SQL - Advanced Interview Questions

(Practice Project)

PW SKILLS

Basic

1. What is normalization, and why is it important in database design?

Answer:
Normalization is a database design process that organizes data into related tables to reduce redundancy and improve data integrity. It involves structuring data according to specific rules, such as ensuring that each table has unique records and that attributes are fully dependent on primary keys.

Why It's Important:

Reduces Redundancy: Eliminates duplicate data.
Enhances Data Integrity: Prevents data anomalies and inconsistencies.
Simplifies Maintenance: Makes updates and changes more manageable.

Normalization is essential for creating efficient, reliable, and scalable databases.

2. What is an Entity-Relationship (ER) diagram, and what is its purpose in database design?

Answer:
An Entity-Relationship (ER) diagram is a visual representation of the entities, relationships, and attributes within a database. It serves as a blueprint for designing and structuring the database.

Key Components:

Entities: Objects or concepts, like "Customer" or "Order," that represent data stored in the database.
Attributes: Details or properties of entities, such as a customer's name or an order date.
Relationships: Connections between entities, like a customer placing an order.

Purpose in Database Design:

Visual Planning: Helps in planning the database structure by showing how entities relate to each other.

Communication: Facilitates communication between developers and stakeholders, ensuring everyone understands the database design.

Foundation for Development: Provides a foundation for creating the actual database schema, ensuring a well-organized and efficient database.

3. What is a primary key in SQL?

Answer:
A primary key is a column (or a set of columns) in a database table that uniquely identifies each row within that table. The primary key ensures that no duplicate entries exist and that each record can be uniquely identified. It automatically enforces uniqueness and creates an index on the primary key column(s). For example, in a table of employees, the EmployeeID could be the primary key because it uniquely identifies each employee.

4. What is the difference between the SELECT and SELECT DISTINCT statements?

Answer:
The SELECT statement is used to retrieve all matching records from a database table based on the specified criteria. It returns all rows, including duplicates.

The SELECT DISTINCT statement, on the other hand, is used to return only unique rows, removing any duplicate records from the result set. For example, SELECT DISTINCT Country FROM Customers; would return a list of unique countries without duplicates.

5. What is a foreign key in SQL?

Answer:
A foreign key is a column or a set of columns in one table that uniquely identifies rows in another table. It creates a link between the two tables, enforcing referential integrity. The foreign key in the child table points to the primary key in the parent table, ensuring that the value in the foreign key column must match an existing value in the parent table's primary key. For example, in a table of Orders, the CustomerID might be a foreign key referencing the CustomerID primary key in the Customers table.

6. What is the difference between DELETE and TRUNCATE in SQL?

Answer:
The DELETE statement is used to remove specific rows from a table based on a condition. You can use the WHERE clause with DELETE to specify which records should be removed. It can be rolled back if used within a transaction.

The TRUNCATE statement, on the other hand, removes all rows from a table, effectively resetting the table to its empty state. Unlike DELETE, TRUNCATE is usually faster and does not generate individual row delete operations. It cannot be rolled back in many databases, and it resets any auto-increment counters associated with the table.

7. What is the difference between the GROUP BY and HAVING clauses in SQL?

Answer:
The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows. It is typically used with aggregate functions like COUNT, SUM, AVG, etc., to produce summary results. For example, GROUP BY Department would group employees by their department.

The HAVING clause is used to filter the results of a GROUP BY query. While the WHERE clause filters rows before they are grouped, the HAVING clause filters groups after they have been formed. For example, HAVING COUNT(*) > 5 would filter out groups (e.g., departments) that have 5 or fewer employees.

Intermediate

8. What is a temporary table in SQL, and how does it differ from a regular table?

Answer:
A temporary table in SQL is a special type of table that is created and used within a specific session or transaction. It is automatically deleted when the session ends or when explicitly dropped.

Differences from a Regular Table:

- **Scope:** Temporary tables exist only within the session or transaction that created them, whereas regular tables persist in the database until manually deleted.
- **Lifetime:** Temporary tables are automatically removed when the session ends, while regular tables remain in the database.
- **Usage:** Temporary tables are typically used for intermediate data storage or processing within complex queries, whereas regular tables store long-term data.

Temporary tables are useful for handling data that is needed only temporarily, without cluttering the database with permanent tables.

9. What are stored procedures, and how do they differ from regular SQL queries?
Answer:

Stored procedures are precompiled collections of SQL statements that can be executed as a single unit. They are stored in the database and can include control-flow logic, variables, and error handling.

Differences from Regular SQL Queries:

- **Precompiled:** Stored procedures are compiled once and can be executed multiple times, whereas regular SQL queries are compiled and executed each time they are run.
- **Reusability:** Stored procedures can be reused with different parameters, while regular SQL queries are typically written and executed for specific tasks.
- **Performance:** Stored procedures often perform better than regular queries because they are precompiled and optimized by the database engine.
- **Functionality:** Stored procedures can include complex logic, such as loops and conditional statements, which are not possible with regular SQL queries.

Stored procedures are valuable for encapsulating complex logic, improving performance, and promoting code reuse in database operations.

10. Explain the difference between input and output parameters in stored procedures.

Answer:
In stored procedures, input and output parameters are used to pass values into and out of the procedure.
Input Parameters:

- **Purpose:** Input parameters allow you to pass data into the stored procedure.
- **Behavior:** The values of input parameters are provided by the caller when the procedure is executed. These values are used within the procedure but do not change after the procedure runs.
- **Example:** A procedure might take a customer ID as an input parameter to retrieve specific customer details.

Output Parameters:
- **Purpose:** Output parameters allow the stored procedure to return data back to the caller.
- **Behavior:** The procedure modifies the value of the output parameter during execution, and this modified value is returned to the caller.
- **Example:** A procedure might calculate a total price and return it through an output parameter.

Key Difference:

Direction of Data Flow: Input parameters bring data into the procedure, while output parameters send data back to the caller.

11. What are User-Defined Functions (UDFs) in SQL, and how do they differ from stored procedures?

Answer:
User-Defined Functions (UDFs) in SQL are custom functions created by users to perform specific tasks and return a value or a table. They are similar to built-in functions like SUM() or AVG(), but they are defined by the user.

Differences from Stored Procedures:

1. **Return Type:**
   - **UDFs:** Always return a value, either a single scalar value (e.g., integer, string) or a table.
   - **Stored Procedures:** May or may not return a value. They often perform operations that don't necessarily produce a direct result, like updating a table.

2. **Usage in Queries:**
   - **UDFs:** Can be used directly in SQL statements (e.g., SELECT, WHERE, JOIN) like any other function.
   - **Stored Procedures:** Cannot be used directly in SQL statements; they must be executed using EXEC or CALL commands.

3. **Side Effects:**
   - **UDFs:** Typically do not allow modifications to the database (e.g., inserting, updating, or deleting records).
   - **Stored Procedures:** Can perform any kind of database modification.

4. **Complexity:**
   - **UDFs:** Designed for specific, often simple tasks, like calculations or formatting data.
   - **Stored Procedures:** Can handle more complex logic, including multiple SQL statements, transaction control, and error handling.

Summary:

UDFs are primarily used to return a value and can be embedded in SQL queries, whereas stored procedures are more versatile, allowing complex operations and database modifications but are executed independently.

12. Explain the difference between Scalar Functions and Table-Valued Functions (TVFs).

Answer:
Scalar Functions and Table-Valued Functions (TVFs) are both types of User-Defined Functions (UDFs) in SQL, but they differ in the type of data they return and how they are used.

Scalar Functions:
- **Return Type:** Scalar functions return a single value, such as an integer, string, or date.
- **Usage:** They can be used wherever a single value is expected, such as in a SELECT list, WHERE clause, or ORDER BY clause.
- **Example:** A scalar function might return the square of a number or format a date string.

Example:

```
CREATE FUNCTION dbo.GetSquare(@Number INT)
RETURNS INT
AS
BEGIN
    RETURN @Number * @Number;
END;
```

This function returns the square of the input number.

Table-Valued Functions (TVFs):

- **Return Type:** TVFs return a table, which can be treated like any other table or view in SQL.
- **Usage:** They can be used in SELECT statements, JOIN operations, or as part of a query that requires a set of rows.
- **Example:** A TVF might return a list of customers based on specific criteria.

Example:

```
CREATE FUNCTION dbo.GetCustomersByCity(@City NVARCHAR(50))
RETURNS TABLE
AS
RETURN
(
    SELECT CustomerID, CustomerName, City
    FROM Customers
    WHERE City = @City
);
```

This function returns a table of customers from a specific city.

Key Differences:

- **Output:** Scalar functions return a single value; TVFs return a table.
- **Use Case:** Scalar functions are used for calculations or operations that result in a single value. TVFs are used when you need to return a set of rows or a table.

13. How do aggregate functions differ from scalar functions in SQL?

Answer:
Aggregate functions and scalar functions in SQL both perform operations on data, but they differ in how they process and return results.

Aggregate Functions:

- **Purpose:** Aggregate functions perform calculations on a set of values and return a single value summarizing that set.
- **Scope:** They work on multiple rows of data and return a summary result.
- **Examples:** Common aggregate functions include SUM(), AVG(), COUNT(), MIN(), and MAX().
- **Usage:** Typically used in SELECT statements with GROUP BY clauses to summarize data across multiple rows.

Example: SELECT AVG(Salary) FROM Employees;
This query calculates the average salary of all employees.

Scalar Functions:

- **Purpose:** Scalar functions perform operations on a single value and return a single result.
- **Scope:** They operate on individual data points rather than a set of rows.
- **Examples:** Examples of scalar functions include ABS(), ROUND(), UPPER(), and user-defined scalar functions.
- **Usage:** Used in SELECT statements, WHERE clauses, or anywhere a single value is needed.

Example: SELECT UPPER(FirstName) FROM Employees;
This query converts the FirstName of each employee to uppercase.

Key Differences:

- **Input and Output:**
  - **Aggregate Functions:** Take multiple values as input and return a single summary value.
  - Scalar Functions: Take a single value as input and return a single value.
- **Context:**
  - **Aggregate Functions:** Often used with GROUP BY to group data and produce summary results.
  - **Scalar Functions:** Used to manipulate or calculate individual data points.

14. What are the key differences between a view and a table in SQL?

Answer:
In SQL, views and tables are both used to store and retrieve data, but they have distinct characteristics and purposes. Here are the key differences between them:

1. Definition:
- **Table:** A table is a physical storage structure in a database that holds data in rows and columns. It is where data is actually stored.
- **View:** A view is a virtual table created by a query that selects data from one or more tables. It does not store data itself; instead, it presents data stored in tables.

2. Data Storage:
- **Table:** Data is physically stored in the database.
- **View:** Does not store data; it dynamically generates the data based on the underlying tables each time it is queried.

3. Updatability:
- **Table:** Data in tables can be inserted, updated, or deleted directly.
- **View:** Views can be read-only or updatable, depending on how they are defined. Certain views (especially those based on joins or aggregate functions) may not allow data modifications.

4. Purpose:
- **Table:** Used for storing actual data in the database.
- **View:** Used to simplify complex queries, encapsulate business logic, or provide a specific representation of data for security or convenience.

5. Performance:
- **Table:** Direct access to table data is generally more efficient since data is physically stored.
- **View:** May have performance overhead since the underlying query needs to be executed each time the view is accessed. However, indexed views can improve performance in some cases.

6. Security:
- **Table:** Access to tables can be controlled through permissions at the table level.
- **View:** Can be used to restrict access to specific data by exposing only certain columns or rows, providing an additional layer of security.

Summary:

Tables are the primary data storage structures in a database, while views are virtual tables that provide a way to simplify data access, enhance security, and present data in a specific format without storing it physically.

15. What are the different types of cursors available in SQL, and how do they differ from each other?

Answer:
In SQL, cursors are used to retrieve and manipulate data row by row. There are several types of cursors, each with distinct characteristics. The main types of cursors are:

1. Static Cursor:
- **Description:** A static cursor creates a temporary copy of the result set. It does not reflect changes made to the underlying data after the cursor is opened.
- **Use Case:** Suitable when the data needs to remain unchanged during processing, such as generating reports.

2. Dynamic Cursor:
- **Description:** A dynamic cursor reflects all changes made to the data in the underlying tables while the cursor is open. This includes inserts, updates, and deletes.
- **Use Case:** Useful when real-time data is required, as it provides the most current view of the data.

3. Forward-Only Cursor:
- **Description:** A forward-only cursor allows traversal of the result set in one direction—from the first row to the last row. You cannot move backward.
- **Use Case:** Efficient for processing large result sets where you only need to read data sequentially.

4. Keyset-Driven Cursor:
- **Description:** A keyset-driven cursor allows updates to the rows identified by the cursor, but new rows added after the cursor is opened are not visible. Changes to existing rows are visible.
- **Use Case:** Suitable when you need to work with a stable set of keys while allowing updates.

5. Read-Only Cursor:
- **Description:** A read-only cursor is designed for retrieving data only. You cannot update the data through this cursor.
- **Use Case:** Efficient for scenarios where data retrieval is the only requirement.

Differences:

- **Data Reflection:** Static cursors do not reflect changes, while dynamic cursors do. Keyset-driven cursors allow updates but not new rows.
- **Directionality:** Forward-only cursors allow movement only in one direction, while others may allow both forward and backward navigation.
- **Modification Capability:** Some cursors allow data modifications (e.g., dynamic and keyset-driven), while others are read-only.

Summary:

The choice of cursor type depends on the specific needs of your application, including whether you require real-time data updates, the direction of data processing, and whether you need to modify the data.

Key Differences from Stored Procedures:

1. **Execution Context:**
   - **Trigger:** Automatically executed in response to data modification events on a table.
   - **Stored Procedure:** Executed explicitly by the user or application through a CALL or EXEC statement.

2. **Invocation:**
   - **Trigger:** Cannot be called directly; it is invoked by the database system when the specified event occurs.
   - **Stored Procedure:** Can be called at any time when needed by the user or application.

3. **Purpose:**
   - **Trigger:** Primarily used for data validation, enforcing referential integrity, logging changes, or auditing data modifications.
   - **Stored Procedure:** Used for a broader range of tasks, including complex business logic, data manipulation, and retrieval.

4. **Return Value:**
   - **Trigger:** Does not return a value to the caller; it operates in the context of the event that triggered it.
   - **Stored Procedure:** Can return values or result sets to the caller.

5. **Transaction Handling:**
   - **Trigger:** Executes as part of the transaction that initiated the triggering event; if the transaction is rolled back, so are any changes made by the trigger.
   - **Stored Procedure:** Can manage its own transactions independently of the calling context.

Summary:

Triggers are automatic, event-driven operations tied to data modifications in a table, while stored procedures are explicit, user-defined functions that can perform a wide range of tasks.

17. Explain the difference between AFTER and BEFORE triggers.

Answer:
AFTER and BEFORE triggers are types of database triggers that determine when the trigger action occurs in relation to the triggering event (such as INSERT, UPDATE, or DELETE). Here are the key differences:

1. Timing of Execution:

- **BEFORE Trigger:**
  - Executes before the triggering event occurs.
  - Can be used to validate or modify data before it is written to the database.

- **AFTER Trigger:**
  - Executes after the triggering event has occurred.
  - Typically used for actions that depend on the successful completion of the data modification.

2. Use Cases:

- **BEFORE Trigger:**
  - Useful for enforcing business rules, such as validating input data, transforming data before insertion, or preventing invalid data from being saved.
  - **Example:** Ensuring a new record meets certain criteria before it is added to the table.

- **AFTER Trigger:**
  - Useful for actions that need to occur as a consequence of the data modification, such as logging changes, updating related records, or performing additional calculations.
  - **Example:** Auditing changes to a record or notifying other systems after a successful update.

3. Impact on Data Modification:

- **BEFORE Trigger:**
  - Can modify the values being inserted or updated, as the trigger runs before the actual data operation.

- **AFTER Trigger:**
  - Cannot modify the values of the operation being performed since it runs after the data modification is completed.

Summary:

BEFORE triggers execute before the data modification, allowing validation and data transformation, while **AFTER** triggers execute after the modification, focusing on follow-up actions or side effects of the change.

18. What is a CASE statement in SQL and What is the difference between CASE and COALESCE in SQL.

Answer:
CASE: Evaluates a list of conditions and returns one of multiple possible result expressions.
COALESCE: Returns the first non-null expression among its arguments.

Example:

```
SELECT
    EmployeeName,
    Salary,
    CASE
        WHEN Salary > 50000 THEN 'High'
        WHEN Salary BETWEEN 30000 AND 50000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryCategory
FROM Employees;
```

Difference: CASE is more flexible with multiple conditions, while COALESCE is simpler and best used for dealing with NULL values.

19. Explain the difference between RANK(), DENSE_RANK(), and ROW_NUMBER() in SQL.

Answer:
RANK(): Assigns a unique rank to each row within a partition of a result set. If there are ties, it leaves gaps in the ranking.

DENSE_RANK(): Similar to RANK(), but without leaving gaps in the ranking sequence if there are ties.
ROW_NUMBER(): Assigns a unique number to each row within a partition, with no regard for ties.

20. Explain the difference between LAG() and LEAD() functions in SQL. In what scenarios might you use each of these functions?

Answer:
LAG(): Retrieves the value of a previous row from the current row within the same result set.
LEAD(): Retrieves the value of a following row from the current row within the same result set.

Use Cases:

LAG(): Useful for comparing a row with the previous one (e.g., comparing sales figures month over month).
LEAD(): Useful for forecasting or planning (e.g., getting the next month's sales target).

Example:

```
SELECT
    EmployeeName,
    Salary,
    LAG(Salary, 1) OVER (ORDER BY Salary DESC) AS
PreviousSalary,
    LEAD(Salary, 1) OVER (ORDER BY Salary DESC) AS
NextSalary
FROM Employees;
```

Advanced

21. Write an SQL query to assign a unique row number to each employee within their respective departments, ordered by their salary. Use the ROW_NUMBER() window function.

Employee_ID	Department	Employee_Name	Salary
1	HR	Alice	60,000
2	HR	Bob	75,000
3	HR	Charlie	70,000
4	IT	David	80,000
5	IT	Eve	90,000
6	IT	Frank	85,000

Answer:

```
SELECT
    Department,
    EmployeeName,
    Salary,
    ROW_NUMBER() OVER (PARTITION BY Department ORDER BY
Salary DESC) AS RowNumber
FROM Employees;
```

22. Write an SQL query to calculate the cumulative total of salaries for employees, ordered by their hiring date. Use the SUM() window function with appropriate ordering.

Employee_ID	Employee_Name	Hiring_Date	Salary
1	Alice	2021-01-15	60,000
2	Bob	2021-03-22	75,000
3	Charlie	2021-04-10	70,000
4	David	2021-05-05	80,000
5	Eve	2021-06-18	90,000
6	Frank	2021-07-25	85,000

Answer:

```
SELECT
    EmployeeName,
    HireDate,
    Salary,
    SUM(Salary) OVER (ORDER BY HireDate) AS
CumulativeSalary
FROM Employees;
```

23. Write an SQL query using a CASE statement to assign a grade ('A', 'B', 'C') to students based on their scores, with the following conditions: 'A' for scores above 80, 'B' for scores between 50 and 80, and 'C' for scores below 50.

Sample Data:

Student_ID	Student_Name	Score
1	John	85
2	Alice	78
3	Bob	62
4	Eve	47
5	Charlie	90

Answer:

A: Scores above 80
B: Scores between 50 and 80
C: Scores below 50

```
SELECT
    StudentName,
    Score,
    CASE
        WHEN Score > 80 THEN 'A'
        WHEN Score BETWEEN 50 AND 80 THEN 'B'
        ELSE 'C'
    END AS Grade
FROM Students;
```

24. Create a temporary table to store a list of employees whose salaries are above the average salary. Then, write a query to retrieve these employees' names and salaries from the temporary table.

Sample Data:

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000
4	David	90,000
5	Eve	40,000

Answer:

```
CREATE TEMPORARY TABLE HighSalaryEmployees AS
SELECT EmployeeName, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);

SELECT * FROM HighSalaryEmployees;
```

25. Create a stored procedure that accepts an employee ID and returns the employee's name and salary.

- *Sample Data:*

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000

Answer:

```
CREATE PROCEDURE GetEmployeeDetails (IN empID INT, OUT
empName VARCHAR(100), OUT empSalary DECIMAL(10,2))
BEGIN
    SELECT EmployeeName, Salary INTO empName, empSalary
    FROM Employees
    WHERE EmployeeID = empID;
END;
```

26. Write a stored procedure to update the salary of an employee. The procedure should accept the employee ID and the new salary as input parameters.

- *Sample Data:*

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000

Answer:

```
CREATE PROCEDURE UpdateEmployeeSalary (IN empID INT, IN
newSalary DECIMAL(10,2))
BEGIN
    UPDATE Employees
    SET Salary = newSalary
    WHERE EmployeeID = empID;
END;
```

27. Create a Scalar Function to Calculate the Net Salary After Deducting a Fixed Percentage of Tax.

Sample Data:

Employee_ID	Employee_Name	Gross_Salary	Tax_Percentage
1	John Doe	5000	10
2	Jane Smith	6000	15
3	Bob Johnson	4500	12

Answer:

```
CREATE FUNCTION CalculateNetSalary (@Salary DECIMAL(10,2),
@TaxRate DECIMAL(5,2))
RETURNS DECIMAL(10,2)
AS
BEGIN
    RETURN @Salary * (1 - @TaxRate/100);
END;
```

28. Create a Table-Valued Function to Retrieve All Employees in a Specific Department.

Sample Data:

Employee_ID	Employee_Name	Department_ID
1	John Doe	101
2	Jane Smith	102
3	Bob Johnson	101
4	Alice Brown	103

Answer:

```
CREATE FUNCTION GetEmployeesByDepartment (@DepartmentID
INT)
RETURNS TABLE
AS
RETURN
(
    SELECT EmployeeID, EmployeeName, Salary
 FROM Employees
    WHERE DepartmentID = @DepartmentID
);
```

29. Create a view that lists all customers who have placed orders worth more than $1000.

- Sample Data:

Order_ID	Customer_Name	Order_Total
101	Alice Brown	1200.00
102	Bob Smith	950.00
103	Carol Jones	1500.00
104	David White	750.00
105	Emma Taylor	1300.00

Answer:

```
CREATE VIEW HighValueCustomers AS
SELECT CustomerName, SUM(OrderAmount) AS TotalOrderAmount
FROM Orders
GROUP BY CustomerName
HAVING SUM(OrderAmount) > 1000;
```

30. Write a cursor that iterates through all orders and applies a 10% discount to orders where the total value exceeds $1000.

- Sample Data:

Customer_ID	Customer_Name	Order_ID
1	Alice Brown	101
2	Bob Smith	102
1	Alice Brown	103
3	Carol Jones	104
2	Bob Smith	105

Answer:

```
DECLARE @OrderID INT, @OrderAmount DECIMAL(10,2);

DECLARE OrderCursor CURSOR FOR
SELECT OrderID, OrderAmount
FROM Orders
WHERE OrderAmount > 1000;

OPEN OrderCursor;
FETCH NEXT FROM OrderCursor INTO @OrderID, @OrderAmount;

WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE Orders
    SET OrderAmount = OrderAmount * 0.9
    WHERE OrderID = @OrderID;

    FETCH NEXT FROM OrderCursor INTO @OrderID,
@OrderAmount;
END;

CLOSE OrderCursor;
DEALLOCATE OrderCursor;
```

31. Create a trigger that automatically logs any deletions from a Products table into a DeletedProductsLog table, capturing the product's Product_ID, Product_Name, and the deletion date.

Sample Data for `Products` Table:

Product_ID	Product_Name	Price
1	Laptop	1000
2	Smartphone	500
3	Tablet	300

Expected Data for `DeletedProductsLog` Table After Deletion:

Log_ID	Product_ID	Product_Name	Deletion_Date
1	2	Smartphone	2024-08-14 14:00:00

Answer:

```
CREATE TRIGGER LogProductDeletion
AFTER DELETE ON Products
FOR EACH ROW
BEGIN
    INSERT INTO DeletedProductsLog (Product_ID,
Product_Name, DeletionDate)
    VALUES (OLD.Product_ID, OLD.Product_Name, NOW());
END;
```