SQL - Basics Interview Questions

(Practice Project)

PW SKILLS

Basic Questions:

1. What is the difference between SQL and MySQL?

SQL (Structured Query Language) is a standardized language used to communicate with and manipulate databases. MySQL, on the other hand, is an open-source relational database management system (RDBMS) that uses SQL to manage and query data. While SQL is the language, MySQL is a software application that implements SQL for database operations.

2. Write a query to create a new database named CompanyDB in MySQL.

CREATE DATABASE CompanyDB;

3. Explain the concept of an RDBMS. Why is it important in database management?

An RDBMS (Relational Database Management System) is a type of database management system that stores data in structured tables with relationships between them. It is important because it enforces data integrity, allows for efficient data retrieval through SQL, and provides mechanisms for managing relationships, ensuring that data is consistent and accurate.

4. Write a SQL query to show all the databases available in MySQL.

SHOW DATABASES;

5. What is the purpose of the CREATE command in SQL?

The CREATE command in SQL is used to define and create new database objects, such as databases, tables, indexes, or views. It specifies the structure of the object, including the data types and constraints for its columns.

6. Write a SQL query to create a table named Employees with the following columns: EmployeeID (INT), FirstName (VARCHAR), LastName (VARCHAR), HireDate (DATE).

```
CREATE TABLE Employees (
    EmployeeID INT,
    FirstName VARCHAR(100),
    LastName VARCHAR(100),
    HireDate DATE
);
```

7. Explain the difference between the ALTER and UPDATE commands.

The ALTER command is used to modify the structure of an existing database object, such as adding or removing columns from a table. The UPDATE command, on the other hand, is used to modify the data within existing records of a table.

8. Write a query to add a new column Salary (DECIMAL) to the Employees table.

ALTER TABLE Employees ADD Salary DECIMAL(10, 2);

9. What is the difference between TRUNCATE and DELETE commands?

TRUNCATE removes all rows from a table without logging individual row deletions, making it faster and less resource-intensive. DELETE removes rows one at a time and can include a WHERE clause to specify which rows to delete, allowing for selective removal.

10. Write a SQL query to delete the Employees table.
DROP TABLE Employees;

11. Describe the differences between CHAR, VARCHAR, and TEXT data types.
CHAR is a fixed-length string data type, meaning it always occupies the same amount of space regardless of the actual length of the string.

VARCHAR is a variable-length string data type that can store strings of varying lengths, saving space when the length of data is less than the defined maximum.

TEXT is used for larger strings (up to 65,535 characters) and is not limited by the maximum length of VARCHAR.

12. Write a query to create a table named Products with columns: ProductID (INT), ProductName (VARCHAR), Price (DECIMAL), ManufactureDate (DATE).

```
CREATE TABLE Products (
    ProductID INT,
    ProductName VARCHAR(100),
    Price DECIMAL(10, 2),
    ManufactureDate DATE
);
```

13. What are arithmetic operators in SQL? How are they used?
Arithmetic operators in SQL are used to perform mathematical calculations on numerical data. The basic arithmetic operators are + (addition), - (subtraction), * (multiplication), and / (division). They are commonly used in SELECT statements to calculate values based on columns.

14. Write a query to calculate the total price of all products from the Products table.
SELECT SUM(Price) AS TotalPrice FROM Products;

Intermediate Questions

15. Explain how comparison operators like = and <> are used in SQL.

Comparison operators in SQL are used to compare values in queries, allowing you to filter and retrieve data based on specific conditions. Here's how the = (equals) and <> (not equal) operators are used:

1. = Operator:
Purpose: The = operator checks if two values are equal. It is commonly used in the WHERE clause to filter records based on specific criteria.

Example Usage:
SELECT * FROM Employees WHERE FirstName = 'John';

This query retrieves all records from the Employees table where the FirstName is equal to 'John'.

2. <> Operator:
Purpose: The <> operator checks if two values are not equal. It is useful for excluding specific values in your results.

Example Usage:
SELECT * FROM Products WHERE Price <> 100.00;

This query retrieves all records from the Products table where the Price is not equal to 100.00.

Summary:
Comparison operators like = and <> are essential for filtering data in SQL queries, enabling you to specify exact matches or exclusions based on your requirements.

16. Write a SQL query to find all products with a price greater than 100 from the Products table.
SELECT * FROM Products WHERE Price > 100;
This query retrieves all columns for products in the Products table where the price exceeds 100.

17. What is a Primary Key constraint, and how does it enforce data integrity?
A Primary Key constraint is a rule applied to a database table that ensures each row has a unique identifier. Here are its key features and how it enforces data integrity:

Key Features:
1. **Uniqueness:** Each value in the primary key column(s) must be unique across the table, meaning no two rows can have the same primary key value.
2. **Non-nullability:** A primary key cannot contain NULL values; every row must have a valid primary key value.
3. **Single-column or Composite:** A primary key can consist of a single column (simple primary key) or multiple columns (composite primary key).

Enforcement of Data Integrity:
1. **Identification:** By providing a unique identifier for each row, primary keys help maintain the uniqueness of records, preventing duplicate entries.
2. **Referential Integrity:** Primary keys can be referenced by foreign keys in other tables, establishing relationships between tables and ensuring that related data remains consistent.
3. **Data Validation:** The constraints prevent invalid or incomplete data from being entered into the primary key column, ensuring that every record is identifiable and retrievable.

Example:
In a table named Employees, if EmployeeID is designated as the primary key, each employee must have a unique EmployeeID, ensuring that no two employees can have the same identifier and that every employee record can be uniquely identified.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(100),
    LastName VARCHAR(100)
);
```

In this example, the EmployeeID column serves as the primary key, enforcing uniqueness and non-nullability, thus maintaining data integrity within the Employees table.

18. Write a query to create a Customers table with CustomerID as the Primary Key and Email as a Unique Key.

Here's a SQL query to create a Customers table with CustomerID as the Primary Key

```
and Email as a Unique Key:
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Email VARCHAR(255) UNIQUE,
    FirstName VARCHAR(100),
    LastName VARCHAR(100)
);
```

In this query:

- CustomerID is defined as the primary key, ensuring that each customer has a unique identifier.
- Email is defined as a unique key, ensuring that no two customers can have the same email address.

19. Explain the purpose of the Foreign Key constraint.

The **Foreign Key constraint** is a fundamental aspect of relational database design that establishes a relationship between two tables. Here are its key purposes:

1. **Establish Relationships:** A foreign key in one table refers to the primary key in another table, creating a link between the two. This relationship is crucial for maintaining the integrity and organization of data.

2. **Enforce Referential Integrity:** The foreign key constraint ensures that the value in the foreign key column must either be NULL or match an existing value in the referenced primary key column. This prevents orphaned records and maintains consistency across related tables.

3. **Support Data Integrity:** By enforcing relationships, foreign keys help prevent invalid data entries. For example, if you have an Orders table that references a Customers table, a foreign key constraint would prevent the insertion of an order for a non-existent customer.

4. **Facilitate Cascading Actions:** Foreign keys can be configured to perform cascading actions (like CASCADE, SET NULL, or SET DEFAULT) when a referenced record is updated or deleted. This ensures that changes in one table automatically propagate to related tables, maintaining data consistency.

Example:

Consider two tables, Customers and Orders. The Orders table has a foreign key that references the CustomerID in the Customers table. This relationship ensures that every order is associated with a valid customer.

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES
Customers(CustomerID)
);
```

In this example, the CustomerID in the Orders table is a foreign key that references the CustomerID in the Customers table, enforcing referential integrity and ensuring that every order is linked to a valid customer.

20. Write a query to create a Foreign Key relationship between Orders and Customers tables, linking CustomerID.

Here's a SQL query to create a Foreign Key relationship between the Orders and **Customers tables, linking the CustomerID:**

```
ALTER TABLE Orders
ADD CONSTRAINT FK_Customer
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID);
```

In this query:

- The **ALTER TABLE** statement is used to modify the Orders table.
- The **ADD CONSTRAINT** clause creates a foreign key constraint named FK_Customer.
- The **FOREIGN KEY (CustomerID)** specifies that the **CustomerID** column in the **Orders** table is the foreign key.
- The **REFERENCES Customers(CustomerID)** indicates that this foreign key references the **CustomerID** column in the **Customers** table.

This establishes a relationship where each order in the Orders table is linked to a valid customer in the **Customers** table.

21. What is the Not Null constraint, and why is it important?

The Not Null constraint is a rule applied to a database column that ensures that the column cannot contain NULL values. Here are its key features and importance:

Key Features:

1. **Non-nullability:** When a column is defined with the Not Null constraint, it must have a valid value for every row in the table; leaving it empty (NULL) is not permitted.
2. **Enforcement at Data Entry:** The database management system (DBMS) enforces this constraint during data insertion or updates, preventing the entry of NULL values.

Importance:

- **Data Integrity:** The Not Null constraint helps maintain data integrity by ensuring that essential fields contain valid data. For example, in a Customers table, the Email field may be marked as Not Null to ensure that every customer has an email address recorded.
- **Consistency:** It promotes consistency within the database by ensuring that all necessary information is provided, reducing the likelihood of incomplete records.
- **Improved Query Performance:** Queries can be more efficient because the database engine does not need to account for NULL values, which can simplify the logic for conditions in queries.

Example:

When creating a table, you might use the Not Null constraint like this:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Email VARCHAR(255) NOT NULL
);
```

In this example, both the Name and Email columns are defined with the Not Null constraint, ensuring that every customer record must include both a name and an email address, thereby enhancing data integrity and consistency within the table.

22. Write a query to modify the Customers table to make the Email column Not Null.

Here's a SQL query to modify the Customers table to make the Email column Not Null:

```
ALTER TABLE Customers
MODIFY Email VARCHAR(255) NOT NULL;
```

In this query:

- The ALTER TABLE statement is used to change the structure of the Customers table.
- The MODIFY clause specifies that the Email column should be altered to enforce the Not Null constraint, ensuring that it cannot contain NULL values.

Note: Before running this query, ensure that there are no existing NULL values in the Email column; otherwise, the query will fail. You may need to update or remove any rows with NULL values before applying this change.

23. What is an INNER JOIN, and when would you use it?

An INNER JOIN is a SQL operation that combines rows from two or more tables based on a related column between them. It returns only the rows where there is a match in both tables.

Key Features:

1. **Matching Rows:** Returns records with matching values in specified columns.
2. **Excludes Non-matching Rows:** Rows without matches in either table are not included.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

Example:

To retrieve customers and their orders:

```
SELECT Customers.CustomerID, Customers.Name, Orders.OrderID
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

When to Use:

- **To Retrieve Related Data:** Use INNER JOIN when you need to get related data from multiple tables.
- **Filtering Results:** Ideal for including only complete records with existing relationships.

24. Write a query to retrieve a list of all employees and their respective department names using an INNER JOIN.

Here's a SQL query to retrieve a list of all employees and their respective department names using an INNER JOIN:

```
SELECT Employees.EmployeeID, Employees.FirstName, Employees.LastName,
Departments.DepartmentName

FROM Employees

INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

In this query:

- The **Employees** table is joined with the Departments table based on the **DepartmentID**.
- It retrieves the **EmployeeID, FirstName, LastName** from the **Employees** table, and the **DepartmentName** from the **Departments** table.

25. Explain the difference between UNION and UNION ALL.

The **UNION** and **UNION ALL** operators in SQL are used to combine the results of two or more SELECT queries, but they have key differences:

UNION:

Duplicate Removal: The UNION operator removes duplicate rows from the result set. If the same row appears in both queries, it will only appear once in the final output.

Performance: Because UNION checks for duplicates, it may be slower than UNION ALL, especially with large datasets.

Example:

```
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

This query returns unique values from both table1 and table2.

UNION ALL:

Includes Duplicates: The UNION ALL operator includes all rows from the combined result set, including duplicates. If the same row appears in both queries, it will be included in the final output multiple times.

Performance: UNION ALL is generally faster than UNION since it does not perform duplicate checking.

Example:

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

This query returns all values from both table1 and table2, including duplicates.

Summary:
UNION removes duplicates; **UNION ALL** includes all records, allowing duplicates. Use UNION when you need distinct results, and use UNION ALL when you want all records without the overhead of duplicate checking.

26. Write a query to combine the results of two queries: one that selects ProductName from the Products table and another that selects ServiceName from a Services table, using UNION.

Here's a SQL query that combines the results of two queries—one selecting ProductName from the Products table and the other selecting ServiceName from the Services table—using UNION:

```
SELECT ProductName FROM Products
UNION
SELECT ServiceName FROM Services;
```

This query retrieves a list of unique names from both the Products and Services tables, ensuring that any duplicate names are included only once in the final result set.

Advanced Questions

27. What are subqueries, and how are they different from JOINs?

Subqueries are nested SQL queries embedded within another SQL query, used to perform operations that require multiple steps. They can return a single value, a single row, or multiple rows and are executed before the outer query.

Example of a Subquery:

```
SELECT EmployeeID, FirstName
FROM Employees
WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'Sales');
```

Difference from JOINs:

1. **Purpose:**
   - **Subqueries:** Used to filter results based on aggregated data or complex conditions.
   - **JOINs:** Combine rows from two or more tables based on a related column.

2. **Performance:**
- **Subqueries:** Can be less efficient, especially if returning large datasets.
- **JOINs:** Typically faster for combining related data.

Example of a JOIN:

```
SELECT Employees.EmployeeID, Employees.FirstName, Departments.DepartmentName
FROM Employees
INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

Summary:
Subqueries are for filtering based on their results, while JOINs combine data from related tables. Each serves a different purpose and can be chosen based on specific query requirements.

28. Write a SQL query using a subquery to find all employees who earn more than the average salary in their department.

Here's a SQL query using a subquery to find all employees who earn more than the average salary in their respective departments:

```
SELECT EmployeeID, FirstName, LastName, Salary
FROM Employees e
WHERE Salary > (
  SELECT AVG(Salary)
  FROM Employees
  WHERE DepartmentID = e.DepartmentID
);
```

In this query:
- The outer query retrieves employees' EmployeeID, FirstName, LastName, and Salary from the Employees table.
- The subquery calculates the average salary for each employee's department by matching the DepartmentID.
- The outer query filters employees whose salary exceeds the average salary of their respective departments.

29. What is a correlated subquery?

A correlated subquery is a type of subquery that depends on the outer query for its values. Unlike a regular subquery, it cannot be executed independently and is evaluated once for each row processed by the outer query.

Key Features:
1. **References Outer Query:** It references one or more columns from the outer query.
2. **Multiple Executions:** Executed repeatedly for each row in the outer query, which can affect performance.

Example:

To find employees whose salary is higher than the average salary of their respective departments:

```
SELECT EmployeeID, FirstName, LastName, Salary
FROM Employees e1
WHERE Salary > (
  SELECT AVG(Salary)
  FROM Employees e2
  WHERE e1.DepartmentID = e2.DepartmentID
);
```

Summary:
Correlated subqueries are used for filtering data based on conditions linked to the outer query's results but may be less efficient due to their multiple executions.

30. Write a correlated subquery to find all products that are priced above the average price in their respective categories.

Here's a SQL query using a correlated subquery to find all products that are priced above the average price in their respective categories:

```
SELECT ProductID, ProductName, Price, CategoryID
FROM Products p1
WHERE Price > (
  SELECT AVG(Price)
  FROM Products p2
  WHERE p1.CategoryID = p2.CategoryID
);
```

Explanation:
- The outer query selects ProductID, ProductName, Price, and CategoryID from the Products table (aliased as p1).
- The correlated subquery calculates the average price of products within the same category as the current product (p1.CategoryID = p2.CategoryID).
- The outer query filters products whose price exceeds the average price of their respective categories.