Week 1 Assignment

Name : DHANAPAL

Date : 27/7/2024 - 28/7/2024

Aggregate and Atomicity

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

Aggregate functions in SQL perform a calculation on a set of values and return a single value. They are commonly used in conjunction with the `GROUP BY` clause to aggregate data from multiple rows into summary results.

Examples of aggregate functions include:

- `COUNT()`: Counts the number of rows in a set.
- `SUM()`: Calculates the sum of a numeric column.
- `AVG()`: Computes the average value of a numeric column.
- `MIN()`: Returns the minimum value in a set.
- `MAX()`: Returns the maximum value in a set.

Example:

```
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;
```

In this example, `COUNT(*)` is an aggregate function that counts the number of employees in each department.

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

The `GROUP BY` clause is used to arrange identical data into groups. When used with aggregate functions, it allows you to perform calculations on each group of data.

Example:

```
SELECT department, AVG(salary) AS average_salary
FROM employees
GROUP BY department;
```

Here, the `GROUP BY` clause groups employees by their department, and `AVG(salary)` calculates the average salary within each department.

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

Atomicity is a fundamental property of database transactions that ensures that a series of operations within a transaction are completed successfully as a whole, or not executed at all. If any part of the transaction fails, the entire transaction is rolled back.

Scenario: Consider a banking system where you need to transfer money from one account to another. This operation involves two steps:

1. Deducting the amount from the sender's account.
2. Adding the same amount to the recipient's account.

If atomicity is not maintained and the system crashes after the amount is deducted from the sender's account but before it is added to the recipient's account, the money could be lost. Ensuring atomicity guarantees that either both operations succeed or neither does, thus preserving the integrity of the transaction.

OLAP and OLTP

4. Mention any 2 of the differences between OLAP and OLTP

OLAP (Online Analytical Processing) and **OLTP (Online Transaction Processing)** serve different purposes and have distinct characteristics:

1. **Purpose and Use Cases**:
    - **OLAP**: Designed for complex queries and analysis. It is used for data mining, business intelligence, and decision-making. OLAP systems support multidimensional analysis and are optimized for read-heavy operations. For example, generating sales reports and analyzing market trends.
    - **OLTP**: Designed for managing day-to-day operations and transactions. It supports a high volume of short, quick transactions and is optimized for

read and write operations. For example, processing customer orders or updating inventory levels.

2. **Data Structure and Storage**:
    - **OLAP**: Uses multidimensional data structures such as data cubes. Data is often aggregated and pre-computed to speed up complex queries and reporting. It focuses on historical data and trends.
    - **OLTP**: Uses a normalized data structure to reduce redundancy and ensure data integrity. Data is typically current and transaction-focused, with frequent updates and inserts.

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

Optimization techniques for OLTP databases often focus on improving transaction speed and efficiency. One key method is indexing:

- **Indexing**: Creating indexes on columns that are frequently used in queries, joins, or filters can significantly speed up data retrieval operations. Indexes allow the database to quickly locate and access the rows needed for a query without scanning the entire table.

Additional optimization techniques for OLTP databases include:

- **Query Optimization**: Writing efficient SQL queries and avoiding complex joins and subqueries where possible.
- **Database Normalization**: Ensuring that the database schema is normalized to reduce data redundancy and improve data integrity.
- **Proper Hardware Resources**: Ensuring that the database server has adequate memory, CPU, and storage to handle the load.
- **Regular Maintenance**: Performing routine database maintenance tasks such as updating statistics, rebuilding indexes, and cleaning up logs.

Data Encryption and Storage

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

Transparent Data Encryption (TDE): Encrypts entire database files at rest, including data and log files.
Always Encrypted: Encrypts sensitive data on the client side, so the server never sees plaintext data.

Cell-Level Encryption: Encrypts specific columns in a table, protecting individual pieces of data.
Backup Encryption: Encrypts database backups to secure them from unauthorized access.

SQL, NOSQL, Applications, Embedded

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

SQL Databases:

- **Data Model**: Relational. Data is organized into tables with rows and columns. Relationships between tables are defined using keys.
- **Schema**: Fixed schema. The structure of the data (tables, columns, data types) must be defined before data is inserted.
- **Query Language**: Uses Structured Query Language (SQL) for defining and manipulating data.
- **Transactions**: Supports ACID (Atomicity, Consistency, Isolation, Durability) properties to ensure reliable transactions.
- **Use Case**: Suitable for applications with complex queries and transactions, such as financial systems and enterprise applications.

NoSQL Databases:

- **Data Model**: Non-relational. Data can be organized in various formats such as key-value pairs, documents, columns, or graphs.
- **Schema**: Flexible or schema-less. The data structure can be adjusted or changed on the fly.
- **Query Language**: Uses different query methods depending on the type of NoSQL database (e.g., MongoDB uses JSON-like queries).
- **Transactions**: Often focus on eventual consistency rather than ACID, optimizing for scalability and performance.
- **Use Case**: Suitable for applications with large volumes of data, unstructured data, or requiring high scalability, such as social networks and big data applications.

DDL

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

To create a new schema in MSSQL, use the `CREATE SCHEMA` statement:

```
CREATE SCHEMA schema_name;
```

This creates a new schema in the current database. You can also specify authorization to a specific user:

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

9. Describe the process of altering an existing table.

To alter an existing table in MSSQL, use the `ALTER TABLE` statement. This can be used to add, modify, or delete columns, constraints, and indexes.

Examples:

Add a new column:

```
ALTER TABLE table_name

ADD column_name datatype;
```

Modify an existing column:

```
ALTER TABLE table_name

ALTER COLUMN column_name datatype;
```

- **Drop a column**:

```
ALTER TABLE table_name

DROP COLUMN column_name;
```

- **Add a constraint**:

```
ALTER TABLE table_name

ADD CONSTRAINT constraint_name constraint_type (column_name);
```

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

- **Table**:
  - **Definition**: A table is a physical object in the database that stores data in rows and columns.
  - **Storage**: Data is physically stored in the table.
  - **Modification**: Tables can be directly modified by inserting, updating, or deleting data.
- **View**:
  - **Definition**: A view is a virtual table that presents data from one or more tables. It does not store data itself but shows a result set based on a query.
  - **Storage**: Views do not store data; they dynamically retrieve data from underlying tables.
  - **Modification**: Views can be used to simplify complex queries and provide a layer of security by restricting access to specific data.

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

Create an Index:

```
CREATE INDEX index_name

ON table_name (column_name);
```

- **Create a Unique Index** (ensures all values in the indexed column are unique):

```
  CREATE UNIQUE INDEX index_name

ON table_name (column_name);
```

- **Drop an Index**:

```
  DROP INDEX index_name ON table_name;
```

- **Manage Indexes**:

Rebuild an Index: Reorganizes the index to improve performance and reclaim space.

```
ALTER INDEX index_name ON table_name REBUILD;
```

- **Reorganize an Index**: Defragments the index without locking the table.

  ```
  ALTER INDEX index_name ON table_name REORGANIZE;
  ```

- **View Indexes**: You can view existing indexes using system views or catalog views like `sys.indexes`.

DML

12. What are the most commonly used DML commands?

DML (Data Manipulation Language) commands are used to interact with and modify data within a database. The most commonly used DML commands are:

SELECT: Retrieves data from one or more tables.

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

- **INSERT**: Adds new rows to a table.

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

- **UPDATE**: Modifies existing rows in a table.

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

- **DELETE**: Removes rows from a table.

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

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

To retrieve data from multiple tables, you use the `JOIN` clause in a `SELECT` statement. There are several types of joins:

Inner Join: Returns only the rows with matching values in both tables.

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

- **Left Join (or Left Outer Join)**: Returns all rows from the left table and the matched rows from the right table. Rows in the left table with no match in the right table will have `NULL` for columns from the right table.

  ```
  SELECT columns FROM table1 LEFT JOIN table2 ON
  table1.common_column = table2.common_column;
  ```
- **Right Join (or Right Outer Join)**: Returns all rows from the right table and the matched rows from the left table. Rows in the right table with no match in the left table will have `NULL` for columns from the left table.

  ```
  SELECT columns FROM table1 RIGHT JOIN table2 ON
  table1.common_column = table2.common_column;
  ```
- **Full Join (or Full Outer Join)**: Returns all rows when there is a match in one of the tables. Rows with no match in either table will have `NULL` for columns from the other table.

  ```
  SELECT columns FROM table1 FULL JOIN table2 ON
  table1.common_column = table2.common_column;
  ```

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

Relational algebra is a theoretical framework used to describe queries on relational databases. SQL queries are based on relational algebra operations, which include:

Selection (σ): Filters rows based on a condition.

```
SELECT * FROM table_name WHERE condition;
```

- **Projection (π)**: Selects specific columns from a table.

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

Union (∪): Combines results from two queries, eliminating duplicates.

```
SELECT column1 FROM table1
```

```
UNION
SELECT column1 FROM table2;
```
Difference (−): Returns rows from the first query that are not in the second query.
```
SELECT column1 FROM table1
EXCEPT SELECT column1 FROM table2;
```
Cartesian Product (×): Combines all rows from one table with all rows from another table.
```
SELECT * FROM table1, table2;
```

- **Join (⋈)**: Combines rows from two tables based on a related column.

  ```
  SELECT * FROM table1 JOIN table2 ON table1.column =
  table2.column;
  ```

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

Complex queries can have several implications for performance:

- **Execution Time**: Complex queries may take longer to execute due to multiple joins, subqueries, and aggregations.
- **Resource Utilization**: They can consume more CPU, memory, and I/O resources, impacting the performance of the database server.
- **Index Usage**: Proper indexing can help optimize complex queries, but poorly designed indexes can also lead to inefficiencies.
- **Query Optimization**: Complex queries often require careful optimization. Use techniques like query refactoring, indexing, and analyzing execution plans to improve performance.
- **Locking and Blocking**: Complex queries can lead to locking and blocking issues if they involve large data sets or long-running operations.

Aggregate Functions

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

Difference Between `HAVING` and `WHERE` Clauses

- **`WHERE` Clause**:
  - Filters rows **before** grouping and aggregation.

- Cannot use aggregate functions.
  `SELECT SalesPerson, Sale AmountFROM Sales WHERE SaleDate >= '2024-01-01';`

- **`HAVING` Clause**:
  - Filters groups **after** grouping and aggregation.
  - Used with aggregate functions.
    `SELECT SalesPerson, SUM(SaleAmount) AS TotalSales FROM Sales GROUP BY SalesPerson HAVING SUM(SaleAmount) > 10000;`
- **Order of Clauses**

1. `FROM` (specify the table)
2. `WHERE` (filter rows)
3. `GROUP BY` (group rows)
4. `HAVING` (filter groups)
5. `SELECT` (specify columns)
6. `ORDER BY` (sort results)

You **cannot** use `HAVING` before `GROUP BY`.

Correct Example:

```
SELECT SalesPerson, SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE SaleDate >= '2024-01-01'
GROUP BY SalesPerson
HAVING SUM(SaleAmount) > 10000;
```

Filters

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

Filters in SQL are conditions applied to data retrieval queries to limit the rows returned based on specified criteria. They are used to extract only the relevant data needed for a particular query.

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

The `WHERE` clause is used to specify the conditions that must be met for the rows to be included in the result set. It can filter data based on various conditions, such as comparisons, ranges, and patterns.

Example:

```
SELECT * FROM Employees WHERE Age > 30;
```

This query retrieves all employees older than 30.

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

You can combine multiple filter conditions using logical operators like `AND`, `OR`, and `NOT`.

`AND`: All conditions must be true.

```
SELECT * FROM Employees WHERE Age > 30 AND Department = 'Sales';
```

- **`OR`**: At least one condition must be true.

  ```
  SELECT * FROM Employees WHERE Age > 30 OR Department =
  'Sales';
  ```
- **`NOT`**: Negates a condition.

  ```
  SELECT * FROM Employees WHERE NOT Department = 'Sales';
  ```

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

The `CASE` statement allows for conditional logic within SQL queries. It can be used to create computed columns, to conditionally filter data, or to apply different conditions within a query.

Example:

```
SELECT

    EmployeeID,

    Name,

    CASE
```

```
        WHEN Age < 30 THEN 'Young'
        WHEN Age BETWEEN 30 AND 50 THEN 'Middle-aged'
        ELSE 'Senior'
    END AS AgeGroup
FROM Employees;
```

This query classifies employees into age groups based on their age.

Using `CASE` in a `WHERE` clause:

```
SELECT *
FROM Employees
WHERE
    CASE
        WHEN Department = 'Sales' THEN Salary > 50000
        ELSE Salary > 30000
    END;
```

This query filters employees based on department-specific salary criteria using a `CASE` statement.

Operators

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

In MSSQL, operators are symbols or keywords that specify the type of operation to be performed on one or more expressions. The main types of operators include:

- **Arithmetic Operators**: Perform mathematical operations.
  - `+` (Addition)
  - `-` (Subtraction)
  - `*` (Multiplication)

    - `/` (Division)
    - `%` (Modulus)
- **Comparison Operators**: Compare two expressions.
    - `=` (Equal to)
    - `>` (Greater than)
    - `<` (Less than)
    - `>=` (Greater than or equal to)
    - `<=` (Less than or equal to)
    - `<>` or `!=` (Not equal to)
- **Logical Operators**: Combine multiple conditions.
    - `AND`
    - `OR`
    - `NOT`
- **Bitwise Operators**: Perform bitwise operations on integer types.
    - `&` (Bitwise AND)
    - `|` (Bitwise OR)
    - `^` (Bitwise XOR)
    - `~` (Bitwise NOT)
    - `<<` (Left shift)
    - `>>` (Right shift)
- **String Concatenation Operator**: Combine strings.
    - `+` (Concatenates two strings)
- **Assignment Operators**: Assign values to variables.
    - `=` (Assign a value)

22. How do arithmetic operators work in SQL?

Arithmetic operators perform mathematical operations on numeric data types. They can be used in `SELECT` statements, `WHERE` clauses, and other parts of SQL queries.

Examples:

Addition:
```
SELECT Price + Tax AS TotalPrice FROM Products;
```

- **Subtraction**:

```
SELECT Price - Discount AS SalePrice FROM Products;
```

- **Multiplication**:

```
SELECT Quantity * UnitPrice AS TotalCost FROM OrderDetails;
```

- **Division**:

```
SELECT TotalAmount / NumberOfItems AS AveragePrice FROM
Sales;
```

- **Modulus**:

```
SELECT EmployeeID % 2 AS Remainder FROM Employees;
```

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

The `LIKE` operator is used in SQL to search for a specified pattern in a column. It is often used with wildcard characters to perform flexible pattern matching.

Wildcards:

- `%`: Represents zero, one, or multiple characters.
- `_`: Represents a single character.

Examples:

- **Using `%` wildcard**:

Find names starting with "A":

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

    - Find names ending with "n":

```
SELECT * FROM Customers WHERE Name LIKE '%n';
```

    - Find names containing "an":

```
SELECT * FROM Customers WHERE Name LIKE '%an%';
```

- **Using _ wildcard**:

Find names with exactly five characters:

```
SELECT * FROM Customers WHERE Name LIKE '_____';
```

- Find names where the second character is "a":

```
SELECT * FROM Customers WHERE Name LIKE '_a%';
```

The `LIKE` operator is particularly useful for pattern matching in text data, allowing for flexible and powerful search capabilities in SQL queries.

Multiple Tables: Normalization

24. What is normalization and why is it important?

Normalization is the process of organizing a database to reduce redundancy and improve data integrity. It involves dividing large tables into smaller, related tables and defining relationships between them.

Importance:

- **Reduces data redundancy**: Ensures that the same piece of data is not stored in multiple places.
- **Improves data integrity**: Enforces data consistency and accuracy.
- **Enhances query performance**: Simplifies complex queries by breaking down tables into simpler, related structures.

25. Describe the basic normal forms.

First Normal Form (1NF):

- Ensures that each column contains atomic (indivisible) values and each row has a unique identifier (primary key).

Second Normal Form (2NF):

- Meets all requirements of 1NF.
- Ensures that all non-key columns are fully dependent on the primary key.

Third Normal Form (3NF):

- Meets all requirements of 2NF.
- Ensures that all non-key columns are not only fully dependent on the primary key but also independent of each other (no transitive dependency).

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

Impact on Database Performance:

- **Improved Data Retrieval**: Normalization can lead to more efficient data retrieval by minimizing data redundancy and reducing the size of the database. However, it can also introduce complexity in query joins, which might affect performance in highly normalized databases.

Indexes & Constraints

27. What are indexes and why are they used?

Indexes are database objects that improve the speed of data retrieval operations on a table at the cost of additional storage and maintenance overhead. They are used to quickly locate and access the data without having to search every row in a database table.

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

A unique constraint ensures that all values in a column are distinct. To create a unique constraint on a table column:

```
ALTER TABLE table_name

ADD CONSTRAINT constraint_name UNIQUE (column_name);
```

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

- **Clustered Index**:
    - **Structure**: Alters the physical order of the table and sorts the rows based on the indexed column.
    - **Table**: A table can have only one clustered index.
    - **Usage**: Typically used for primary keys.

- **Non-Clustered Index**:
    - **Structure**: Creates a separate object within the table that points to the physical rows.
    - **Table**: A table can have multiple non-clustered indexes.
    - **Usage**: Used for frequently searched columns that are not the primary key.

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

Optimizing Index Usage:

- **Index Selectivity**: Create indexes on columns with high selectivity (where the number of distinct values is high).
- **Composite Indexes**: Use composite indexes for queries that filter on multiple columns.
- **Index Maintenance**: Regularly rebuild and reorganize indexes to minimize fragmentation.
- **Avoid Over-Indexing**: Only create necessary indexes to avoid performance overhead on insert, update, and delete operations.
- **Monitor Performance**: Use database monitoring tools to analyze query performance and adjust indexing strategy accordingly.

Joins

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

- **Inner Join**: Returns rows that have matching values in both tables.
- **Left Join (Left Outer Join)**: Returns all rows from the left table and the matched rows from the right table; unmatched rows from the right table will have NULL values.
- **Right Join (Right Outer Join)**: Returns all rows from the right table and the matched rows from the left table; unmatched rows from the left table will have NULL values.
- **Full Outer Join**: Returns all rows when there is a match in one of the tables. Rows with no match in either table will have NULL values.
- **Cross Join**: Returns the Cartesian product of both tables.
- **Self Join**: Joins a table to itself.

32. Provide an example of a LEFT JOIN query.

```
SELECT Employees.EmployeeID, Employees.Name, Orders.OrderID
```

```
FROM Employees

LEFT JOIN Orders ON Employees.EmployeeID = Orders.EmployeeID;
```

This query retrieves all employees and their orders, showing NULL for employees who have no orders.

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

A **self-join** is a regular join but the table is joined with itself. It's used to compare rows within the same table.

Example:

```
SELECT A.EmployeeID, A.Name AS EmployeeName, B.Name AS
ManagerName

FROM Employees A

JOIN Employees B ON A.ManagerID = B.EmployeeID;
```

This query retrieves employees and their managers from the same `Employees` table.

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

A **full outer join** combines the results of both left and right outer joins. It returns all rows from both tables, with NULLs where there is no match.

Example:

```
SELECT Employees.EmployeeID, Employees.Name, Orders.OrderID

FROM Employees

FULL OUTER JOIN Orders ON Employees.EmployeeID =
Orders.EmployeeID;
```

This query retrieves all employees and all orders, showing NULLs where there is no match.

Significance:

- Useful for getting a complete set of data from both tables, including rows that don't have corresponding matches in the other table

Alias

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

An **alias** in SQL is a temporary name given to a table or column for the duration of a query. It makes the query easier to read and write, especially when dealing with complex queries or when the table/column names are long.

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

```
SELECT E.Name, O.OrderID

FROM Employees E

JOIN Orders O ON E.EmployeeID = O.EmployeeID;
```

In this example, `E` is an alias for `Employees`, and `O` is an alias for `Orders`.

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

```
SELECT Department, COUNT(*) AS EmployeeCount, AVG(Salary) AS
AverageSalary

FROM Employees

GROUP BY Department;
```

Here, `EmployeeCount` and `AverageSalary` are column aliases for the results of the `COUNT` and `AVG` functions, respectively.

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

Benefits of using aliases:

- **Improves Readability**: Simplifies the query, making it easier to understand.
- **Shortens Long Names**: Reduces the need to repeatedly type long table or column names.

- **Clarifies Join Conditions**: Makes it clearer which tables are being joined.
- **Avoids Ambiguity**: Helps distinguish between columns with the same name from different tables.

Joins vs Subqueries

39. What is the difference between joins and subqueries?

Joins: Combine rows from two or more tables based on a related column. They can retrieve data from multiple tables in a single query.
Example:

```
SELECT Employees.Name, Orders.OrderID

FROM Employees

JOIN Orders ON Employees.EmployeeID = Orders.EmployeeID;
```

- **Subqueries**: Nested queries within a main query that return data to be used by the main query. They can be used to filter, compute, or retrieve data.
  Example:

  ```
  SELECT Name FROM Employees WHERE EmployeeID IN (SELECT
  EmployeeID FROM Orders WHERE OrderDate > '2024-01-01');
  ```

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

Prefer subqueries:

- When you need to filter results based on aggregate values or complex conditions.
- When you want to avoid retrieving and joining large datasets, especially in cases where a subquery simplifies the query structure.

41. Explain how correlated subqueries work with an example.

A **correlated subquery** is a subquery that references columns from the outer query. It is executed for each row processed by the outer query.

Example:

```
SELECT E.Name FROM Employees E WHERE EXISTS ( SELECT 1 FROM
Orders O  WHERE O.EmployeeID = E.EmployeeID AND O.OrderDate >
'2024-01-01');
```

In this example, the subquery checks for each employee whether they have orders placed after January 1, 2024.

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

- **Joins**:
    - **Performance**: Generally more efficient for combining rows from multiple tables, especially with proper indexing.
    - **Use Case**: Preferred for retrieving related data from multiple tables in a single result set.
- **Subqueries**:
    - **Performance**: Can be less efficient, especially if the subquery is executed multiple times or involves large datasets.
    - **Use Case**: Useful for filtering or computations that are not easily handled by joins.

Data Types

43. What are the different data types available in MSSQL?
MSSQL offers various data types including:

- **Numeric Types**: `INT`, `BIGINT`, `SMALLINT`, `TINYINT`, `DECIMAL`, `NUMERIC`, `FLOAT`, `REAL`.
- **Character Types**: `CHAR`, `VARCHAR`, `TEXT`, `NCHAR`, `NVARCHAR`, `NTEXT`.
- **Date and Time Types**: `DATE`, `DATETIME`, `DATETIME2`, `TIME`, `SMALLDATETIME`, `DATETIMEOFFSET`.
- **Binary Types**: `BINARY`, `VARBINARY`, `IMAGE`.
- **Other Types**: `BIT`, `UNIQUEIDENTIFIER`, `XML`, `JSON`.

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

- **Data Requirements**: Choose a type based on the nature of the data (e.g., `VARCHAR` for variable-length strings, `INT` for integers).

- **Size and Range**: Select a type that fits the size and range requirements (e.g., `DECIMAL` for precise numbers, `BIGINT` for large integers).
- **Performance**: Opt for types that minimize storage and optimize performance (e.g., `CHAR` for fixed-length, `VARCHAR` for variable-length).
- **Constraints**: Consider constraints and data validation needs (e.g., `DATE` for dates, `BIT` for boolean values).

45. How do you handle data type conversions in queries?
Use SQL functions to convert between data types:

- **Implicit Conversion**: SQL Server automatically converts data types in certain operations.

CAST() Function: Converts an expression to a specified data type.

```
SELECT CAST(Price AS DECIMAL(10, 2)) AS PriceDecimal

FROM Products;
```

- **CONVERT() Function**: Converts an expression to a specified data type with formatting options.
  ```
  SELECT CONVERT(VARCHAR, OrderDate, 101) AS FormattedDate FROM Orders;
  ```

Correlation and Uncorrelation

46. What is a correlated subquery?
A **correlated subquery** references columns from the outer query and is executed once for each row processed by the outer query.
Example:

```
SELECT E.Name FROM Employees E WHERE EXISTS (

  SELECT 1  FROM Orders O WHERE O.EmployeeID = E.EmployeeID AND O.OrderDate > '2024-01-01' );
```

47 .What is a non-correlated subquery?
A **non-correlated subquery** is independent of the outer query. It executes once and returns a result set that is used by the outer query.
Example:

```
SELECT Name FROM Employees WHERE EmployeeID IN (
```

```
  SELECT EmployeeID  FROM Orders  WHERE OrderDate > '2024-01-01'
);
```

48.Explain how correlated subqueries can affect query performance.
Correlated subqueries can be less efficient because they are executed multiple times—once for each row of the outer query. This can lead to performance issues, especially with large datasets, as it increases the number of executions and resource consumption

Introduction to T SQL, Procedures, Functions, Triggers, Indices

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

T-SQL (Transact-SQL) is an extension of SQL used in Microsoft SQL Server. It adds procedural programming capabilities, including:

- **Control-of-Flow**: Constructs like `BEGIN`, `END`, `IF`, `ELSE`, `WHILE`, `GOTO`, etc.
- **Error Handling**: Features like `TRY`, `CATCH`.
- **Transaction Control**: Statements such as `BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`.
- **Additional Functions**: Custom functions and system functions for enhanced data manipulation and control.

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

A **stored procedure** is a precompiled collection of one or more SQL statements. To create a stored procedure:

```
CREATE PROCEDURE ProcedureName

AS BEGIN

       SELECT * FROM TableName;

END;
```

Example:

```
CREATE PROCEDURE GetEmployeeDetails
```

```
    @EmployeeID INT
AS
BEGIN
    SELECT * FROM Employees
    WHERE EmployeeID = @EmployeeID;
END;
```

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

- **Functions**:
  - **Purpose**: Return a single value or a table.
  - **Usage**: Can be used in queries, as part of expressions.
  - **Syntax**: `CREATE FUNCTION FunctionName (...) RETURNS DataType AS BEGIN ... END;`

Example:

```
CREATE FUNCTION GetEmployeeName (@EmployeeID INT)
RETURNS VARCHAR(100)
AS
BEGIN
    DECLARE @Name VARCHAR(100);
    SELECT @Name = Name FROM Employees WHERE EmployeeID = @EmployeeID;
    RETURN @Name;
END;
```

- **Procedures**:
  - **Purpose**: Perform actions, like modifying data or running a complex logic.

- **Usage**: Cannot be used directly in queries; executed using `EXEC` or `CALL`.
- **Syntax**: `CREATE PROCEDURE ProcedureName AS BEGIN ... END;`

Example:

```
CREATE PROCEDURE UpdateEmployeeSalary

    @EmployeeID INT,

    @NewSalary DECIMAL(10, 2)

AS

BEGIN

    UPDATE Employees

    SET Salary = @NewSalary

    WHERE EmployeeID = @EmployeeID;

END;
```

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

Triggers are special types of stored procedures that automatically execute in response to certain events on a table or view, such as `INSERT`, `UPDATE`, or `DELETE`.

Example Scenario:

- **Use Case**: Automatically log changes to a table.

Trigger Example:

```
CREATE TRIGGER LogEmployeeChanges

ON Employees

AFTER UPDATE

AS
```

```
BEGIN

    INSERT INTO EmployeeChangeLog (EmployeeID, ChangeDate,
OldSalary, NewSalary)

    SELECT i.EmployeeID, GETDATE(), d.Salary, i.Salary

    FROM inserted i

    JOIN deleted d ON i.EmployeeID = d.EmployeeID;

END;
```

This trigger logs changes to employee salaries, recording the old and new salary values whenever an update occurs.

Comparison input on TSQL with PL/SQL

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

1. Platform:

- **T-SQL**: Used with Microsoft SQL Server.
- **PL/SQL**: Used with Oracle Database.

2. Procedural Extensions:

- **T-SQL**: Adds procedural elements like `BEGIN`, `END`, `IF`, `ELSE`, `WHILE`, `TRY`, `CATCH`.
- **PL/SQL**: Provides extensive procedural capabilities with blocks (`BEGIN...END`), exception handling (`EXCEPTION`), cursors, and packages.

3. Error Handling:

- **T-SQL**: Uses `TRY...CATCH` blocks for error handling.
- **PL/SQL**: Uses `EXCEPTION` blocks for error handling.

4. Syntax and Functions:

- **T-SQL**: Includes functions like `GETDATE()`, `DATEPART()`, and `ROW_NUMBER()`.
- **PL/SQL**: Includes functions like `SYSDATE`, `TO_DATE()`, and `ROWNUM`.

5. Variables and Constants:

- **T-SQL**: Declares variables with `DECLARE`, sets values with `SET`, and can use `TABLE VARIABLES`.
- **PL/SQL**: Uses `DECLARE` for variables, supports constants, and has robust cursor handling.

6. Package Support:

- **T-SQL**: Does not support packages but has procedures and functions.
- **PL/SQL**: Supports packages, which group related procedures and functions together.

7. Procedural Control:

- **T-SQL**: Control-of-flow statements include `GOTO`, `WAITFOR`.
- **PL/SQL**: Provides more advanced procedural control features such as `LOOP`, `EXIT`, and `CONTINUE`.

Aggregate and Atomicity

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

Scenario: You have a `Sales` table that records each sale with a `SaleDate` and `Amount` column. To calculate the total sales for each month, you can use the `SUM` function with a `GROUP BY` clause to aggregate sales amounts by month.

SQL Query:

```
SELECT

    MONTH(SaleDate) AS SaleMonth,

    YEAR(SaleDate) AS SaleYear,

    SUM(Amount) AS TotalSales

FROM Sales

GROUP BY MONTH(SaleDate), YEAR(SaleDate);
```

This query sums the `Amount` for each month and year, giving you the total sales for each month.

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

Implementation of Atomicity: To ensure transactions are all-or-nothing, you use database transactions. A transaction is a sequence of operations performed as a single unit. Atomicity ensures that either all operations succeed or none are applied.

Example:

```
BEGIN TRANSACTION;

UPDATE Account SET Balance = Balance - 100 WHERE AccountID = 1;

UPDATE Account SET Balance = Balance + 100 WHERE AccountID = 2;

IF @@ERROR <> 0

BEGIN

    ROLLBACK TRANSACTION;

END

ELSE

BEGIN

    COMMIT TRANSACTION;

END
```

In this example, if any operation fails, the `ROLLBACK` ensures that no changes are applied, maintaining the all-or-nothing property. If all operations succeed, the `COMMIT` applies all changes.

Security and Accessibility

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 Authorized Access to Sensitive Customer Data:

1. **User Roles and Permissions**:
    - **Create Roles**: Define specific roles (e.g., Admin, Support, Analyst) with appropriate permissions.
    - **Grant Permissions**: Assign permissions to these roles to control access to sensitive data.
2. **Authentication**:
    - **Use Strong Authentication**: Implement strong password policies and multi-factor authentication (MFA) for accessing the database.
3. **Access Control**:
    - **Limit Access**: Ensure that only authorized users have access to sensitive tables and data. Use the principle of least privilege, granting only necessary permissions.
4. **Data Encryption**:
    - **Encrypt Data**: Use encryption for sensitive data both at rest (in the database) and in transit (during transmission).
5. **Audit Logging**:
    - **Enable Auditing**: Track and log access to sensitive data to monitor for unauthorized access or suspicious activities.
6. **Regular Reviews**:
    - **Review Permissions**: Regularly review and update user roles and permissions to ensure they align with current access needs.

These measures help secure sensitive customer data by controlling who can access it and ensuring that access is appropriately monitored and managed.

MSSQL Install and Configure, OLAP and OLTP

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?

Design and Optimization for OLTP and OLAP Systems:

OLTP System Design and Optimization

1. **Design**:
    - **Schema**: Use a normalized schema (3NF) to reduce redundancy and ensure data integrity.

    - **Tables**: Focus on transactional tables such as `Sales`, `Customers`, `Orders`, `Payments`.
2. **Optimization**:
    - **Indexes**: Create indexes on columns frequently used in `WHERE` clauses and joins to speed up queries.
    - **Query Optimization**: Use stored procedures and parameterized queries to improve performance.
    - **Transactions**: Use proper transaction handling (`BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`) to ensure data consistency.
    - **Concurrency Control**: Implement locking mechanisms and isolation levels to handle concurrent transactions efficiently.

OLAP System Design and Optimization

1. **Design**:
    - **Schema**: Use a star or snowflake schema for denormalized data to optimize for query performance.
    - **Tables**: Create fact tables (e.g., `SalesFact`) and dimension tables (e.g., `DateDimension`, `ProductDimension`).
2. **Optimization**:
    - **Indexes**: Use bitmap indexes or aggregated indexes to speed up read-heavy analytical queries.
    - **Partitioning**: Partition large fact tables by time or other criteria to enhance query performance.
    - **Data Warehousing**: Consider using an OLAP cube or data warehouse for pre-aggregated data and multidimensional analysis.
    - **Caching**: Implement caching mechanisms to store frequently accessed data and improve query response times.

This design and optimization ensure that the OLTP system handles high transaction volumes efficiently while the OLAP system supports complex queries and large-scale data analysis effectively.

Data Encryption and Storage

58 .What is authentication vs authorization?

Authentication vs. Authorization:

- **Authentication**: The process of verifying the identity of a user or system. It answers the question, "Who are you?" Examples include logging in with a username and password or using biometric verification.
- **Authorization**: The process of determining what an authenticated user or system is allowed to do. It answers the question, "What can you do?" Examples include granting permissions to access specific resources or perform certain actions based on user roles.

DDL

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?

1. Create a New Table with Constraints

SQL Statement:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Position VARCHAR(50),
    Salary DECIMAL(10, 2),
    CONSTRAINT UQ_EmployeeName UNIQUE (Name)
);
```

- `EmployeeID` **is defined as the primary key.**
- **A unique constraint (**`UQ_EmployeeName`**) is added to ensure** `Name` **is unique.**

60. Alter Table to Add Column if It Doesn't Exist

SQL Statement (requires a conditional check):

```
IF NOT EXISTS (
    SELECT *
```

```
    FROM INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_NAME = 'Employees'
    AND COLUMN_NAME = 'Email'
)
BEGIN
    ALTER TABLE Employees
    ADD Email VARCHAR(255);
END
```

- **This script checks if the `Email` column exists before adding it.**

61. What is a Composite Key?

A composite key is a primary key composed of two or more columns used together to uniquely identify a record in a table. It is used when a single column is not sufficient to uniquely identify each row.

Example:

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

- **In this example, the combination of `OrderID` and `ProductID` forms the composite key.**

DML

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

SQL Query:

```
UPDATE Employees

SET Salary = Salary * 1.10;
```

- **This query increases all salaries by 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?

SQL Query:

```
SELECT O.* FROM Orders O JOIN Customers C ON O.CustomerID =
C.CustomerID WHERE C.City = 'SpecificCity';
```

- **Replace `'SpecificCity'` with the name of the city you are interested in.**

Aggregate Functions

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

Find the Average Salary Grouped by Department

SQL Query:

```
SELECT Department, AVG(Salary) AS AverageSalary FROM Employees

GROUP BY Department;
```

- **This query calculates the average salary for each department.**

65. Use the RANK Function to Assign Ranks Based on Sales Performance

Scenario: You have a `Sales` table with columns `EmployeeID`, `EmployeeName`, and `SalesAmount`. To rank employees based on their total sales:

SQL Query:

```
SELECT  EmployeeID, EmployeeName, SalesAmount, RANK() OVER (ORDER BY SalesAmount DESC) AS SalesRank FROM Sales;
```

- **This query ranks employees by their `SalesAmount`, with the highest sales receiving the top rank.**

Filters

66. Filter Products with Price Greater Than 100

SQL Query:

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

- This query retrieves all products with a price greater than 100.

67. Use the CASE Statement to Categorize Products into Price Ranges

SQL Query:

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

- This query categorizes products into 'Low', 'Medium', or 'High' price ranges based on their price.

Operators

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

SQL Query:

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

- **This query retrieves all employees whose names start with the letter 'A'.**

Multiple Tables: Normalization

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

Steps to Normalize a Denormalized Database up to 3NF

1. **First Normal Form (1NF)**:
    - **Ensure Atomicity**: Make sure all columns contain atomic (indivisible) values.
    - **Remove Duplicate Data**: Eliminate repeating groups and ensure each column contains unique values.
2. **Example**: Separate a table with multiple values in a single column into individual rows.
3. **Second Normal Form (2NF)**:
    - **Eliminate Partial Dependencies**: Ensure all non-key attributes are fully functionally dependent on the entire primary key.
    - **Create Separate Tables**: Split tables to ensure that attributes depend only on the full primary key.
4. **Example**: Move attributes that depend on only part of a composite key to a new table.
5. **Third Normal Form (3NF)**:
    - **Eliminate Transitive Dependencies**: Ensure that non-key attributes are only dependent on the primary key and not on other non-key attributes.
    - **Create Separate Tables**: Further decompose tables to remove transitive dependencies.
6. **Example**: Create separate tables for attributes that depend on other non-key attributes and link them through foreign keys.

Performance Improvement:

- **Indexes**: Add indexes to frequently queried columns.
- **Optimize Queries**: Rewrite queries to leverage the normalized schema efficiently.
- **Regular Maintenance**: Perform regular database maintenance and analyze query performance.

These steps will help organize the database structure efficiently and improve performance by reducing redundancy and ensuring data integrity.

Indexes & Constraints

70. Create an Index on the Email Column:

SQL Statement:

```
CREATE INDEX idx_Email ON Users (Email);
```

- This statement creates an index named `idx_Email` on the `Email` column of the `Users` table to improve query performance.

Joins

71. Inner Join Between Orders and Customers:

SQL Query:

```
SELECT O.OrderID, O.OrderDate, C.CustomerName

FROM Orders O

INNER JOIN Customers C ON O.CustomerID = C.CustomerID;
```

- This query retrieves all orders along with customer names by joining the `Orders` and `Customers` tables on `CustomerID`.

72.Scenario for Full Outer Join:

Scenario: To find all orders and customers, including those with no matches in the other table, such as identifying customers who have not placed orders and orders with no associated customers.

SQL Query:

```
SELECT O.OrderID, O.OrderDate, C.CustomerName

FROM Orders O

FULL OUTER JOIN Customers C ON O.CustomerID = C.CustomerID;]
```

- This query retrieves all records from both `Orders` and `Customers`, including unmatched records from either table.

Alias

73. Query Using Table Aliases for Complex Join:

SQL Query:

```
SELECT O.OrderID, C.CustomerName, P.ProductName

FROM Orders O

INNER JOIN Customers C ON O.CustomerID = C.CustomerID

INNER JOIN Products P ON O.ProductID = P.ProductID;
```

- **Aliases**: `O`, `C`, and `P` simplify references to the `Orders`, `Customers`, and `Products` tables, respectively.

74. Using Column Aliases with Aggregate Functions:

SQL Query:

```
SELECT

    Department,

    AVG(Salary) AS AverageSalary

FROM Employees

GROUP BY Department;
```

- **Column Alias**: `AverageSalary` makes the result of the `AVG(Salary)` function more readable.

Joins vs. Subqueries

75. Scenario Where a Subquery is More Appropriate:

Scenario: Finding employees who earn more than the average salary of their department.

SQL Query:

```
SELECT EmployeeID, Name, Salary
```

```
FROM Employees E

WHERE Salary > (

    SELECT AVG(Salary)

    FROM Employees

    WHERE Department = E.Department

);
```

- **Subquery**: Used to compare each employee's salary with the average salary of their own department.

76.Performance Implications:

- **Joins**: Generally more efficient for large tables because they can leverage indexes and perform direct comparisons.
- **Subqueries**: Can be slower, especially if correlated, because they may execute multiple times, but useful for complex filtering.

In summary, joins are typically preferred for performance, while subqueries offer flexibility for specific filtering scenarios.

Types

77. Scenario for Using DATETIME Data Type

Scenario: Tracking the exact date and time when a customer places an order in an e-commerce system.

Storing Data:

```
CREATE TABLE Orders (

    OrderID INT PRIMARY KEY,

    OrderDate DATETIME,

    CustomerID INT,

    TotalAmount DECIMAL(10, 2)
```

```
);

INSERT INTO Orders (OrderID, OrderDate, CustomerID, TotalAmount)

VALUES (1, '2024-07-29 14:30:00', 101, 250.00);
```

Retrieving Data:

```
SELECT OrderID, OrderDate, CustomerID, TotalAmount

FROM Orders

WHERE OrderDate >= '2024-07-01' AND OrderDate < '2024-08-01';
```

- **Explanation**: The `DATETIME` data type is used to store both date and time values, allowing precise tracking and querying of events.

78. Handling Data Type Conversions from CSV

Steps:

1. **Data Cleaning**:
    - **Inspect the CSV**: Identify and clean mixed data types (e.g., dates in various formats, inconsistent numerical values).
2. **Data Import**:
    - **Use Data Import Tools**: Utilize tools or features of the database management system to import data, specifying data types for each column.
3. **Data Conversion**:
    - **Convert Data Types**: Use SQL functions or transformations to convert and standardize data types during or after import.
    - **Example**: Convert text dates to `DATETIME` and numbers stored as text to `INT` or `DECIMAL`.

SQL Example:

```
SELECT

    CAST(DateColumn AS DATETIME) AS OrderDate,

    CAST(NumberColumn AS DECIMAL(10, 2)) AS Amount
```

```
INTO Orders

FROM StagingTable;
```

- **Explanation**: Handling data type conversions ensures that data is correctly interpreted and stored in the appropriate format.

Correlation and Non-Correlation

79. Correlated Subquery to Find Employees with Salary Higher Than Department Average

SQL Query:

```
SELECT EmployeeID, Name, Salary, Department

FROM Employees E

WHERE Salary > (

    SELECT AVG(Salary)

    FROM Employees

    WHERE Department = E.Department

);
```

- **Explanation**: The subquery calculates the average salary for each department, and the main query retrieves employees whose salaries are higher than this average. The subquery is correlated because it references the outer query's `Department` column.

80. Optimizing Non-Correlated Subqueries

Non-Correlated Subqueries:

- **Description**: These subqueries do not depend on the outer query and are executed once.

Optimization Techniques:

1. **Indexing**:

    - **Use Indexes**: Ensure that columns used in the subquery are indexed to speed up data retrieval.
2. **Materialization**:
    - **Temporary Tables**: Store the results of the subquery in a temporary table or view if it is reused multiple times.
3. **Query Rewriting**:
    - **Simplify**: Rewrite the query to avoid unnecessary complexity and reduce execution time.

Example Optimization:

```
SELECT EmployeeID, Name

FROM Employees

WHERE DepartmentID IN (

    SELECT DepartmentID

    FROM Departments

    WHERE DepartmentName = 'Sales'

);

SELECT E.EmployeeID, E.Name

FROM Employees E

JOIN Departments D ON E.DepartmentID = D.DepartmentID

WHERE D.DepartmentName = 'Sales';
```

- **Explanation**: By converting the subquery to a `JOIN`, performance can be improved as joins are often optimized better by the database engine compared to subqueries.

Introduction to TSQL, Procedures, Functions, Triggers, Indices

81. Simple Stored Procedure to Insert a New Record

SQL Statement:

```
CREATE PROCEDURE InsertEmployee
```

```
    @EmployeeID INT,
    @Name VARCHAR(100),
    @Position VARCHAR(50),
    @Salary DECIMAL(10, 2)
AS
BEGIN
    INSERT INTO Employees (EmployeeID, Name, Position, Salary)
    VALUES (@EmployeeID, @Name, @Position, @Salary);
END;
```

- **Explanation**: This stored procedure takes parameters for `EmployeeID`, `Name`, `Position`, and `Salary`, and inserts a new record into the `Employees` table.

82. Scenario for Using a Trigger to Enforce Business Rules

Scenario: Ensuring that no employee's salary exceeds a certain threshold when inserting or updating records.

SQL Trigger:

```
CREATE TRIGGER CheckSalary
ON Employees
AFTER INSERT, UPDATE
AS
BEGIN
    IF EXISTS (
        SELECT *
        FROM inserted
```

```
        WHERE Salary > 100000
    )
    BEGIN
        RAISERROR('Salary exceeds the allowed limit of 100,000.',
16, 1);
        ROLLBACK TRANSACTION;
    END
END;
```

- **Explanation**: This trigger checks if any inserted or updated salary exceeds 100,000. If so, it raises an error and rolls back the transaction, enforcing the business rule that salaries must not exceed the specified limit.

Security and Accessibility

83. Common Security Measures to Protect a SQL Server Database

1. **Authentication and Authorization**:
    - **Description**: Use strong authentication methods (e.g., Windows Authentication) and assign appropriate roles and permissions to users to control access to database resources.
2. **Encryption**:
    - **Description**: Implement encryption for sensitive data to protect it from unauthorized access, using methods like Transparent Data Encryption (TDE) or column-level encryption.

84. Creating a User and Assigning Roles in MSSQL

SQL Statements:

```
-- Create a new user
CREATE USER NewUser FOR LOGIN NewLogin;
-- Assign roles to the user
ALTER ROLE db_datareader ADD MEMBER NewUser;
```

```
ALTER ROLE db_datawriter ADD MEMBER NewUser;
```

- **Explanation**: The `CREATE USER` statement creates a new user associated with a login, and the `ALTER ROLE` statements assign roles (e.g., `db_datareader`, `db_datawriter`) to the user.

85. Implementing Encryption for Data at Rest in MSSQL

Transparent Data Encryption (TDE):

Create a Master Key:

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

1. **Create a Certificate**:

   ```
   CREATE CERTIFICATE MyCertificate
WITH SUBJECT = 'My Database Encryption Certificate';
   ```

2. **Create a Database Encryption Key**:

   ```
   USE YourDatabase;
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE MyCertificate;
   ```

3. **Enable Encryption on the Database**:

   ```
   ALTER DATABASE YourDatabase
SET ENCRYPTION ON;
   ```

4. **Explanation**:

   TDE encrypts the entire database, including data files and transaction logs, to protect data at rest. The process involves creating a master key, a certificate, a database encryption key, and enabling encryption for the database.