensures data availability and durability through replication:

- Primary replica set member: Handles write operations and acts as the authoritative data source.
- Secondary replica set members: Maintain copies of the data and provide read capabilities.
- Replica set arbiter: Ensures data consistency and helps with election of a new primary.

Sharding

MongoDB scales horizontally using sharding:

- Sharded cluster: Distributes data across multiple servers called shards.
- Config servers: Manage the sharded cluster's metadata.
- Mongos: Act as the entry point for client applications, routing queries to the appropriate shards.

Key Features and Benefits

- Flexible Schema: Documents can have different structures, accommodating evolving data requirements.
- High Performance: Efficient query processing, indexing, and storage engine options.
- Scalability: Horizontal scaling through sharding for handling increasing data volumes.
- High Availability: Replication ensures data redundancy and fault tolerance.

- Rich Query Language: Supports complex queries, including aggregations and full-text search.

Q 20. Explain udf in SQL

A User-Defined Function (UDF) is a custom function written by a user to perform a specific task within a database. It encapsulates a block of code that can be reused multiple times, improving code readability, maintainability, and efficiency.

How UDFs Work

- **Input:** UDFs can accept zero or more input parameters.
- **Processing:** The function performs calculations, manipulations, or other operations based on the input parameters.
- **Output:** The function returns a single value (scalar function) or a table (table-valued function).

Types of UDFs

There are primarily two types of UDFs:

1. **Scalar UDFs:**
   - Return a single value based on the input parameters.
   - Example: A function to calculate the total price of an order based on quantity and unit price.
2. **Table-Valued UDFs:**
   - Return a table of data.
   - Example: A function to return a list of employees in a specific department.

Advantages of UDFs

- **Reusability:** Avoids code duplication.
- **Modularity:** Enhances code organization and maintainability.
- **Performance:** Can optimize complex calculations.
- **Abstraction:** Hides complex logic from the main query.

Example (SQL Server)

SQL

```
CREATE FUNCTION dbo.CalculateTotalPrice (@Quantity int, @UnitPrice decimal(10,2))

RETURNS decimal(10,2)

AS
```

```
BEGIN
  RETURN @Quantity * @UnitPrice;
END;
```

Q 21. What is Cartesian join?

Cartesian Join (Cross Join)

A Cartesian join, also known as a cross join, is a type of join in SQL that combines every row from one table with every row from another table. This results in a new table where the number of rows is equal to the product of the number of rows in the original tables.

How it works

Imagine you have two tables:

- **Table A:** With 3 rows
- **Table B:** With 4 rows

A Cartesian join between these tables would result in a new table with 3 * 4 = 12 rows. Each row in Table A will be combined with every row in Table B.

Syntax

The syntax for a Cartesian join is simple:

```
SELECT * FROM table1
CROSS JOIN table2;
```

Q.22 Explain Difference in aggregate functions vs window function in sql

Key Differences

Feature	Aggregate Functions	Window Functions
Output	Single value per group	Value for each row
Syntax	GROUP BY clause	OVER clause
Purpose	Summarize data	Compare values within a set
Examples	SUM, AVG, COUNT	RANK, DENSE_RANK, ROW_NUMBER

Q.23 Explain subquery and optimization techniques

Subquery Optimization

Subqueries can impact query performance, so it's essential to optimize them. Here are some techniques:

- **Index creation:** Create indexes on columns used in subquery conditions.
- **Avoid correlated subqueries:** If possible, rewrite correlated subqueries as joins or use temporary tables.
- **Limit subquery results:** Use LIMIT or TOP to restrict the number of rows returned by the subquery.
- **Consider derived tables:** For complex subqueries, create a derived table to improve performance.
- **Use EXISTS instead of IN:** In some cases, EXISTS can be more efficient than IN.
- **Leverage query hints:** Some database systems allow using query hints to influence the optimizer's choices.
- **Join optimization:** Use appropriate join types (inner, left, right, full outer) based on data relationships.
- **Aggregation optimization:** Use GROUP BY and HAVING clauses effectively.
- **Data partitioning:** If applicable, partition large tables to improve query performance.
- **Query rewriting:** Rewrite complex queries using simpler logic or different approaches.
- **Database-specific optimization:** Utilize database-specific features and optimization techniques.

Q.24 How will you sort table in sql?

The ORDER BY clause is the primary method for sorting data in SQL. It allows you to arrange the result set of a SELECT statement based on one or more columns.

Basic Syntax

SQL

SELECT column1, column2, ...

FROM table_name

ORDER BY column_name ASC | DESC;

Q.25 Scalar function in SQL

A **scalar function** in SQL operates on a single value and returns a single value as output. It's a built-in function that performs a specific operation on data.

Types of Scalar Functions

While there are many specific functions available depending on the database system (SQL Server, MySQL, Oracle, etc.), they generally fall into these categories:

- **Mathematical functions:** Perform calculations like ABS, SQRT, ROUND, CEILING, FLOOR, etc.
- **String functions:** Manipulate text data like UPPER, LOWER, CONCAT, SUBSTRING, LENGTH, etc.
- **Date and time functions:** Work with dates and times like DATE, TIME, NOW, DATEDIFF, etc.
- **Conversion functions:** Convert data types like CAST, CONVERT.

Using Scalar Functions in Queries

Scalar functions can be used in various parts of a SQL query:

- **SELECT clause:** To transform data before displaying it.
- **WHERE clause:** To filter data based on function results.
- **HAVING clause:** To filter groups based on function results.
- **ORDER BY clause:** To sort data based on function results.

Example:

SQL

```
SELECT customer_name, UPPER(city) AS city_upper

FROM customers

WHERE LEN(customer_name) > 10;
```

Q26. Differences Between SQL and NoSQL

Aspect	**SQL (Relational)**	**NoSQL (Non-relational)**
Data Structure	Tables with rows and columns	Document-based, key-value, column-family, or graph-based
Schema	Fixed schema (predefined structure)	Flexible schema (dynamic and adaptable)
Scalability	Vertically scalable (upgrading hardware)	Horizontally scalable (adding more servers)
Data Integrity	ACID-compliant (strong consistency)	BASE-compliant (more available, less consistent)
Query Language	SQL (Structured Query Language)	Varies (e.g., MongoDB uses its own query language)
Performance	Efficient for complex queries and transactions	Better for large-scale data and fast read/write operations

Use Case	Best for transactional systems (banking, ERP, etc.)	Ideal for big data, real-time web apps, and data lakes
Examples	MySQL, PostgreSQL, Oracle, MS SQL Server	MongoDB, Cassandra, CouchDB, Neo4j

Q.27 Datetime functions

Now()	To get current date and time	Now() 2024-09-16 20:02:59
Curdate()	To get todays' only date	Curdate() 2024-09-16
Date_format(date,format)	Converts the date in the given formatted string Y --- will display 4 digit year y will display 2 digit year M- month name in character	Date_format(curdate(),'%d/%m /%Y') 16/09/2024
	m-month in 2 digit number c- month in either 1 digit or 2 digit number d- date in number D- display th or st after date b--- display months in 3 letter (jan, feb,....) r ---- to print time in 12 hrs (hh:mm:ss AM/PM) T-----to print time in 24 hrs format W---- to show day of week (Sunday, Monday,....)	Date_format(cur date(),'%D , %M %Y') 3rd, November 2024

Date_add(date,interval n period)	It will find the date in future, you should specify the interval	date_add(curdat e(),interval 1 year)
		to find date after 1 year
		
		date_add(curdat e(),interval 15 day)
		to find date after 15 days

Date_sub(date,interval n period)	It will find the date in past, you should specify the interval	date_sub(curdat e(),interval 1 year)
		to find date 1 year before
		
		date_sub(curdat e(),interval 15 day)
		to find date 15 days before

Timestampdiff(period,date1,date2)	To find number of years/months or days difference	Timestampdiff(y ear,’2024-01-01’,’2027-01-01)
		3
		
		Timestampdiff(month,’2024-01-01’,’2027-01-01)
		36

Datediff(date1,date2)	It always given difference in days	Datediff(‘2024- 01-01’,’2024-12-31’) 366
Month(date)	Displays month in number	Month(‘2024- 02-22’) 2
Day(date)	Displays only date part of the given date	day(‘2024-02- 22’) 22
Year(date)	Displays year of the give date	year(‘2024-02- 22’) 2024
Week(date)	Displays week number of the year from the given date	week(‘2024-02- 22’) 7
Dayname(date)	Shows day as a string, Monday, Tuesday …..	dayname(‘2024- 02-22’) Thursday
quarter(date)	Shows quarter number of the year	quarter(‘2024- 03-02’) 1
Extract(portion from date)	To retrieve portion of the date	Extract (month from ‘2024-02-22’); 02 Select * from emp -> where extract(year from hiredate)=1981;
Last_day(date)	Gives last date of the month	Last_day(‘2024- 02 10’) 2024-02-29

Q.28 Acid properties with example

ACID property

1. Automicity--> Every transaction will be executed as a single unit, either all steps will

happen or None step will happen, is called as automicity

2. Consistency--> After every transaction the data in the database is always in correct state, is called consistency

3. Isolation--> When the transaction is in intermediate state, then the changes are visible only to the user, who is performing it.

And these changes will be visible to other users after the transaction is completed successfully,

And changes are committed.

4. Durability--> if every time transaction is in correct state, and if this happens for longer

duration, then it is called as durable

Q.29 Explain Views and its types.

Views

1. Views are like virtual tables
2. It helps us to give only access to required information of the table
3. It increases security by hiding table name
4. It also allows you to hide complexity of the queries

Types of views

1. Normal view	When you are creating normal view data will never get stored at separate place, every time you fire the query on views , the data will be retrieved every time from the base table by using base query.
2. Materialized view	When you are creating materialized view then the data will be stored in clients machine's ram for the current session, and after that every time you fire the query on views , the data will be retrieved from the clients RAM You may not get UpToDate data every time But these faster to access than normal views

Views are either simple views or complex views

If the base query in the view, uses joins, union, aggregate functions,

Then those views are called as complex views, otherwise called as simple views

Complex views are always read only

Simple views are by default readonly , if it does not contain all not null columns

1. Limited access to few columns or few rows from the table	Create view magr10 As Select * from emp Where deptno=10
2. It hides the table name	Create view mgr10 As Select empno,ename,price from emp Where deptno=10;
3. To hide the complexity of the queries	Create view myview as Select deptno,avg(sal+ifnull(comm,0)) avgsal, sum (sal) sumsal From emp Group by deptno Having count(*)> 2

Q.30 Explain Indexing

Indexes in

mysql Types of

indexs

Clustered Index	There is only one clustered index, it remains along with data, and keeps the data in the sorted order always, It does not require extra space Primary key
Non clustered Index	the other indexes on unique constraint or if user creates index, then all those are called as non-clustered index, but it needs extra space

Primary key and unique constraints Indexes on these columns will be automatically created

Why we use index

1. To increase the speed of searching by select query
2. So if any of the select query takes lot of time, then to improve the performance one of the choices is create index
3. But if you create indexes unnecessarily on multiple columns, then it will reduce the speed of DML operation, because every DML operation will need to update all the indices

To create index	create index idx_sal on emp(sal desc);
	create index idx_name on emp(ename,job);

To drop the index	Drop index <index name> on <table name> drop index my_dept_idx on emp;
To view all indexes on table	Show indexes from emp