SQL Interview Questions and Answers

1. What is a SQL JOIN? Explain different types of JOINS.

Definition of SQL Join:

A `JOIN` in SQL is used to combine rows from two or more tables based on a related column between them. It allows you to retrieve and work with data that is distributed across multiple tables. `JOIN` is essential for querying relational databases where data is normalized into multiple tables.

Types of SQL Joins:

1. `INNER JOIN`
    - **Definition**: An `INNER JOIN` returns only the rows where there is a match in both tables based on the specified condition. If no match is found, those rows are excluded from the result.
    - **Example**:

      ```
      SELECT orders.order_id, customers.customer_name
      FROM orders
      INNER JOIN customers ON orders.customer_id = customers.customer_id;
      ```

2. **`LEFT JOIN` (or `LEFT OUTER JOIN`)**
    - **Definition**: A `LEFT JOIN` returns all rows from the left table and the matching rows from the right table. If no match is found, `NULL` values are returned for columns from the right table.
    - **Example**:

      ```
      SELECT orders.order_id, customers.customer_name
      FROM orders
      LEFT JOIN customers ON orders.customer_id = customers.customer_id;
      ```

3. **`RIGHT JOIN` (or `RIGHT OUTER JOIN`)**
    - **Definition**: A `RIGHT JOIN` returns all rows from the right table and the matching rows from the left table. If no match is found, `NULL` values are returned for columns from the left table.
    - **Example**:

      ```
      SELECT orders.order_id, customers.customer_name
      FROM orders
      RIGHT JOIN customers ON orders.customer_id = customers.customer_id;
      ```

4. **`FULL JOIN` (or `FULL OUTER JOIN`)**
    - **Definition**: A `FULL JOIN` returns all rows from both tables, and where there is no match, `NULL` values are returned for the columns of the table without a match.
    - **Example**:

```
SELECT orders.order_id, customers.customer_name
FROM orders
FULL OUTER JOIN customers ON orders.customer_id =
customers.customer_id;
```

5. `CROSS JOIN`

   - **Definition**: A `CROSS JOIN` returns the Cartesian product of both tables. This means it combines every row from the first table with every row from the second table.
   - **Example**:

     ```
     SELECT products.product_name, categories.category_name
     FROM products
     CROSS JOIN categories;
     ```

6. `SELF JOIN`

   - **Definition**: A `SELF JOIN` is a regular `JOIN` but the table is joined with itself. This can be useful for comparing rows within the same table.
   - **Example**:

     ```
     SELECT A.employee_name, B.employee_name AS manager_name
     FROM employees A
     INNER JOIN employees B ON A.manager_id = B.employee_id;
     ```

7. `NATURAL JOIN`

   - **Definition**: A `NATURAL JOIN` automatically joins tables based on columns with the same name and compatible data types in both tables.
   - **Example**:

     ```
     SELECT orders.order_id, customers.customer_name
     FROM orders
     NATURAL JOIN customers;
     ```

---

2. What is a PRIMARY KEY?

Definition:

A `PRIMARY KEY` is a field (or a combination of fields) that uniquely identifies each record in a table. It must contain unique values and cannot contain NULLs.

Example:

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(100)
);
```

---

3. What is the difference between DELETE, TRUNCATE, and DROP?

Answer:

- **DELETE**: Removes rows from a table based on the `WHERE` condition. It can be rolled back.
- **TRUNCATE**: Removes all rows from a table but does not log individual row deletions. It is faster than `DELETE` but cannot be rolled back (in most RDBMS).
- **DROP**: Removes the table structure itself, along with all data, indexes, and constraints.

Example:

```
DELETE FROM customers WHERE customer_id = 1;
TRUNCATE TABLE customers;
DROP TABLE customers;
```

---

4. What is the difference between UNION and UNION ALL?

Answer:

- **UNION**: Combines the results of two or more `SELECT` queries and removes duplicates.
- **UNION ALL**: Combines the results of two or more `SELECT` queries but does not remove duplicates.

Example:

```
SELECT customer_name FROM customers
UNION
SELECT customer_name FROM employees;

SELECT customer_name FROM customers
UNION ALL
SELECT customer_name FROM employees;
```

---

5. What is a Foreign Key?

Definition:

A `Foreign Key` is a column (or a group of columns) in a table that links to the primary key in another table, ensuring referential integrity.

Example:

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

---

6. What is normalization?

Definition:

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. The common normal forms are:

- **1NF (First Normal Form)**: Each column contains atomic values, and each record is unique.
- **2NF (Second Normal Form)**: Meets 1NF, and all non-key attributes are fully functionally dependent on the primary key.
- **3NF (Third Normal Form)**: Meets 2NF, and all attributes are functionally dependent on the key, the whole key, and nothing but the key.

---

7. What is the difference between HAVING and WHERE clauses?

Answer:

- **WHERE**: Filters rows before the aggregation takes place.
- **HAVING**: Filters rows after the aggregation has been performed (used with `GROUP BY`).

Example:

```
SELECT COUNT(*), customer_id
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 10;  -- This filters the result of the GROUP BY
```

---

8. What are aggregate functions in SQL?

Answer:

Aggregate functions perform a calculation on a set of values and return a single value. Common aggregate functions include:

- **COUNT()**: Returns the number of rows.
- **SUM()**: Returns the sum of a numeric column.
- **AVG()**: Returns the average value of a numeric column.
- **MIN()**: Returns the minimum value.
- **MAX()**: Returns the maximum value.

Example:

```
SELECT AVG(salary) FROM employees;
```

4. What is the difference between WHERE and HAVING clauses in SQL?

- **`WHERE` Clause**: It is used to filter rows before any groupings are made. It operates on individual rows and is used with non-aggregated data.

- **`HAVING` Clause**: It is used to filter groups created by the `GROUP BY` clause. It operates on aggregated data.

Example:

```
SELECT department, COUNT(employee_id)
FROM employees
WHERE salary > 50000
GROUP BY department
HAVING COUNT(employee_id) > 5;
```

This query selects departments where employees' salaries are greater than 50,000 and there are more than 5 employees in the department.

---

9. What is indexing in SQL?

Answer:

An index is a data structure used to improve the speed of data retrieval operations on a database table. Indexes can speed up query execution but can also slow down data modification operations (like `INSERT`, `UPDATE`, and `DELETE`).

An index in SQL is a database object used to speed up the retrieval of rows from a table. It works like a lookup table that provides quick access to data without scanning the entire table.

Example:

```
CREATE INDEX idx_customer_name ON customers(customer_name);
```

---

10. What is the GROUP BY clause in SQL?

Definition:

-- The `GROUP BY` clause is used to group rows that have the same values into summary rows, often with aggregate functions like `COUNT`, `SUM`, `AVG`, etc.

-- The `GROUP BY` statement in SQL is used to arrange identical data into groups. It is often used with aggregate functions (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`) to perform operations on each group of data.

Example:

```
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id;

SELECT department, COUNT(employee_id) FROM employees GROUP BY department;
```

---

11. What is the `JOIN` condition and how does it work in SQL?

Definition:

A `JOIN` condition is the criteria used to match rows from two or more tables in SQL. The most common condition is the equality condition, where a column from one table is compared with a column from another table.

Example:

```
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

This query joins the `orders` table and the `customers` table based on the `customer_id` column.

---

12. What is the purpose of `DISTINCT` keyword in SQL?

Definition:

The `DISTINCT` keyword is used to remove duplicate records from the result set. It ensures that each value in the result set is unique.

Example:

```
SELECT DISTINCT department FROM employees;
```

This query retrieves all unique departments from the `employees` table.

---

5. What is a `UNION` operator in SQL?

Definition:

The `UNION` operator is used to combine the results of two or more `SELECT` statements. It removes duplicate records by default. The columns in each `SELECT` statement must be the same and in the same order.

Example:

```
SELECT product_name FROM products_2023
UNION
SELECT product_name FROM products_2024;
```

This query combines product names from two tables and removes duplicates.

6. What is a SUBQUERY in SQL?

Definition:

A SUBQUERY is a query nested inside another query. It is used to retrieve a value or set of values to be used in the outer query.

Example:

```
SELECT product_name
FROM products
WHERE product_id = (SELECT product_id FROM orders WHERE order_id = 100);
```

This query retrieves the product name associated with a specific order.

2. ACID Properties

The ACID properties define the characteristics that ensure reliable transactions in a relational database management system (RDBMS). These properties are crucial to maintaining the integrity of the database and ensuring that transactions are processed reliably.

ACID stands for:

1. **Atomicity**
    - **Definition**: Atomicity ensures that a transaction is treated as a single unit, which either completely succeeds or completely fails. If any part of the transaction fails, the entire transaction is rolled back, ensuring no partial updates are made.
    - **Example**: If you transfer money between two bank accounts, both the withdrawal and deposit must succeed; if either fails, the entire transaction fails.
2. **Consistency**
    - **Definition**: Consistency ensures that a transaction brings the database from one valid state to another. The database must always remain in a consistent state, adhering to predefined rules and constraints (like primary keys, foreign keys, etc.).
    - **Example**: If a transaction involves adding a new record to the database, the data must meet the integrity constraints, such as valid values or relationships with other tables.
3. **Isolation**
    - **Definition**: Isolation ensures that the operations of one transaction are isolated from others. Even if multiple transactions are happening concurrently, the results of each transaction should not be visible to other transactions until they are committed. This prevents dirty reads, non-repeatable reads, and phantom reads.
    - **Example**: If two transactions are updating the same bank account balance simultaneously, isolation ensures that one transaction's changes are not seen by the other until it's fully completed.
4. **Durability**

- **Definition**: Durability ensures that once a transaction has been committed, it will remain persistent in the database, even in the event of a system crash or failure. The database will ensure that no data is lost after the transaction is committed.
- **Example**: If you update a customer's address in the database and the system crashes immediately after the transaction commits, the new address will still be present once the system is restored.

---

3. Example of Using ACID Principles:

Let’s say we are transferring money between two bank accounts:

- **Step 1**: **Atomicity** - If the system crashes after deducting money from the sender’s account but before crediting the recipient’s account, the system will roll back the deduction, ensuring no partial transaction occurs.
- **Step 2**: **Consistency** - The system ensures that after the transaction, both the sender’s and recipient’s balances reflect accurate data, respecting any constraints such as account numbers or balance limits.
- **Step 3**: **Isolation** - If another transaction is happening simultaneously, the withdrawal or deposit from the first transaction will not affect the other until it’s fully completed.
- **Step 4**: **Durability** - Once the transaction is committed, even if the system crashes, the changes (withdrawal and deposit) will not be lost and will persist in the database.

---

4. Optimizing SQL Joins for Performance

- **Use indexed columns**: Ensure that columns used in `JOIN` conditions are indexed to speed up lookups.
- **Limit the result set**: Use `WHERE` clauses to filter unnecessary rows before performing the `JOIN`.
- **Avoid `CROSS JOIN` unless needed**: This type of join can produce a very large result set, so avoid it unless necessary.
- **Use `EXPLAIN` for query optimization**: The `EXPLAIN` command helps you understand how the database executes your query, allowing you to optimize it further.

1. SELECT Query

The `SELECT` statement is used to retrieve data from a database.

Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

Example:

```
SELECT customer_name, city FROM customers;
```

---

2. SELECT DISTINCT

The `DISTINCT` keyword is used to return only unique (non-duplicate) values.

Syntax:

```
SELECT DISTINCT column_name
FROM table_name;
```

Example:

```
SELECT DISTINCT city FROM customers;
```

---

3. WHERE Clause

The `WHERE` clause is used to filter records based on a specified condition.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
SELECT * FROM customers
WHERE city = 'New York';
```

---

4. AND, OR, NOT Operators

These logical operators are used to filter records based on more than one condition.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2;
```

Example:

```
SELECT * FROM customers
WHERE city = 'New York' AND age > 25;
```

---

5. ORDER BY Clause

The `ORDER BY` clause is used to sort the result set in either ascending (`ASC`) or descending (`DESC`) order.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC | DESC];
```

Example:

```
SELECT * FROM customers
ORDER BY customer_name ASC;
```

---

6. INSERT INTO Query

The `INSERT INTO` statement is used to insert new records into a table.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO customers (customer_name, city, age)
VALUES ('John Doe', 'Los Angeles', 30);
```

---

7. UPDATE Query

The `UPDATE` statement is used to modify existing records in a table.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

```
UPDATE customers
SET age = 31
WHERE customer_name = 'John Doe';
```

---

8. DELETE Query

The `DELETE` statement is used to remove existing records from a table.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:

```
DELETE FROM customers
WHERE customer_name = 'John Doe';
```

---

9. TRUNCATE Query

The TRUNCATE statement is used to remove all rows from a table, but the structure of the table remains intact.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE customers;
```

---

10. JOIN Operations

INNER JOIN

Combines rows from both tables where there is a match in both tables.

Syntax:

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

Example:

```
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

LEFT JOIN

Returns all records from the left table and matched records from the right table.

Syntax:

```
SELECT column1, column2, ...
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Example:

```
SELECT orders.order_id, customers.customer_name
FROM orders
LEFT JOIN customers ON orders.customer_id = customers.customer_id;
```

---

11. GROUP BY Clause

The `GROUP BY` statement is used to arrange identical data into groups, typically with aggregate functions like `COUNT`, `SUM`, etc.

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name;
```

Example:

```
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id;
```

---

12. HAVING Clause

The `HAVING` clause is used to filter groups based on a condition (similar to `WHERE`, but applied after the `GROUP BY`).

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name
HAVING aggregate_function(column_name) condition;
```

Example:

```
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 10;
```

---

13. ALTER TABLE Query

The `ALTER TABLE` statement is used to modify an existing table structure, such as adding or dropping columns.

Syntax (Add Column):

```
ALTER TABLE table_name
ADD column_name column_definition;
```

Syntax (Drop Column):

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example:

```
ALTER TABLE customers
ADD phone_number VARCHAR(15);
```

---

14. CREATE TABLE Query

The `CREATE TABLE` statement is used to create a new table in the database.

Syntax:

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
);
```

Example:

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(100),
  city VARCHAR(50),
  age INT
);
```

---

15. DROP TABLE Query

The `DROP TABLE` statement is used to remove a table and all of its data from the database.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE customers;
```

---

16. INDEX Query

The `CREATE INDEX` statement is used to create an index on one or more columns of a table. Indexes improve the speed of data retrieval.

Syntax:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Example:

```
CREATE INDEX idx_customer_name
ON customers (customer_name);
```

---

17. DISTINCT Query with Aggregates

The `DISTINCT` keyword can also be used with aggregate functions.

Example:

```
SELECT COUNT(DISTINCT city)
FROM customers;
```

---

18. Subquery (Nested SELECT)

A subquery is a query within another query, often used in `WHERE`, `FROM`, or `SELECT` clauses.

Example:

```
SELECT customer_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_date > '2025-
01-01');
```

---

19. CASE Statement

The `CASE` statement is used to create conditional logic inside SQL queries.

Syntax:

```
SELECT column1,
       CASE
           WHEN condition THEN result
           ELSE default_result
       END AS alias_name
FROM table_name;
```

Example:

```
SELECT order_id,
       CASE
           WHEN order_amount > 100 THEN 'Large'
           ELSE 'Small'
       END AS order_size
FROM orders;
```