**Interview Questions:**

**SQL :**

1. What are the ACID properties in SQL? Explain each one briefly.

Solution-

**Atomicity**: This property ensures that each transaction is treated as a single unit, which either succeeds completely or fails completely.

**Consistency**: This property ensures that the database remains in a consistent state before and after the execution of a transaction

**Isolation**: Isolation ensures that the execution of multiple transactions concurrently does not result in any interference between them

**Durability** ensures that once a transaction is committed, its effects are permanent, even in the event of a system failure such as a power outage or crash

1. Differentiate between DDL, DML, DCL, and TCL commands in SQL.
2. Solution- **DDL (Data Definition Language)**:
   * DDL commands are used to define, modify, and manage the structure of database objects.
   * Examples include:
     + **CREATE**: Used to create new database objects such as tables, views, indexes, etc.
     + **ALTER**: Used to modify the structure of existing database objects.
     + **DROP**: Used to delete or remove existing database objects.
     + **TRUNCATE**: Used to remove all records from a table, but not the table structure itself.
     + **RENAME**: Used to rename an existing database object.
3. **DML (Data Manipulation Language)**:
   * DML commands are used to manipulate data stored in the database.
   * Examples include:
     + **INSERT**: Used to add new records or rows to a table.
     + **SELECT**: Used to retrieve data from one or more tables.
     + **UPDATE**: Used to modify existing records in a table.
     + **DELETE**: Used to remove records from a table.
4. **DCL (Data Control Language)**:
   * DCL commands are used to control access to data within the database.
   * Examples include:
     + **GRANT**: Used to give specific privileges to users or roles.
     + **REVOKE**: Used to remove specific privileges from users or roles.
5. **TCL (Transaction Control Language)**:
   * TCL commands are used to manage transactions within the database.
   * Examples include:
     + **COMMIT**: Used to permanently save the changes made in a transaction.
     + **ROLLBACK**: Used to undo the changes made in a transaction.
     + **SAVEPOINT**: Used to set a point within a transaction to which you can later roll back.
     + **SET TRANSACTION**: Used to set characteristics for the transaction, such as isolation level.
6. Explain the concept of database normalization and its importance.

Solution- Database normalization is the process of organizing the attributes and tables of a relational database to minimize redundancy and dependency. It involves splitting large tables into smaller ones and defining relationships between them, thereby ensuring data integrity and optimizing database performance. The normalization process typically follows a set of rules or normal forms, with each successive normal form building upon the previous one.

1. What are the types of indexes in SQL? Provide examples of when to use each type.

Solution- In SQL, there are several types of indexes, each designed to optimize query performance for specific scenarios. Here are some common types of indexes along with examples of when to use each type:

1. **Primary Index**:
   * A primary index is created on a table's primary key column(s). It enforces uniqueness and provides fast access to rows based on their primary key values.
   * Example: Use a primary index on the **id** column of a **users** table to quickly retrieve user information based on their unique identifier.
2. **Unique Index**:
   * A unique index ensures that the values in the indexed column(s) are unique across the table. It provides fast access for queries that need to enforce uniqueness.
   * Example: Create a unique index on the **email** column of a **users** table to ensure that each user has a unique email address.
3. **Clustered Index**:
   * A clustered index determines the physical order of rows in the table based on the indexed column(s). Each table can have only one clustered index.
   * Example: Use a clustered index on the **date** column of a **transactions** table to physically order transactions by date, facilitating efficient range queries.
4. **Non-Clustered Index**:
   * A non-clustered index creates a separate structure that contains pointers to the actual rows in the table. It is used to speed up queries that don't match the clustered index order.
   * Example: Create a non-clustered index on the **product\_id** column of an **order\_items** table to quickly retrieve order items based on their product IDs.
5. **Composite Index**:
   * A composite index is created on multiple columns. It can improve query performance for queries that involve filtering or sorting on multiple columns.
   * Example: Create a composite index on the **(last\_name, first\_name)** columns of an **employees** table to speed up queries that involve searching for employees by their last name and first name together.
6. **Covering Index**:
   * A covering index includes all the columns needed to satisfy a query in the index itself, eliminating the need to access the actual table data.
   * Example: Create a covering index on **(customer\_id, order\_date)** columns of an **orders** table if queries frequently filter by **customer\_id** and retrieve **order\_date**, thus avoiding accessing the main table.
7. **Bitmap Index**:
   * A bitmap index is used for columns with low cardinality, where each value appears frequently. It stores a bitmap for each distinct value, indicating which rows contain that value.
   * Example: Use a bitmap index on a **gender** column with values like 'Male' and 'Female' in a large dataset where gender distribution is relatively balanced.
8. Describe the Entity-Relationship Model in SQL. What are entities, attributes, and relations?

Solution- The Entity-Relationship (ER) model is a conceptual data model used to describe the relationships between entities in a database. It provides a graphical representation of the database schema, showing the entities, attributes, and relationships among them

1. Discuss various data types in SQL and provide examples of each.
2. Solution- **Numeric Data Types**:
   * **INT**: Used to store whole numbers (integers) within a specified range. Example: **age INT**
   * **DECIMAL/NUMERIC**: Used to store fixed-point numbers with a specified precision and scale. Example: **price DECIMAL(10, 2)**
   * **FLOAT/REAL**: Used to store floating-point numbers with a specified precision. Example: **temperature FLOAT**
   * **BIT/BOOLEAN**: Used to store boolean values (true/false). Example: **is\_active BIT**
3. **Character Data Types**:
   * **CHAR(n)**: Fixed-length character string with a maximum length of 'n'. Example: **name CHAR(50)**
   * **VARCHAR(n)**: Variable-length character string with a maximum length of 'n'. Example: **address VARCHAR(100)**
   * **TEXT**: Variable-length character string with a maximum length greater than that of VARCHAR. Example: **description TEXT**
4. **Date and Time Data Types**:
   * **DATE**: Used to store date values in the format 'YYYY-MM-DD'. Example: **birthdate DATE**
   * **TIME**: Used to store time values in the format 'HH:MM:SS'. Example: **appointment\_time TIME**
   * **DATETIME**: Used to store date and time values in the format 'YYYY-MM-DD HH:MM:SS'. Example: **created\_at DATETIME**
   * **TIMESTAMP**: Similar to DATETIME but stores the timestamp in UTC timezone. Example: **last\_updated TIMESTAMP**
5. **Binary Data Types**:
   * **BLOB**: Used to store large binary objects such as images, audio, or video files. Example: **image\_data BLOB**
   * **BINARY(n)**: Fixed-length binary string with a maximum length of 'n'. Example: **binary\_data BINARY(64)**
   * **VARBINARY(n)**: Variable-length binary string with a maximum length of 'n'. Example: **binary\_data VARBINARY(256)**
6. **Other Data Types**:
   * **ENUM**: Used to specify a set of predefined values for a column. Example: **gender ENUM('Male', 'Female', 'Other')**
   * **JSON**: Used to store JSON data. Example: **user\_data JSON**

7. Explain the purpose of SQL set operations. Provide examples of UNION, UNION ALL, INTERSECT, and MINUS operations.

Solution- SELECT column1 FROM table1

UNION

SELECT column1 FROM table2;

SELECT column1 FROM table1

UNION ALL

SELECT column1 FROM table2;

SELECT column1 FROM table1

INTERSECT

SELECT column1 FROM table2;

SELECT column1 FROM table1

MINUS

SELECT column1 FROM table2;

8. What are the different types of SQL joins? Provide scenarios where each type would be useful.

1. Solution- **INNER JOIN**:
   * An INNER JOIN returns rows from both tables where there is a match based on the join condition.
   * Example scenario: Suppose you have two tables, **orders** and **customers**, and you want to retrieve information about orders along with the corresponding customer details. An INNER JOIN would be appropriate here to fetch only the orders that have associated customers.
2. **LEFT JOIN (or LEFT OUTER JOIN)**:
   * A LEFT JOIN returns all rows from the left table (the first table mentioned in the query) and the matching rows from the right table based on the join condition. If there is no match, NULL values are returned for columns from the right table.
   * Example scenario: In a similar scenario to the previous example, if you want to retrieve information about all orders along with the corresponding customer details (if available), including orders without associated customers, a LEFT JOIN would be suitable. This ensures that all orders are included in the result, even if there is no matching customer.
3. **RIGHT JOIN (or RIGHT OUTER JOIN)**:
   * A RIGHT JOIN is similar to a LEFT JOIN but returns all rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for columns from the left table.
   * Example scenario: If you want to retrieve information about all customers along with their corresponding orders (if available), including customers who haven't placed any orders yet, a RIGHT JOIN would be appropriate. This ensures that all customers are included in the result, even if there is no matching order.
4. **FULL JOIN (or FULL OUTER JOIN)**:
   * A FULL JOIN returns all rows from both tables, matching rows where possible and including NULL values where there is no match.
   * Example scenario: If you want to retrieve a combined list of all customers and all orders, along with the corresponding details where available, a FULL JOIN would be useful. This ensures that all customers and all orders are included in the result, with NULL values indicating where there are no matches.
5. **CROSS JOIN**:
   * A CROSS JOIN returns the Cartesian product of the two tables, combining each row from the first table with every row from the second table.
   * Example scenario: If you want to generate a list of all possible combinations of items from two different tables, such as generating all possible combinations of products and categories, a CROSS JOIN would be appropriate.

9. Describe SQL expressions and provide examples of BOOLEAN, NUMERIC, and DATE expressions.

Solution- Boolean

SELECT \* FROM customers WHERE is\_active = TRUE;

Numeric-

SELECT order\_id, (unit\_price \* quantity) \* (1 - discount) AS total\_price FROM order\_details;

Date-

SELECT name, birthdate, CURRENT\_DATE() - birthdate AS age FROM customers;

10. Explain the usage of aggregate functions in SQL with examples of COUNT, SUM, AVG, MAX, and MIN.

Solution- SELECT COUNT(\*) FROM students;

SELECT COUNT(age) FROM students;

SELECT SUM(sales) FROM orders;

SELECT AVG(age) FROM students;

SELECT MAX(salary) FROM employees;

SELECT MIN(temperature) FROM weather;

11. Discuss different types of SQL clauses such as GROUP BY, HAVING, and ORDER BY. Provide examples for each.

1. Solution- **GROUP BY:**
   * The GROUP BY clause is used to group rows that have the same values into summary rows.
   * It is often used with aggregate functions like COUNT(), SUM(), AVG(), etc., to perform calculations on groups of data.
   * Example: Suppose we have a table named **sales** with columns **region** and **sales\_amount**, and we want to find the total sales amount for each region.

SELECT region, SUM(sales\_amount) AS total\_sales

FROM sales

GROUP BY region;

1. **HAVING:**
   * The HAVING clause is used to filter the results of aggregate functions applied to grouped data.
   * It is similar to the WHERE clause but is used specifically with GROUP BY.
   * Example: Suppose we want to find regions where the total sales amount exceeds 10000.

SELECT region, SUM(sales\_amount) AS total\_sales

FROM sales

GROUP BY region

HAVING SUM(sales\_amount) > 10000;

1. **ORDER BY:**
   * The ORDER BY clause is used to sort the result set in ascending or descending order based on one or more columns.
   * By default, it sorts in ascending order, but you can specify "ASC" for ascending or "DESC" for descending order.
   * Example: Suppose we want to retrieve a list of employees sorted by their salaries in descending order.

SELECT employee\_name, salary

FROM employees

ORDER BY salary DESC;

12. Explain SQL constraints like NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, and DEFAULT with suitable examples.

Solution- **NOT NULL:**

* Ensures that a column cannot have a NULL value.
* Example: Creating a table named **students** with a column **student\_id** where NULL values are not allowed.

CREATE TABLE students (

student\_id INT NOT NULL,

name VARCHAR(50)

);

**UNIQUE:**

* Ensures that all values in a column are unique.
* Example: Adding a UNIQUE constraint to the **email** column in a table named **users**

CREATE TABLE users (

user\_id INT PRIMARY KEY,

email VARCHAR(100) UNIQUE,

name VARCHAR(50)

);

**PRIMARY KEY:**

* Unique identifier for each record in a table. It combines the functionalities of UNIQUE and NOT NULL constraints.
* Example: Creating a table named **employees** with a primary key on the **employee\_id** column.

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

department VARCHAR(50)

);

**FOREIGN KEY:**

* Establishes a link between data in two tables. It ensures referential integrity by enforcing a relationship between the foreign key column and the primary key or unique key in another table.
* Example: Adding a foreign key constraint to the **department\_id** column in a table named **employees** referencing the **department\_id** column in a table named **departments**.

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

department\_id INT,

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50)

);

**DEFAULT:**

* Specifies a default value for a column if no value is specified during insertion.
* Example: Adding a DEFAULT constraint to the **status** column in a table named **orders**, setting the default value to 'Pending'.

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

order\_date DATE,

status VARCHAR(20) DEFAULT 'Pending'

);

13. What is a SQL subquery? How is it different from a regular query? Provide examples.

Solution- A SQL subquery, also known as a nested query or inner query, is a query nested within another SQL query. It is used to retrieve data from one or more tables based on a condition defined in the outer query. Subqueries can be used in various parts of a SQL statement such as SELECT, FROM, WHERE, and so on.

The main difference between a regular query and a subquery is that a regular query retrieves data directly from tables or views, while a subquery retrieves data from the result set of another query.

Regular Query

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE department\_id = 10;

Sub Query

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE department\_name = 'HR'

);

14. Describe triggers in SQL. How are they useful in database management? Give an example.

Solution- In SQL, a trigger is a special type of stored procedure that automatically executes in response to certain events occurring in the database. These events can include INSERT, UPDATE, DELETE operations on specific tables or views. Triggers are useful in database management for enforcing data integrity, implementing business rules, auditing changes, and maintaining consistency.

Triggers are categorized into two types based on when they are executed:

1. **Before Triggers (BEFORE INSERT/UPDATE/DELETE):**
   * These triggers execute before the specified operation (INSERT, UPDATE, DELETE) is performed on the table.
   * They can be used to validate data or perform actions based on certain conditions before the change is applied to the table.
2. **After Triggers (AFTER INSERT/UPDATE/DELETE):**
   * These triggers execute after the specified operation (INSERT, UPDATE, DELETE) has been performed on the table.
   * They can be used to perform actions such as logging changes, updating related tables, or sending notifications after the change has been made.

Here's an example of a trigger in SQL:

Let's say we have a table named **employees** with columns **employee\_id**, **first\_name**, **last\_name**, and **salary**, and we want to enforce a business rule that restricts salary increases to a maximum of 10%. We can create an "Before Update" trigger to check and enforce this rule:

CREATE OR REPLACE TRIGGER enforce\_salary\_increase

BEFORE UPDATE OF salary ON employees

FOR EACH ROW

BEGIN

IF :NEW.salary > (:OLD.salary \* 1.1) THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary increase exceeds the maximum limit of 10%.');

END IF;

END;

/

In this trigger:

* **BEFORE UPDATE OF salary ON employees** specifies that the trigger will be executed before any updates to the **salary** column of the **employees** table.
* **FOR EACH ROW** indicates that the trigger will execute for each row being updated.
* **:OLD.salary** represents the old value of the **salary** column before the update.
* **:NEW.salary** represents the new value of the **salary** column after the update.
* The trigger checks if the new salary exceeds 10% more than the old salary (**:OLD.salary \* 1.1**). If it does, it raises an error using **RAISE\_APPLICATION\_ERROR**.

15. Discuss various SQL operators such as arithmetic, comparison, and logical operators. Provide examples for each.

Solution-

**Arithmetic Operators:**

* Arithmetic operators perform mathematical operations on numeric values.
* Common arithmetic operators include addition (+), subtraction (-), multiplication (\*), division (/), and modulo (%).

Example

SELECT 10 + 5 AS addition,

20 - 8 AS subtraction,

6 \* 4 AS multiplication,

20 / 5 AS division,

17 % 5 AS modulo;

OUTPUT

+-----------+-------------+----------------+----------+--------+

| addition | subtraction | multiplication | division | modulo |

+-----------+-------------+----------------+----------+--------+

| 15 | 12 | 24 | 4 | 2 |

+-----------+-------------+----------------+----------+--------+

**Comparison Operators:**

* Comparison operators are used to compare values and return a Boolean result (true or false).
* Common comparison operators include equal to (=), not equal to (!= or <>), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

SELECT \* FROM employees WHERE salary > 50000;

**Logical Operators:**

* Logical operators are used to combine multiple conditions in SQL queries.
* Common logical operators include AND, OR, and NOT.

SELECT \* FROM employees WHERE department = 'IT' AND salary > 60000;

**Concatenation Operator:**

* The concatenation operator (||) is used to concatenate two or more strings into a single string.

SELECT first\_name || ' ' || last\_name AS full\_name FROM employees;

**NULL Comparison Operators:**

* Special operators used to compare values with NULL.
* Common NULL comparison operators include IS NULL and IS NOT NULL

SELECT \* FROM students WHERE grade IS NULL;