WEEKLY TASK

1. What is an aggregate function in SQL? Give an example.

An aggregate function in SQL performs a calculation on a set of values and returns a single value. Some common aggregate functions include:

- COUNT()
- SUM()
- AVG()
- MIN()
- MAX()

Example

```
select max(salary)
from employee;
```

2. How can you use the GROUP BY clause in combination with aggregate functions?

- **Specify the Columns to Group By**: Use the `GROUP BY` clause to define which column(s) should be used to group the rows.
- **Apply Aggregate Functions**: Use aggregate functions such as `COUNT()`, `SUM()`, `AVG()`, `MIN()`, or `MAX()` to perform calculations on the grouped data.

Example

```
SELECT customer_id, SUM(order_amount) AS total_order_amount
FROM Orders
GROUP BY customer_id;
```

3. Describe a scenario where atomicity is crucial for database operations.

Atomicity in a database context ensures that a transaction is treated as a single, indivisible unit of work. It guarantees that either all operations within the transaction are executed successfully or none of them are, maintaining data integrity and consistency.

Example:Amount transaction in Bank

4. Mention any 2 of the difference between OLAP and OLTP?

Criteria	OLAP	OLTP
Purpose	OLAP helps you analyze large volumes of data to support decision-making.	OLTP helps you manage and process real-time transactions.

Data source	OLAP uses historical and aggregated data from multiple sources.	OLTP uses real-time and transactional data from a single source.

5. How do you optimize an OLTP database for better performance?Hint: index

- Create Indexes
- Use Composite Indexes
- Avoid Over-Indexing

6. What are the different types of data encryption available in MSSQL?

1. Transparent Data Encryption (TDE)
2. Always Encrypted
3. Cell-Level Encryption
4. Backup Encryption
5. Transport Layer Security (TLS)

7. What is the main difference between SQL and NoSQL databases?

SQL	NoSQL
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have a dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.

8. How do you create a new schema in MSSQL?

CREATE SCHEMA Sales

9. Describe the process of altering an existing table.

To alter the table or add the column, we use the syntax

ALTER TABLE table_name ADD column_name data_type

10. What is the difference between a VIEW and a TABLE in MSSQL?

Basis	View	Table
Definition	A view is a database object that allows generating a logical subset of data from one or more tables.	A table is a database object or an entity that stores the data of a database.
Dependency	The view depends on the table.	The table is an independent data object.
Recreate	We can easily use replace option to recreate the view.	We can only create or drop the table.
Aggregation of data	Aggregate data in views.	We can not aggregate data in tables.

11. Explain how to create and manage indexes in a table.

```
Create index
idx_firstname
on case_players
(firstname);

select * from case_players;

Drop index case_players.idx_firstname;
```

12. What are the most commonly used DML commands?

- Select
- update
- delete
- insert

13. How do you retrieve data from multiple tables using a JOIN?

```
Select player_id,customer_id,location
from player_details
natural join
order_details
natural join
 customer_details
```

14. Explain how relational algebra is used in SQL queries.

relational algebra is a theoretical framework for querying relational databases, providing a set of operations to manipulate and retrieve data from relational tables. SQL (Structured Query Language) is based on relational algebra, and many SQL operations correspond to relational algebra operations.

Some of the relational algebraic expression are,

- Union
- Difference

15. What are the implications of using complex queries in terms of performance?

Using complex queries in SQL can have several implications for performance. Understanding these implications can help you optimize queries and maintain efficient database operations. Here are the key performance considerations:

1. Execution Time
2. Resource Utilization
3. Index Utilization

16. How does the HAVING clause differ from the WHERE clause when using aggregate functions? Show whether you could use having before group by in the select statement

`WHERE` Clause:

- Purpose: Filters rows before any grouping or aggregation occurs.

`HAVING` Clause:

- Purpose: Filters groups of rows after grouping and aggregation have been performed.

In SQL, you cannot use the `HAVING` clause before the `GROUP BY` clause in the `SELECT` statement.

Correct SQL Query Structure

1)SELECT
2)FROM
3)WHERE
4)GROUP BY
5)HAVING
6)ORDER BY

17. What are filters in SQL and how are they used in queries?

Using the

`WHERE` Clause

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

```
WHERE condition;
```

Comparison Operators:

```
SELECT *
FROM orders
WHERE order_date >= '2024-01-01';
```

18. How do you use the `WHERE` clause to filter data in MSSQL?

Using the

`WHERE` Clause

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

19. How can you combine multiple filter conditions using logical operators?

Logical Operators

1. **AND Operator:**

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

2. `OR` Operator:

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

20. Explain the use of `CASE` statements for filtering data in a query.

```
CASE expression
WHEN value1 THEN result1
WHEN value2 THEN result2
...
ELSE default_result
END
```

21. What are the different types of operators available in MSSQL?

1. **Arithmetic Operators**
2. **Comparison Operators**
3. **Logical Operators**
4. **Bitwise Operators**

22. How do arithmetic operators work in SQL?

+ (Addition):

```
SELECT 10 + 5 AS Sum;  -- Result: 15
```

- (Subtraction):

SELECT 10 - 5 AS Difference; -- Result: 5

***** (Multiplication):

SELECT 10 * 5 AS Product; -- Result: 50

/ (Division):

SELECT 10 / 5 AS Quotient; -- Result: 2

% (Modulus):

SELECT 10 % 3 AS Remainder; -- Result: 1

23. Explain the use of LIKE operator with wildcards for pattern matching.

The `LIKE` operator in SQL is used for pattern matching within string values. It allows you to search for a specified pattern in a column. The `LIKE` operator is often used with wildcards to define the pattern. Here's a quick rundown of how it works with the common wildcards:

1. Percent Sign (%): Represents zero or more characters.
    - Example: `SELECT * FROM Employees WHERE Name LIKE 'J%';`
2. Underscore (_): Represents a single character.
    - Example: `SELECT * FROM Employees WHERE Name LIKE 'J_n';`

24. What is normalization and why is it important?

Normalization is the process of reducing data redundancy in a table and improving data integrity. Then why do you need it? If there is no normalization in SQL, there will be many problems, such as:

- Insert Anomaly
- Update Anomaly
- Delete exception

25. Describe the basic normal forms. Hint: 1NF 2NF NF

First Normal Form, or 1NF

removes repeated groups from a table to guarantee atomicity.

The Second Normal Form, or 2NF

lessens redundancy by eliminating partial dependencies.

The Third Normal Form, or 3NF

reduces data duplication by removing transitive dependencies.

26. Mention any one impact of normalization on database performance.

- reduce the storage space required for your data
- eliminate duplicate or unnecessary values

27. What are indexes and why are they used?

Indexes contain all the necessary information needed to access items quickly and efficiently. Indexes serve as lookup tables to efficiently store data for quicker retrieval. Table keys are stored in indexes.

USES:

used by the database search engine to speed up data retrieval.

28. How do you create a unique constraint on a table column?

```
CREATE TABLE Persons (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int,
CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

29. Explain the difference between clustered and non-clustered indexes.

CLUSTERED INDEX	NON-CLUSTERED INDEX
A clustered index is faster.	A non-clustered index is slower.
The clustered index requires less memory for operations.	A non-Clustered index requires more memory for operations.
In a clustered index, the clustered index is the main data.	In the Non-Clustered index, the index is the copy of data.

30. How would you optimize index usage in a highly transactional database?

1: Check your database server.

2: Improve indexing strategies.

3: Identify access to database.

4: Evaluate connection capacity.

5: Optimize Queries.

6: Database Performance Resources.

31. What are the different types of joins available in MSSQL?

1. Inner Join
2. Left join
3. Right join
4. Self join

32. Provide an example of a LEFT JOIN query.

```
SELECT column_name(s)
FROM tableA
LEFT JOIN
tableB ON
tableA.column_name = tableB.column_name;
```

***33.* Explain the concept of a self-join and when it might be used.**

A Self Join is a type of a JOIN query used to compare rows within the same table. Unlike other SQL JOIN queries that join two or more tables, a self join joins a table to itself. To use a self join, a table must have a unique identifier column, a parent column, and a child column.

34. How do you perform a full outer join and what is its significance?

An full outer join is a method of combining tables so that the result includes unmatched rows of both tables. If you are joining two tables and want the result set to include unmatched rows from both tables, use a FULL OUTER JOIN clause. The matching is based on the join condition.

Significance

when we need to merge data from two tables and want to ensure that no data from either table is excluded, even if there are no matching rows between them.

35. What is an alias in SQL and how is it used?

SQL aliases are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable. An alias only exists for the duration of that query. An alias is created with the AS keyword.

36. Give an example of using table aliases in a query.

```
SELECT id, CONCAT(firstname, ' ', lastname) AS fullname
FROM students;
```

37. How do you use column aliases in conjunction with aggregate functions?

Using column aliases in conjunction with aggregate functions in SQL allows you to provide meaningful names to the results of aggregate calculations, making your query results easier to understand. Aggregate functions perform a calculation on a set of values and return a single value. Common aggregate functions include SUM, COUNT, AVG, MIN, and MAX.

Syntax:

```
SELECT aggregate_function(column_name) AS alias_name
FROM table_name
 GROUP BY column_name;
```

38. Explain the benefits of using aliases in complex queries.

1. Improved Readability Simplify References:

Aliases shorten table and column names, making queries easier to read and write, especially when dealing with long or complex names

2. Manage Complex Queries Multiple Joins:

In queries involving multiple joins, aliases help differentiate between columns from different tables.

3. Enhance Query Maintainability Easier Updates:

Queries with aliases are easier to modify and maintain. Changing a table or column name requires only a single update in the alias definition rather than throughout the entire query. Consistent Naming: Aliases ensure consistent naming conventions, which helps avoid confusion when querying complex data structures.

4. Resolve Naming Conflicts Avoid Ambiguities:

Aliases resolve conflicts when different tables have columns with the same names, such as during joins.

5. Improve Performance Optimized Execution:

While aliases themselves don't directly improve performance, they can make the query more manageable and help in optimizing the execution plan by clearly defining table and column relationships. 6. Facilitate Query Design and Debugging Design Queries:

Aliases help in designing queries by simplifying the logical structure, especially in cases involving nested queries or multiple layers of data retrieval. Debugging: Aliases make it easier to identify and troubleshoot issues by providing clear, descriptive names for each part of the query.

39. What is the difference between joins and subqueries?

Subquery	JOIN
A subquery is a query nested inside another query and is used to return data that will be used in the	A JOIN is a means for combining fields from two

Subquery	JOIN
main query as a condition to further restrict the data to be retrieved.	tables by using values common to each.
Subqueries can be slower than JOINs, especially if the subquery returns a large number of rows.	JOINs are generally faster than subqueries, especially for large datasets.
Subqueries can be more complex and harder to read, especially when there are multiple levels of nesting.	JOINs can be easier to read and understand, especially for simple queries.
Subqueries can be used in SELECT, WHERE, and FROM clauses, offering more flexibility.	JOINs are used to combine rows from two or more tables based on a related column.

40. When would you prefer a subquery over a join?

Use a join or a subquery anytime that you reference information from multiple tables. Joins and subqueries are often used together in the same query. In many cases, you can solve a data retrieval problem by using a join, a subquery, or both. Here are some guidelines for using joins and queries.

- If you need to combine related information from different rows within a table, then you can join the table with itself.

- Use subqueries when the result that you want requires more than one query and each subquery provides a subset of the table involved in the query.

41. Explain how correlated subqueries work with an example.

A correlated subquery is a type of subquery that depends on the outer query for its values. Unlike a regular subquery, which is executed once and its result is used by the outer query, a correlated subquery is executed once for each row processed by the outer query. This means that the subquery references columns from the outer query.

EXAMPLE:

```
SELECT e.employee_id, e.salary, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id
WHERE e.salary > (
SELECT AVG(salary)
FROM employees e2
WHERE e2.department_id = e.department_id
);
```

42. Discuss the performance implications of using joins vs subqueries.

Performance Implications of Joins

1. **Efficiency with Large Datasets**:
    - **Optimized Execution**: Most relational database management systems (RDBMS) are highly optimized for joins. They can use indexes and various algorithms (like hash joins or merge joins) to efficiently combine large datasets.
    - **Single Scan**: Joins often involve scanning each table once and then combining the rows based on the join conditions, which can be more efficient than multiple scans required by some subqueries.
2. **Indexes Utilization**:
    - **Indexing**: Joins can benefit from indexed columns, especially when joining large tables. The optimizer can use indexes on the join keys to speed up the operation.

Performance Implications of Subqueries

1. **Correlated Subqueries**:
    - **Row-by-Row Execution**: Correlated subqueries are executed once for each row processed by the outer query, which can be inefficient for large datasets. This can lead to a significant performance hit if the subquery is complex or the outer query processes many rows.
2. **Uncorrelated Subqueries**:
    - **Efficiency**: Uncorrelated subqueries are executed once and the result is used by the outer query. If properly indexed, uncorrelated subqueries can perform well. However, the performance depends on how the subquery is optimized and whether it returns a manageable amount of data.

43. What are the different data types available in MSSQL?

In Microsoft SQL Server (MSSQL), there are several data types available that you can use to store different kinds of data. Here's a rundown of the primary data types:

Numeric Data Types:

- int
- bigint
- smallint
- tinyint
- decimal(p, s)
- numeric(p, s)
- float
- real

Date and Time Data Types:

- date
- datetime

Character Data Types :

- char(n)
- nvarchar(n)

44. How do you choose the appropriate data type for a column?

Numeric Data:

Determine if you need integers or floating-point numbers. For whole numbers, choose `int`, `bigint`, `smallint`, or `tinyint`. For precise numeric values with fixed decimal places, use `decimal` or `numeric`. For approximate values, use `float` or `real`.

Character Data:

Identify if the data will be in a fixed or variable length and whether it needs to support Unicode. Use `char` for fixed-length and `varchar` for variable-length. For Unicode, use `nchar` and `nvarchar`.

Date and Time:

Choose `date` for just the date, `time` for just the time, `datetime` or `datetime2` for both date and time, and `datetimeoffset` if you need time zone support.

45. How do you handle data type conversions in queries?

1. Using CAST and CONVERT Functions

a. CAST Function:

• Syntax: CAST(expression AS target_data_type) •
Usage: Converts an expression to a specified data type.

b. CONVERT Function:

• Syntax: CONVERT(target_data_type, expression [, style])

• Usage: Similar to CAST, but with additional options for date and time formatting.

2. Implicit Conversion

a. Automatic Conversion:

• SQL Server automatically converts data types in expressions and operations when compatible data types are involved.

3. Using the TRY_CAST and TRY_CONVERT Functions

a. TRY_CAST Function:

• Syntax: TRY_CAST(expression AS target_data_type)

• Usage: Attempts to cast an expression to a specified data type and returns NULL if the conversion fails.

b. TRY_CONVERT Function:

• Syntax: TRY_CONVERT(target_data_type, expression [, style])

• Usage: Similar to TRY_CAST, but allows formatting styles for date and time conversions.

4. Handling Data Type Conversions in Expressions

a. Concatenation:

When concatenating different data types, SQL Server automatically converts non-string types to strings.

b. Comparisons:

Ensure data types match in comparisons or use explicit conversion to avoid conversion errors.

5. Formatting Date and Time Data

a. Date and Time Conversion:

Use CONVERT with style codes for specific date formats.

46. What is a correlated subquery?

A correlated subquery is a type of subquery that references columns from the outer query. Unlike a regular subquery, which is independent and does not depend on the outer query, a correlated subquery is executed once for each row processed by the outer query. This makes it more dynamic and specific to the context of the outer query.

Example of a Correlated Subquery

```
SELECT e.name, e.salary
FROM employees e
WHERE e.salary > (
SELECT AVG(e2.salary)
FROM employees e2
WHERE e2.department_id = e.department_id
);
```

47. What is non-correlated subquery. HINT: Outer query checks the data from inner sub query

A non-correlated subquery is a type of subquery that is independent of the outer query. Unlike a correlated subquery, a non-correlated subquery does not reference any columns from the outer query and is executed only once, regardless of how many rows are returned by the outer query.

Example of a Non-Correlated Subquery

```
SELECT name, salary
FROM employees
WHERE salary > (
SELECT AVG(salary)
FROM employees
);
```

48. Explain how correlated subqueries can affect query performance.

Correlated subqueries can significantly impact query performance, often in ways that may be less efficient than other query strategies. Here's a detailed explanation of how they affect performance and what factors contribute to their performance characteristics:

- Repeated Execution
- Complexity and Execution Plan
- Index Usage
- Scalability Issues
- Optimization Strategies

49. What is TSQL and how does it extend standard SQL?

Transact-SQL (T-SQL) is an extension of the SQL (Structured Query Language) developed by Microsoft and Sybase. It is used primarily with Microsoft SQL Server and Azure SQL Database. TSQL extends standard SQL by adding additional features and functionality to support complex querying, procedural programming, and administrative tasks.

50. How do you create a stored procedure in MSSQL?

Creating a stored procedure in Microsoft SQL Server (MSSQL) involves defining a reusable block of SQL code that can be executed with a single call. Stored procedures can encapsulate complex business logic, data manipulation, and querying, and can accept parameters to provide dynamic behavior.

Syntax:

```
CREATE PROCEDURE ProcedureName
[ @Parameter1 DataType [ = DefaultValue ],
 @Parameter2 DataType [ = DefaultValue ] ]
AS
BEGIN -- SQL statements go here
END;
```

51. Explain the difference between functions and procedures in MSSQL.

S.NO	Function	Procedure
1.	Functions always return a value after the execution of queries.	The procedure can return a value using “IN OUT” and “OUT” arguments.
2.	In SQL, those functions having a DML statement can not be called from SQL statements. But autonomous transaction functions can be called from SQL queries.	A procedure can not be called using SQL queries.
3.	Each and every time functions are compiled they provide output according to the given input.	Procedures are compiled only once but they can be called many times as needed without being compiled each time.

52. Describe the use of triggers and provide an example scenario.

Uses of Triggers

1. Enforcing Business Rules:

• Triggers can enforce rules that cannot be enforced through constraints alone.

For example, ensuring that an employee's salary does not exceed a certain limit based on their job title.

2. Maintaining Data Integrity:

• Triggers can help maintain data consistency by performing validation or adjustments when data is inserted, updated, or deleted. For instance, a trigger can ensure that inventory levels are updated when a new order is placed.

3. Automatic Auditing:

• Triggers can automatically log changes to a table into an audit table, tracking who made the change and when it occurred. This is useful for compliance and monitoring purposes.

4. Synchronizing Tables:

• Triggers can synchronize data between tables. For example, when a record is updated in one table, a trigger can update corresponding records in related tables.

5. Preventing Invalid Transactions:

• Triggers can prevent certain operations from occurring if specific conditions are not met. For example, a trigger can prevent an employee record from being deleted if it is linked to active projects.

Scenario: Maintaining an Audit Trail

Suppose you have an Employees table, and you want to keep a history of all changes made to employee records. You can create an audit table and use a trigger to automatically insert a
record into the audit table whenever an employee record is updated.

53. What are the main differences between TSQL and PL/SQL?

S.No	T-SQL	PL/SQL
1.	The full Form of TL-SQL is Transact Structure Query language.	The full Form of PL/SQL is Procedural Language Structural Query Language.
2.	T-SQL was developed by Microsoft.	PL-SQL is developed by Oracle.
3.	T-SQL provides a higher degree of control to the programmers.	It is a natural programming language that is much compatible with the SQL and provides higher functionality.
4.	T-SQL performs best with Microsoft SQL server.	PL-SQL performs best with Oracle database server.

54. Describe a scenario where you would use the SUM aggregate function to calculate total sales for each month.

Query:

```
SELECT
YEAR(SaleDate) AS SalesYear, - - Extracts the year from SaleDate
MONTH(SaleDate) AS SalesMonth, - - Extracts the month from SaleDate
SUM(Amount) AS TotalSales - - Calculates the total sales for the month
FROM Sales GROUP BY
YEAR(SaleDate), - - Groups by year
MONTH(SaleDate) - - Groups by month
ORDER BY
```

```
SalesYear, - - Orders by year
SalesMonth; -- Orders by month
```

55. You have a banking application where transactions must be all-or-nothing. Explain how you would implement atomicity to ensure this.

```
- - Begin transaction
    BEGIN TRANSACTION;
- - Declare variables
    DECLARE @SourceAccountID INT = 1;
    DECLARE @DestinationAccountID INT = 2;
    DECLARE @Amount DECIMAL(18, 2) = 100.00;
- - Perform operations
    BEGIN TRY
-- Deduct amount from source account
    UPDATE Accounts
    SET Balance = Balance - @Amount
    WHERE AccountID = @SourceAccountID;
-- Check if the source account has sufficient balance
    IF (SELECT Balance FROM Accounts WHERE AccountID =
@SourceAccountID) < 0
    BEGIN
- - If not sufficient, rollback and raise an error
    ROLLBACK TRANSACTION;
    RAISERROR ('Insufficient funds in source account.', 16, 1);
    RETURN;
    END
-- Add amount to destination account
    UPDATE Accounts
    SET Balance = Balance + @Amount
    WHERE AccountID = @DestinationAccountID;
-- If all operations succeed, commit the transaction
    COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
-- If an error occurs, rollback the transaction
    ROLLBACK TRANSACTION;
-- Optional: Log error details or rethrow
    THROW;
    END CATCH;
```

56. Imagine you are setting up a new database for an e-commerce website. How would you ensure that only authorized users have access to sensitive customer data?

Ensuring that only authorized users have access to sensitive customer data in an e-commerce website involves a multi-layered approach to security and access control. Here's a comprehensive strategy to achieve this:

1. Define User Roles and Permissions

1. **Identify Roles:**
    - **Administrators:** Full access to all data and system functions.
    - **Customer Support:** Limited access to customer information required for support.
    - **Finance Team:** Access to transaction and billing information.
    - **Developers:** Access to development and testing environments but limited access to production data.
    - **Regular Users:** Access only to their own account and order information.
2. **Assign Permissions:**
    - Create roles with specific permissions that match their responsibilities. For example, customer support roles might have access to view customer profiles but not modify them.

2. Implement Role-Based Access Control (RBAC)

1. **Define Roles and Permissions:**
    - Implement RBAC in your database and application to enforce the access control based on user roles.
2. **Database-Level Permissions:**
    - Use SQL GRANT statements to assign roles and permissions to database users.
    - Example:

```
GRANT SELECT ON customer_data TO support_role;
GRANT SELECT, INSERT, UPDATE ON order_data TO finance_role;
```

3. **Use Encryption**
    1. **Data Encryption:**
        - **At Rest:** Encrypt sensitive data stored in the database to protect it from unauthorized access.
            - Example: Use Transparent Data Encryption (TDE) in SQL Server or similar features in other DBMSs.
        - **In Transit:** Use SSL/TLS to encrypt data transmitted between the client and the server to protect it from eavesdropping.
    2. **Column-Level Encryption:**

- Encrypt sensitive columns like credit card numbers or personal identification numbers (PINs).
- Example in MySQL:

```
ALTER TABLE customer_data
ADD COLUMN encrypted_credit_card VARBINARY(255);
```

4. **Implement Strong Authentication and Authorization**
    1. **Authentication:**
        - Require strong passwords and implement multi-factor authentication (MFA) for accessing sensitive data.
        - Use OAuth, OpenID Connect, or similar standards for user authentication.
    2. **Authorization:**
        - Use fine-grained access controls to restrict what authenticated users can see or modify based on their roles.
        - Implement session management and enforce session timeouts.

5. Auditing and Monitoring

1. **Audit Logs:**
    - Maintain logs of all access to sensitive data and changes made. This includes tracking who accessed what data and when.
    - Example: Enable auditing features in your DBMS or application to capture detailed logs.
2. **Regular Reviews:**
    - Periodically review access logs and permissions to ensure that only authorized users have access and that permissions are up-to-date.

6. Data Masking and Redaction

1. **Data Masking:**
    - Use data masking techniques to obscure sensitive information in non-production environments or during reporting.
    - Example: Mask credit card numbers so that only the last four digits are visible.
2. **Redaction:**
    - Apply redaction to sensitive fields when displaying data to users who do not have the appropriate permissions.

7. Database Security Best Practices

1. **Apply Security Patches:**
    - Regularly update your database management system and related software to address known vulnerabilities.
2. **Backup and Recovery:**
    - Ensure that backups are encrypted and stored securely. Regularly test backup and recovery procedures.
3. **Least Privilege Principle:**
    - Grant users the minimum level of access necessary for their role to minimize the risk of data exposure.

8. Education and Training

1. **User Training:**
    - Provide training to users on data protection policies and the importance of safeguarding sensitive information.
2. **Security Awareness:**
    - Educate users on recognizing phishing attacks, managing passwords securely, and other security best practices.

Example Scenario

For an e-commerce website, you might set up the following roles and permissions:

- **Admin Role:** Full access to all tables and management functions.
- **Support Role:** Access to `customer_profiles` and `order_history` tables with read-only permissions.
- **Finance Role:** Access to `transaction_records` with read and write permissions but no access to `customer_profiles`.
- **Development Role:** Read-only access to non-sensitive data and access to development environments.

Example SQL for Role-Based Access Control

`sql`

```
-- Create roles
CREATE ROLE support_role;
CREATE ROLE finance_role;

-- Grant permissions to roles
```

```
GRANT SELECT ON customer_profiles TO support_role;
GRANT SELECT, INSERT, UPDATE ON transaction_records TO finance_role;

-- Assign roles to users
GRANT support_role TO user_support;
GRANT finance_role TO user_finance;
```

57. A large retail company needs a high-performing OLTP system for processing sales and an OLAP system for analyzing sales data. Explain how you would design and optimize these systems.

OLTP System Design and Optimization:

1. Database Schema Design:

• Normalization: Use normalization techniques to reduce data redundancy and ensure data integrity. Typically, this involves organizing the data into multiple related tables.

• Entity-Relationship Model: Design a schema that captures the business entities (e.g., customers, orders, products) and their relationships. For instance:

• Customers: CustomerID, Name, Address, etc.

• Orders: OrderID, CustomerID, OrderDate, TotalAmount, etc.

• OrderDetails: OrderDetailID, OrderID, ProductID, Quantity, Price, etc.

• Products: ProductID, Name, Category, Price, etc.

2. Indexing:

• Primary Keys: Ensure primary keys are indexed to speed up data retrieval.

• Secondary Indexes: Create secondary indexes on frequently queried fields, such as CustomerID or ProductID, to enhance search performance.

3. Transaction Management:

• ACID Properties: Ensure the system adheres to ACID (Atomicity, Consistency, Isolation, Durability) principles to handle transactions reliably and maintain data integrity.

• Concurrency Control: Implement locking mechanisms or multiversion concurrency control (MVCC) to handle concurrent transactions and prevent issues like dirty reads or lost updates.

4. Performance Optimization:

• Query Optimization: Optimize SQL queries for speed. Use execution plans to understand query performance and adjust indexes or queries accordingly.

• Partitioning: Consider partitioning large tables to improve performance and manageability.

• Caching: Implement caching strategies to reduce database load and improve response times for frequent queries.

5. Scalability:

• Horizontal Scaling: Use database sharding or clustering techniques to distribute the load across multiple servers if necessary.

• Load Balancing: Implement load balancers to evenly distribute database requests and improve availability.

6. Security and Backup:

• Data Security: Implement robust access controls, encryption, and regular security audits to protect sensitive data.

• Backup Strategy: Regularly back up data to prevent loss due to hardware failures or other issues. Consider automated backup solutions.

OLAP System Design and Optimization

1. Database Schema Design:

• Star Schema: Organize data into a central fact table (e.g., Sales) connected to dimension tables (e.g., Time, Product, Customer). This schema is effective for complex queries and reporting.

• Snowflake Schema: A variation where dimension tables are normalized into multiple related tables. This can save storage but might complicate queries.

2. Data Aggregation:

• Pre-Aggregation: Pre-calculate and store aggregate values (e.g., total sales per month) to speed up query responses. Use materialized views or summary tables for this purpose.

• Cube Creation: Design OLAP cubes to aggregate data along various dimensions (e.g., sales by region, time, and product). This facilitates fast multidimensional analysis.

3. Indexing and Storage:

• Bitmap Indexes: Use bitmap indexes on dimension attributes to improve query performance, especially for categorical data.

• Columnar Storage: Store data in columnar format to improve performance for read-heavy operations and aggregations.

4. Query Optimization:

• Query Optimization: Optimize complex queries for performance. Ensure that they are written to leverage the pre-aggregated data and indexing.

• Execution Plans: Analyze execution plans for queries to identify and resolve bottlenecks.

5. Performance Optimization:

• Data Partitioning: Partition large fact tables by date or other dimensions to improve query performance and manageability.

• Caching: Implement caching mechanisms for frequently accessed data or query results to reduce load on the OLAP system.

6. Scalability:

• Data Warehouse Appliances: Consider using data warehouse appliances or cloud-based solutions that offer scalable storage and compute resources for large- scale analytics.

• Distributed Processing: Use distributed computing frameworks (e.g., Apache Hadoop, Apache Spark) for processing and analyzing large datasets.

7. Data Integration and ETL:

• ETL Processes: Design efficient ETL (Extract, Transform, Load) processes to regularly update the OLAP system with data from the OLTP system. Ensure that these processes are optimized for performance and handle data quality issues.

8. Security and Governance:

• Access Controls: Implement role-based access controls to ensure that only authorized users can access sensitive or critical data.

• Data Governance: Establish data governance policies to ensure data quality,
consistency, and compliance with regulations.

58. What is authentication vs autorization?

Authentication	Authorization
In the **authentication** process, the identity of users are checked for providing the access to the system.	While in **authorization** process, a the person's or user's authorities are checked for accessing the resources.
In the authentication process, users or persons are verified.	While in this process, users or persons are validated.
It is done before the authorization process.	While this process is done after the authentication process.
It needs usually the user's login details.	While it needs the user's privilege or security levels.

59. You need to create a new table with columns for EmployeeID, Name, Position, and Salary. Write the SQL statement to create this table. Add unique constraint, primary key accordingly

```
CREATE TABLE Employees (
EmployeeID INT NOT NULL AUTO_INCREMENT,  -- or SERIAL in PostgreSQL,  or IDENTITY in SQL Server
Name VARCHAR(100) NOT NULL,
Position VARCHAR(50),
Salary DECIMAL(10, 2),
PRIMARY KEY (EmployeeID),
UNIQUE (Name)  -- Optional: Ensure that employee names are unique (if required)
);
```

60. A table needs to be altered to add a new column Email, but only if it doesn't already exist. Write the SQL statement to achieve this.

```
-- Check if the column exists
IF NOT EXISTS (
SELECT 1
FROM sys.columns
WHERE Name = 'Email'
AND Object_ID = Object_ID('YourTableName')
)
BEGIN
-- Add the column if it does not exist
EXEC sp_executesql N'
ALTER TABLE YourTableName
ADD Email VARCHAR(255);
';
END
```

61. What is Composite key?

A composite key, also known as a compound key or a concatenated key, is a primary key that consists of two or more columns in a database table. It is used to uniquely identify a record in a table where a single column is not sufficient to provide uniqueness.

```
CREATE TABLE CourseRegistrations (
StudentID INT NOT NULL,
CourseID INT NOT NULL,
RegistrationDate DATE,
PRIMARY KEY (StudentID, CourseID)
);
```

62. Write a query to update the Salary column in the Employees table, increasing all salaries by 10%.

```
UPDATE Employees
```

SET Salary = Salary * 1.10;

63. You need to retrieve data from the Orders and Customers tables to find all orders placed by customers from a specific city. Write the query.

```
SELECT o.OrderID, o.OrderDate, o.OrderAmount, c.CustomerID,
c.CustomerName, c.City
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
WHERE c.City = 'SpecificCityName';
```

64. Write a query to find the average Salary in the Employees table, grouped by Department.

```
SELECT DepartmentID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DepartmentID;
```

65. Describe a scenario where you would use the RANK function to assign ranks to employees based on their sales performance.

The RANK function is a useful SQL analytic function for assigning ranks to rows within a partition of data, often used for ranking employees or items based on specific criteria. It's particularly handy in scenarios where you want to rank employees or items while dealing with ties (i.e., where multiple rows have the same value).

Scenario: Ranking Employees Based on Sales Performance

```
SELECT
e.EmployeeID,
e.EmployeeName,
SUM(s.SaleAmount) AS TotalSales,
RANK() OVER (ORDER BY SUM(s.SaleAmount) DESC) AS SalesRank
FROM
Employees e
JOIN
Sales s ON e.EmployeeID = s.EmployeeID
GROUP BY
e.EmployeeID, e.EmployeeName
ORDER BY
SalesRank;
```

66. Write a query to filter out all products from the Products table that have a Price greater than rs.100.

```
SELECT *
FROM Products
WHERE Price > 100;
```

67. Explain how you would use the CASE statement to filter data based on multiple conditions, such as categorizing products into different price ranges.

```
SELECT
ProductID,
ProductName,
Price,
CASE
WHEN Price <= 50 THEN 'Low'
WHEN Price <= 100 THEN 'Medium'
ELSE 'High'
END AS PriceRange
FROM Products;
```

68. Write a query to find all employees whose Name starts with the letter 'A'.

```
SELECT *
FROM Employees
WHERE Name LIKE 'A%';
```

69. A denormalized database is causing performance issues. Describe the steps you would take to normalize it and improve performance till 3NF

1. Identify the Current Schema

• Review the existing schema to understand its structure and relationships.
• Identify the tables and their columns, and look for redundancy and anomalies.

2. Apply First Normal Form (1NF)

• Ensure that each table has a primary key.
• Eliminate repeating groups or arrays by creating separate rows for each instance.
• Ensure that each column contains atomic (indivisible) values.

3. Apply Second Normal Form (2NF)

• Identify and remove partial dependencies: columns should depend on the whole primary key.
• Create separate tables for sets of related data and establish foreign key relationships.
• Ensure that non-key attributes are fully functionally dependent on the entire primary key.

4. Apply Third Normal Form (3NF)

• Remove transitive dependencies: non-key columns should not depend on other non-key columns.

• Create new tables to hold transitive dependencies and link them with foreign keys.

• Ensure that each non-key attribute is non-transitively dependent on the primary key.

5. Review and Refactor

• Test Performance: Check the performance of the normalized schema with sample queries.

• Optimize Indexing: Add appropriate indexes to improve query performance.

• Update Application Logic: Modify application queries and logic to accommodate the normalized schema.

70. Write a SQL statement to create an index on the Email column of the Users table.

CREATE INDEX idx_email ON Users (Email);

71. Write a query to perform an inner join between the Orders and Customers tables to retrieve all orders along with customer names.

Query:

```
SELECT
o.OrderID,
o.OrderDate,
o.TotalAmount,
c.CustomerID,
c.CustomerName
FROM Orders o
INNER JOIN Customers c ON o.CustomerID = c.CustomerID;
```

72. Describe a scenario where a full outer join would be necessary, and provide a query example.

Scenario: Employee Department Mismatches

Suppose you have two tables: Employees and Departments.

• **Employees Table:** Contains information about employees, including their department assignments.

• **Departments Table**: Contains information about departments, including their status.

You want to generate a report showing all employees and all departments, even if some

employees are not assigned to any department or some departments do not have any employees.

Query:

```
SELECT
e.EmployeeID,
e.EmployeeName,
e.DepartmentID AS EmployeeDepartmentID,
d.DepartmentName,
d.DepartmentID AS DepartmentID
FROM Employees e
FULL OUTER JOIN Departments d ON e.DepartmentID =
d.DepartmentID;
```

73. Write a query using table aliases to simplify a complex join between three tables: Orders, Customers, and Products.

```
SELECT
o.OrderID,
o.OrderDate,
c.CustomerName,
p.ProductName,
o.Quantity,
p.Price
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
JOIN Products p ON o.ProductID = p.ProductID;
```

74. Explain how column aliases can be used in a query with aggregate functions to make the results more readable.

Query:

```
SELECT
ProductID,
SUM(Quantity) AS TotalQuantity,
SUM(SaleAmount) AS TotalSales,
AVG(SaleAmount) AS AverageSaleAmount
FROM Sales
GROUP BY ProductID;
```

75. Provide a scenario where a subquery would be more appropriate than a join, and write the corresponding query.

Scenario:

Suppose you have an e-commerce database with two tables: Orders and Customers.

- Orders: Contains details about customer orders.
  - order_id (Primary Key)

- customer_id
- order_amount
- order_date
- Customers: Contains details about customers.
- customer_id (Primary Key)
- customer_name
- customer_email

You want to find customers who have placed more than 5 orders. In this scenario, a subquery can be more appropriate than a join because you need to filter based on an aggregated
result (the number of orders per customer) rather than combining rows from both tables.
Query:

```
SELECT customer_id, customer_name, customer_email
FROM Customers
WHERE customer_id IN (
SELECT customer_id
FROM Orders
GROUP BY customer_id
HAVING COUNT(order_id) > 5
);
```

76. Compare the performance implications of using a join versus a subquery for retrieving data from large tables.

Joins
Pros:
- Efficient with Indexes: Fast if proper indexes are in place.
- Optimized by DBMS: Modern systems optimize joins well.

Cons:
- Complexity and Cost: Can be expensive with large datasets and no indexes.
- Intermediate Results: May involve costly intermediate data handling.

Subqueries
Pros:
- Granular Filtering: Useful for complex filters and aggregated results.
- Avoids Cartesian Products: Can prevent unnecessary large intermediate results.

Cons:
- Execution Overhead: May be less efficient if the DBMS executes the subquery
repeatedly.
- Less Predictable: Performance can vary based on how the subquery is optimized.

77. Describe a scenario where using a DATETIME data type would be essential, and explain how you would store and retrieve this data.

Storing Data

Table Schema:

```
CREATE TABLE UserActivity (
activity_id INT AUTO_INCREMENT PRIMARY KEY,
user_id INT NOT NULL,
login_time DATETIME NOT NULL,
logout_time DATETIME,
FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

Example Insert Statement:

```
INSERT INTO UserActivity (user_id, login_time, logout_time)
VALUES (1, '2024-07-27 08:30:00', '2024-07-27 10:15:00');
```

Retrieving Data

```
SELECT user_id, login_time, logout_time
FROM UserActivity WHERE DATE(login_time) = '2024-07-27';
```

Query to Calculate Session Durations:

```
SELECT user_id, login_time, logout_time,
TIMESTAMPDIFF(MINUTE, login_time, logout_time) AS
session_duration_minutes
FROM UserActivity
WHERE user_id = 1;
```

78. Explain how you would handle data type conversions when importing data from a CSV file with mixed data types.

- **Understand the CSV File:** Know the structure and data types.
- **Prepare the Schema:** Define the database schema with appropriate data types.
- **Pre-process and Convert Data:** Clean and convert data formats before importing.
- **Handle Errors:** Log and correct conversion issues.
- **Verify and Validate:** Ensure data is correctly imported and valid.

79. Write a query using a correlated subquery to find all employees who have a salary higher than the average salary in their department.

```
SELECT e.EmployeeID, e.Name, e.Salary, e.DepartmentID
FROM Employees e
WHERE e.Salary > (
SELECT AVG(e2.Salary)
FROM Employees e2
WHERE e2.DepartmentID = e.DepartmentID
);
```

80. Explain how non-correlated subqueries can be optimized for better performance compared to correlated subqueries.

Non-Correlated Subqueries

Definition:

• A non-correlated subquery does not reference any columns from the outer query. It is executed once, and its result is used by the outer query.

Correlated Subqueries

Definition:

• A correlated subquery references columns from the outer query and is executed for each row of the outer query. This means it is executed multiple times, once for each row in the outer query.

Key Differences and Performance Considerations:

1. Execution Frequency:

• Non-Correlated Subqueries: Executed once and reused.

• Correlated Subqueries: Executed multiple times, once for each row in the outer query.

2. Optimization Potential:

• Non-Correlated Subqueries: Often more straightforward to optimize due to single execution and result caching.

• Correlated Subqueries: Requires techniques to minimize repeated execution, such as rewriting to joins or using temporary tables.

3. Use Cases:

• Non-Correlated Subqueries: Best for operations where the subquery provides a static value or set of values used by the outer query.

• Correlated Subqueries: Useful for operations where the result depends on the row being processed by the outer query but can often be optimized by rethinking the query structure.

81. Write a simple stored procedure to insert a new record into the Employees table.

Example Table Structure

```
CREATE TABLE Employees (
employee_id INT AUTO_INCREMENT PRIMARY KEY,
employee_name VARCHAR(100) NOT NULL,
department_id INT NOT NULL,
salary DECIMAL(10, 2) NOT NULL,
hire_date DATE NOT NULL
);
```

Stored Procedure Definition

```
DELIMITER //
CREATE PROCEDURE InsertEmployee(
IN p_employee_name VARCHAR(100),
IN p_department_id INT,
```

```
IN p_salary DECIMAL(10, 2),
IN p_hire_date DATE
)
BEGIN
INSERT INTO Employees (employee_name, department_id, salary,
hire_date)
VALUES (p_employee_name, p_department_id, p_salary, p_hire_date);
END //
DELIMITER ;
```

How to Call the Stored Procedure

```
CALL InsertEmployee('John Doe', 2, 55000.00, '2024-07-27');
```

82. Describe a scenario where you would use a trigger to enforce business rules.

Scenario: Enforcing Minimum Salary

Business Rule: Employees must have a salary of at least $30,000.

Using a Trigger:

1. Table Definition:

```
CREATE TABLE Employees (
employee_id INT AUTO_INCREMENT PRIMARY KEY,
employee_name VARCHAR(100) NOT NULL,
department_id INT NOT NULL,
salary DECIMAL(10, 2) NOT NULL,
hire_date DATE NOT NULL
);
```

2.Create Triggers:

```
DELIMITER //

CREATE TRIGGER CheckSalaryBeforeInsert
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
IF NEW.salary < 30000.00 THEN
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Salary cannot be less than $30,000.';
END IF;
END //
CREATE TRIGGER CheckSalaryBeforeUpdate
BEFORE UPDATE ON Employees
FOR EACH ROW
BEGIN
IF NEW.salary < 30000.00 THEN
SIGNAL SQLSTATE '45000'
```

```
SET MESSAGE_TEXT = 'Salary cannot be less than $30,000.';
END IF;
END //
DELIMITER ;
```

83. Mention any 2 of the common security measures to protect a SQL Server database?

To protect a SQL Server database, you can implement several security measures. Two common ones are:

1. **User Authentication and Authorization**: Ensure that only authorized users have access to the database. This involves using strong authentication methods (such as Windows Authentication or SQL Server Authentication) and managing user permissions carefully. Grant users only the permissions they need to perform their jobs (principle of least privilege) and regularly review and adjust permissions as necessary.
2. **Encryption**: Protect sensitive data both at rest and in transit. For data at rest, you can use Transparent Data Encryption (TDE) to encrypt the database files on disk. For data in transit, use SSL/TLS to encrypt the data transmitted between the client and the SQL Server. This helps protect against unauthorized access and eavesdropping.

Implementing these measures helps ensure that your SQL Server database is better protected from unauthorized access and data breaches.

84. How do you create a user and assign roles in MSSQL?

1. Creating a Login

```
CREATE LOGIN [NewLogin] WITH PASSWORD = 'YourStrongPassword';
```

2. Creating a Database User

```
CREATE USER [NewUser] FOR LOGIN [NewLogin];
```

3. Assigning Roles to the User

```
ALTER ROLE db_datareader ADD MEMBER [NewUser];
ALTER ROLE db_datawriter ADD MEMBER [NewUser];
```

85. Explain how encryption can be implemented for data at rest in MSSQL.

1. Transparent Data Encryption (TDE)

TDE encrypts the entire database, including the data files, log files, and backups. This is done
transparently without requiring changes to the application or the database schema. Here’s how
you can implement TDE:

1. Create a Master Key:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD =
'YourStrongPassword';
```

2. Create a Certificate:

```
CREATE CERTIFICATE MyTDECert
WITH SUBJECT = 'TDE Certificate';
```

3. Create a Database Encryption Key (DEK):

```
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256 ENCRYPTION BY SERVER CERTIFICATE
MyTDECert;
```

4. Enable Encryption on the Database:

```
ALTER DATABASE YourDatabaseName
SET ENCRYPTION ON;
```

5. Backup the Certificate and Master Key:

```
BACKUP CERTIFICATE MyTDECert
TO FILE = 'C:\Backup\MyTDECert.cer'
WITH PRIVATE KEY (
FILE = 'C:\Backup\MyTDECert.pvk',

ENCRYPTION BY PASSWORD = 'YourStrongPassword'
);
BACKUP MASTER KEY
TO FILE = 'C:\Backup\MasterKey.bak'
ENCRYPTION BY PASSWORD = 'YourStrongPassword';
```