Weekly Assignment

## Aggregate and Atomicity

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

An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

There are 5 types of SQL aggregate functions:

- Count()
- Sum()
- Avg()
- Min()
- Max()

Ex: SELECT AVG (salary)

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

The GROUP BY clause in SQL is used to arrange identical data into groups. This is particularly useful when you want to perform aggregate functions on each group of data. Aggregate functions include operations like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.

Basic Syntax:

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
GROUP BY column1, column2;
```

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

Atomicity is a fundamental property of database transactions that ensures operations are completed entirely or not at all, maintaining the database's integrity. A scenario where atomicity is crucial involves financial transactions, such as transferring money between two bank accounts.

Importance of Atomicity:

1.Consistency of Data

2.Avoiding Partial Updates

3.Maintaining Integrity

## OLAP and OLTP

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

1. **Purpose and Use Cases**

- **OLAP (Online Analytical Processing):**
    - **Purpose:** OLAP systems are designed for complex querying and data analysis. They are used for data mining, trend analysis, and business intelligence. The focus is on aggregating large volumes of data to provide insights and support decision-making.

- **Use Cases:** Examples include generating reports on sales performance, analyzing market trends, and creating dashboards for executive summaries.

- **OLTP (Online Transaction Processing):**
  - **Purpose:** OLTP systems are designed for managing and processing real-time transactional data. They are optimized for handling a high volume of short online transactions, such as insertions, updates, and deletions, ensuring quick response times and data integrity.
  - **Use Cases:** Examples include order processing systems, banking transactions, and customer relationship management (CRM) systems.

**2. Data Structure and Query Types**

- **OLAP:**
  - **Data Structure:** OLAP systems often use multi-dimensional data models (e.g., star schema or snowflake schema) that are optimized for complex queries and aggregations. Data is typically organized in a way that facilitates fast retrieval of summarized information.
  - **Query Types:** Queries in OLAP are generally complex and involve aggregations, calculations, and multidimensional analysis (e.g., "What is the total sales revenue for each region over the last five years?").
- **OLTP:**
  - **Data Structure:** OLTP systems use normalized relational data models to reduce redundancy and ensure data integrity. This structure supports efficient insertions, updates, and deletions.
  - **Query Types:** Queries in OLTP are usually simple and involve retrieving or updating individual records (e.g., "What is the balance of account number 123456?").

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

1.**Use Indexes Effectively:**

**Create Indexes:** On frequently queried columns, including primary and foreign keys.

**Choose Index Types:** Use B-Tree for general cases, hash indexes for equality checks, and composite indexes for multi-column queries.

**Maintain Indexes:** Regularly rebuild and monitor their usage to avoid fragmentation.

**2.Optimize Queries:**

**Write Efficient Queries:** Specify columns, use WHERE clauses, and minimize joins.

**Avoid Complex Joins/Subqueries:** Simplify queries for faster execution.

**3.Optimize Database Design:**

**Normalize Data:** Reduce redundancy and improve integrity.

**Consider Controlled Denormalization:** For specific performance gains.

**4.Optimize Transaction Management:**

**Minimize Transaction Scope:** Keep transactions short.
**Select Appropriate Isolation Levels:** Balance consistency with performance.

**5.Optimize Hardware and Configuration:**

**Improve Disk I/O:** Use SSDs for faster access.
**Tune Parameters:** Adjust buffer pool, cache, and log file settings.

**6.Regular Maintenance:**

**Update Statistics:** Keep database statistics current.
**Monitor Performance:** Use tools to track and address bottlenecks.

## Data Encryption and Storage

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

- Transparent Data Encryption (TDE)
- Column-Level Encryption
- Always Encrypted
- Backup Encryption
- Dynamic Data Masking

## SQL, NOSQL, Applications, Embedded

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

**SQL Databases:**

- Data Model: Relational. Data is structured in tables with rows and columns.
- Schema: Fixed schema, meaning the structure of the data (tables, columns, data types) must be defined and followed.
- Examples: MySQL, PostgreSQL, Microsoft SQL Server, Oracle.

**NoSQL Databases:**

- Data Model: Non-relational. Data can be stored in various formats such as key-value pairs, documents, graphs, or wide-column stores.
- Schema: Flexible schema, allowing for dynamic or semi-structured data without a predefined schema.
- Examples: MongoDB (document-based), Redis (key-value store), Neo4j (graph database), Cassandra (wide-column store).

## DDL

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

It is a blueprint that outlines the tables, views, indexes, relationships, and constraints within a database.

Syntax: CREATE SCHEMA schema_name;

2.Describe the process of altering an existing table

- **Add a Column:** ALTER TABLE table_name ADD new_column_name data_type;
- **Modify a Column:** ALTER TABLE table_name ALTER COLUMN column_name new_data_type;
- **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);
- **Drop a Constraint:** ALTER TABLE table_name DROP CONSTRAINT constraint_name;

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

| Aspect | Table | View |
|---|---|---|
| **Definition** | A physical structure that stores data in rows and columns | A virtual table based on a SELECT query |
| **Data Storage** | Stores data persistently on disk | Does not store data; provides a dynamic view of data |
| **Schema** | Fixed schema with defined columns and constraints | Defined by a SELECT query; schema can change based on query |
| **Data Modification** | Directly insert, update, or delete data | Data modification depends on the view's complexity and underlying tables |
| **Indexes** | Can have indexes to improve query performance | Does not have indexes; indexed views can be created for performance |
| **Usage** | Used for storing actual data | Used for presenting data, simplifying complex queries, and providing security |

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

**Creating a Simple Index**

A simple index is created on a single column and is used to speed up searches on that column.

SYNTAX : CREATE INDEX index_name ON table_name (column_name);

**Managing Indexes**

**Viewing Existing Indexes**

To view existing indexes on a table, you can query the sys.indexes catalog view or use SQL Server Management Studio (SSMS).

Ex : SELECT * FROM sys.indexes
WHERE object_id = OBJECT_ID('table_name');

**Rebuilding an Index**

Rebuilding an index can help defragment and optimize its performance. This is typically done during maintenance operations.

Ex : ALTER INDEX index_name
ON table_name REBUILD;

## DML

1.What are the most commonly used DML commands?

- **SELECT:** Retrieves data.
- **INSERT:** Adds new data.
- **UPDATE:** Modifies existing data.
- **DELETE:** Removes data.
- **MERGE:** Combines insert, update, and delete operations based on a condition.

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

One of the most common approaches to retrieve data from multiple tables in SQL is by utilizing JOIN clauses to combine data from different tables based on specified conditions.

Syntax:

```
SELECT t1.column1, t2.column2
FROM table1 t1
JOIN table2 t2 ON t1.id = t2.id;
```

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

Relational algebra provides a theoretical foundation for SQL operations by defining how data can be manipulated and combined. SQL queries implement these operations to retrieve and manipulate data from relational databases:

- **Selection** (σ) is equivalent to WHERE.
- **Projection** (π) is equivalent to SELECT.
- **Union** (∪) is equivalent to UNION.
- **Intersection** (∩) is equivalent to INTERSECT.
- **Difference** (−) is equivalent to EXCEPT.
- **Cartesian Product** (×) is equivalent to CROSS JOIN.
- **Join** (⋈) is equivalent to various types of JOIN operations.
- **Division** (÷) can be more complex and often requires subqueries.

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

- **Execution Time:** Complex queries often take longer to execute due to the additional processing required.
- **Resource Consumption:** They can consume more CPU, memory, and disk I/O, affecting overall system performance.
- **Query Plan Complexity:** More complex execution plans can be less efficient and harder to optimize.
- **Locking and Blocking:** Increased risk of locking and blocking, potentially affecting other transactions.

- **Index Utilization:** May not use indexes effectively, requiring more frequent index maintenance.
- **Scalability:** Performance can degrade with larger datasets, requiring additional optimization.
- **Maintenance:** Harder to debug and maintain, needing thorough testing and monitoring.

## Aggregate Functions

1.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

Differences Between HAVING and WHERE

1. **Stage of Filtering**:
    - **WHERE**: Filters rows before any grouping or aggregation takes place. It is used to specify conditions on individual rows in a table.
    - **HAVING**: Filters groups of rows after aggregation has been performed. It is used to specify conditions on the aggregated data.
2. **Usage with Aggregate Functions**:
    - **WHERE**: Cannot be used with aggregate functions like SUM(), COUNT(), AVG(), etc., because it filters rows before aggregation.
    - **HAVING**: Specifically designed to work with aggregate functions. It is used to filter the results of aggregate functions after the GROUP BY clause has processed the data.

Can You Use HAVING Before GROUP BY?

No, you cannot use HAVING before the GROUP BY clause. The HAVING clause is designed to work with the results of the GROUP BY clause, so it must come after GROUP BY in the query. The SQL query processing sequence is as follows:

1. FROM clause (including joins)
2. WHERE clause (filters individual rows)
3. GROUP BY clause (groups rows into aggregates)
4. HAVING clause (filters groups based on aggregate functions)
5. SELECT clause (selects the final output columns)
6. ORDER BY clause (sorts the final result set)

## Filters

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

1. **WHERE Clause**:

- **Purpose**: Filters rows from a result set based on a specified condition.

- **Usage**: Used to specify criteria for individual rows before any grouping or aggregation is performed.

**2. HAVING Clause**:

- **Purpose**: Filters groups of rows after aggregation has been performed.
- **Usage**: Used to specify criteria for aggregated data created by the GROUP BY clause.

**3. JOIN Conditions**:

- **Purpose**: Filters rows based on relationships between tables.
- **Usage**: Used in JOIN clauses to specify how rows from different tables should be matched.

4. **WHERE Clause in Subqueries**:

- **Purpose**: Filters rows within subqueries that are part of the main query.
- **Usage**: Used to define conditions for subqueries that contribute to the main query's result.

## 2.How do you use the WHERE clause to filter data in MSSQL?

In Microsoft SQL Server (MSSQL), the WHERE clause is used to filter records that meet certain conditions. It allows you to specify criteria for individual rows, ensuring that only the rows satisfying these conditions are included in the query results.

Syntax:

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

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

In SQL, combining multiple filter conditions using logical operators allows you to create complex queries that return results meeting various criteria. The main logical operators used for combining conditions are:

1. **AND**: Returns results where all specified conditions are true.
2. **OR**: Returns results where at least one of the specified conditions is true.
3. **NOT**: Returns results where the specified condition is false.
4. **BETWEEN**: Checks if a value is within a range of values.
5. **LIKE**: Checks if a value matches a specified pattern.

Ex:

```
SELECT * FROM employees
WHERE department = 'Sales' AND salary > 50000;
```

## 4.Explain the use of CASE statements for filtering data in a query.

- **Conditional Logic**: The CASE statement allows you to evaluate conditions and return different results based on those conditions. It’s often used to create new columns or modify output values.
- **Filtering**: While CASE isn’t typically used directly in the WHERE clause for filtering, you can use it in conjunction with other clauses to apply complex filtering logic.

- **Aggregation and Sorting**: CASE can be used in GROUP BY for conditional aggregation and in ORDER BY for custom sorting logic.

Syntax :

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
     ...
    ELSE default_result
END
```

## Operators

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

An operator is a symbol specifying an action that is performed on one or more expressions. The following table lists the operator categories that the SQL Server Database Engine uses.

- :: (Scope resolution)
- = (Assignment operator)
- Arithmetic operators
- Bitwise operators
- Comparison operators
- Compound operators
- Logical operators
- Relational operators
- Set operators: EXCEPT and INTERSECT, UNION
- String operators
- Unary operators: + (positive), - (negative)

2.How do arithmetic operators work in SQL?

Arithmetic operators, + , - , * , and / are used to perform calculations on data being returned by a query and can be included in the SELECT , WHERE , and ORDER BY clauses.

Ex:

```
SELECT monthyear, animal_type, outcome_type,
(age_in_days / 365) AS `Years Old`
FROM austin_animal_center_age_at_outcome
```

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

The LIKE operator in SQL is used for pattern matching in WHERE clauses to search for values that match a specific pattern. It allows for flexible querying of text data using wildcard characters. Wildcards enable you to match varying patterns of characters within your data.

**Wildcards Used with the LIKE Operator**

Here are the primary wildcards used with the LIKE operator:

1. **% (Percent Sign)**: Represents zero or more characters.
2. **_ (Underscore)**: Represents a single character.

Ex:

```
SELECT *
FROM employees
WHERE employee_name LIKE 'J%';
```

## Multiple Tables: Normalization

1.What is normalization and why is it important?

Normalization is a process in database design used to organize a database to reduce redundancy and improve data integrity. It involves decomposing a database into multiple related tables and defining relationships between them to ensure that each piece of data is stored only once. This helps to avoid anomalies and inconsistencies in the data.

**Importance of Normalization**

1. **Reduces Redundancy**: By dividing data into related tables, normalization eliminates duplicate data, reducing storage requirements and inconsistency.
2. **Improves Data Integrity**: Ensures that data dependencies make sense, thereby reducing anomalies and ensuring accurate data updates and deletions.
3. **Enhances Query Performance**: Although normalization can sometimes make queries more complex, it generally improves performance by reducing the amount of duplicated data that needs to be processed.
4. **Facilitates Maintenance**: Normalized databases are easier to maintain because changes to the schema or data are less likely to cause problems across multiple locations.

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

1. **First Normal Form (1NF)**

**Requirement**: A table is in 1NF if it contains only atomic (indivisible) values and each column contains only a single value for each row. This means that there should be no repeating groups or arrays within a table.

2. **Second Normal Form (2NF)**

**Requirement**: A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the entire primary key. This means that every non-key column must depend on the whole primary key, not just part of it.

3. **Third Normal Form (3NF)**

**Requirement**: A table is in 3NF if it is in 2NF and all the attributes are only dependent on the primary key, meaning there are no transitive dependencies (non-key attributes should not depend on other non-key attributes).

4. **Boyce-Codd Normal Form (BCNF)**

**Requirement**: A table is in BCNF if it is in 3NF and for every one of its non-trivial functional dependencies, the left-hand side is a superkey. It is a stricter version of 3NF designed to handle certain types of anomalies that 3NF does not address.

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

**Increased Query Complexity**

**Explanation**: Normalization involves breaking down a database into multiple related tables to eliminate redundancy and improve data integrity. While this leads to a more organized and consistent data structure, it can also make queries more complex.

**Impact**:

- **Join Operations**: To retrieve related data, normalized tables often require multiple JOIN operations. For example, if data is spread across several tables due to normalization, a query that needs to fetch a complete view of the data might require complex joins to piece together the information.
- **Performance Overhead**: These joins can add overhead to query processing, particularly if the database has a large volume of data or if the joins are not optimized. This can potentially slow down query performance compared to a denormalized structure where related data might be stored in fewer tables.

## Indexes & Constraints

1.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 space and potentially slower data modification operations. They work similarly to an index in a book, allowing the database to quickly locate specific rows in a table without scanning the entire table.

**What Are Indexes?**

An index is a data structure that provides a fast way to look up and retrieve rows from a database table based on the values in one or more columns. In most database systems, indexes are implemented using data structures like B-trees or hash tables.

Uses:

**Faster Queries**: They speed up data searches and retrieval by allowing quick access to rows without scanning the entire table.

- **Efficient Sorting**: Indexes help with fast sorting and ordering of query results.
- **Optimized Joins**: They enhance the performance of join operations by quickly matching rows between tables.
- **Uniqueness Enforcement**: Unique indexes ensure that values in specified columns are unique, maintaining data integrity.
- **Better Aggregations**: Indexes improve the performance of aggregate functions like COUNT, SUM, and AVG.

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

To create a unique constraint on a table column in SQL, you use the UNIQUE keyword. A unique constraint ensures that all values in a column or a combination of columns are distinct across the table, which helps maintain data integrity by preventing duplicate entries.

Ex:

```
CREATE TABLE employees (
  employee_id INT PRIMARY KEY,
  email VARCHAR(255) UNIQUE,
  name VARCHAR(100) );
```

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

- **Data Storage**:
  **Clustered Index**: Alters the physical order of the table data.
  **Non-Clustered Index**: Maintains a separate structure with pointers to the data.
- **Number per Table**:
  **Clustered Index**: One per table.
  **Non-Clustered Index**: Multiple per table.
- **Performance for Queries**:
  **Clustered Index**: Optimal for queries that benefit from sorted data (range queries, ordered results).
  **Non-Clustered Index**: Efficient for lookups, searches, and queries on non-primary key columns.
- **Impact on Data Modification**:
  **Clustered Index**: Can impact insert/update performance due to physical reordering.
  **Non-Clustered Index**: May slow down insert/update operations due to index maintenance.

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

Optimizing index usage in a highly transactional database involves carefully balancing query performance with the overhead introduced by maintaining indexes.

- Analyzing query patterns to design effective indexes.
- Monitoring and maintaining indexes to avoid performance degradation.
- Balancing index creation with data modification performance.
- Using partitioning or sharding for very large datasets.
- Optimizing queries to ensure effective index usage.

## Joins

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

- **Inner Join**: Returns rows with matching values in both tables.

- **Left Join**: Returns all rows from the left table and matched rows from the right table (NULLs if no match).
- **Right Join**: Returns all rows from the right table and matched rows from the left table (NULLs if no match).
- **Full Join**: Returns all rows with matches from either table (NULLs where no match).
- **Cross Join**: Returns the Cartesian product of the tables.
- **Self Join**: Joins the table with itself to compare rows within the same table.

2.Provide an example of a LEFT JOIN query.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example:

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments ON employees.department_id = departments.department_id;
```

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

**Definition**: Joins a table with itself to compare rows within the same table.

**Usage**: Useful for hierarchical data or when you need to compare rows within the same table.

Syntax:

```
SELECT a.columns, b.columns
FROM table a
INNER JOIN table b
ON a.column = b.column;
```

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

**Definition**: Returns all rows when there is a match in either left or right table. If there is no match, NULL values are returned for columns from the table without a match.

**Usage**: Useful when you need all records from both tables, with NULLs for non-matching rows.

Syntax:

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

## Alias

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

In SQL, an **alias** is a temporary name assigned to a table or a column within a query. Aliases are used to simplify complex queries, make the query results more readable, or provide a more meaningful name for columns and tables. Aliases are especially useful in scenarios involving multiple tables or complex calculations.

**Using Aliases for Columns :**

```
SELECT column_name AS alias_name FROM table_name;
```

**Using Aliases for Tables :**

```
SELECT column_name FROM table_name AS alias_name;
```

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

Ex:

```
SELECT e.first_name, e.last_name
FROM employees AS e;
```

3.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;
```

4.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.

## Joins vs Sub Queries

1.What is the difference between joins and subqueries?

1. **Performance**:
   - **Joins**: Typically more efficient for combining large datasets because SQL engines are optimized for join operations.
   - **Subqueries**: Can be less efficient, especially if used in the WHERE clause, as they may be executed multiple times for each row of the outer query.
2. **Complexity**:
   - **Joins**: Generally clearer when combining data from multiple tables and are more intuitive for many users.
   - **Subqueries**: Can be useful for complex conditions but might be harder to understand and manage, especially in deeply nested scenarios.
3. **Use Cases**:
   - **Joins**: Preferred for straightforward data retrieval from multiple tables where relationships are defined.
   - **Subqueries**: Useful for filtering results, performing operations on intermediate results, or when a query needs to use results from another query.
4. **Readability**:
   - **Joins**: Tend to be more readable for many users, especially when working with relational data.
   - **Subqueries**: Can be less readable, particularly when they are deeply nested.

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

- Filtering based on aggregated or calculated results.
- Working with complex conditions or comparisons.

- Avoiding Cartesian products.
- Using scalar results for comparison.
- Performing in situ calculations.
- Handling cases where joins are inefficient.
- Keeping queries simpler and more manageable.

3.Explain how correlated subqueries work with an example.

A **correlated subquery** is a type of subquery that references columns from the outer query. Unlike a regular subquery, which is self-contained and can be executed independently, a correlated subquery is executed once for each row processed by the outer query. This means the inner query (the subquery) depends on the outer query for its values, and its results vary based on the values of the current row in the outer query.

**How Correlated Subqueries Work**

1. **Execution Process**:
    - For each row processed by the outer query, the correlated subquery is executed.
    - The subquery uses values from the current row of the outer query to perform its operations and return a result.
    - The result of the subquery influences the rows returned by the outer query.
2. **Usage**:
    - Correlated subqueries are typically used to perform row-by-row comparisons or operations based on the values of the current row in the outer query.
    - They are often used with WHERE, SELECT, or HAVING clauses.

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

**Joins**

- **Efficiency**: Typically faster for combining data from multiple tables due to optimized join algorithms.
- **Indexes**: Utilize indexes effectively, enhancing performance.
- **Complexity**: Can handle large datasets well, but may become complex with many joins.

**Subqueries**

- **Efficiency**: Can be slower, especially correlated subqueries, as they may execute multiple times for each row of the outer query.
- **Flexibility**: Useful for complex conditions or calculations that are hard to express with joins.
- **Performance Overhead**: Correlated subqueries can be less efficient due to repeated execution.

## Types

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

- **Numeric**: INT, SMALLINT, BIGINT, DECIMAL, NUMERIC, FLOAT, REAL, BIT.
- **Character/String**: CHAR, VARCHAR, TEXT, NCHAR, NVARCHAR, NTEXT.
- **Date/Time**: DATE, TIME, DATETIME, SMALLDATETIME, DATETIME2, DATETIMEOFFSET.

- **Binary**: BINARY, VARBINARY, IMAGE.
- **Other**: UNIQUEIDENTIFIER, XML, JSON.
- **Spatial**: GEOGRAPHY, GEOMETRY.
- **Special**: ROWVERSION.

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

- **Data Nature**: Choose based on whether the data is numeric, character, date/time, binary, etc.
- **Precision and Scale**: Ensure the data type matches the precision and scale requirements.
- **Storage and Performance**: Select data types that optimize storage and performance.
- **Data Integrity and Constraints**: Ensure data types enforce integrity and constraints.
- **Future Growth**: Anticipate future changes and select scalable data types.

3.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.

## Correlation and Non-Correlation

1.What is a correlated subquery?

A **correlated subquery** is a type of subquery (or inner query) that references columns from the outer query. Unlike a regular subquery, which is independent of the outer query and can be executed on its own, a correlated subquery relies on the values of the current row of the outer query for its execution. This means that the subquery is executed once for each row processed by the outer query, and its results can vary based on the values of the outer query's row.

2.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, which depends on the values of the outer query and is executed once for each row of the outer query, a non-correlated subquery is executed only once for the entire outer query. Its results are static and do not vary based on the outer query's rows.

3.Explain how correlated subqueries can affect query performance.

- **Row-by-Row Execution**: Correlated subqueries are executed once for each row of the outer query, leading to potential performance issues with large datasets.
- **Increased Resource Usage**: They can increase I/O and CPU usage due to repeated executions.
- **Complex Execution Plans**: Complex query execution plans may affect optimization.
- **Index Utilization**: Poor index utilization can impact performance.
- **Locking and Concurrency**: Increased locking can affect concurrent query execution.

## Introduction to TSQL, Procedures, Functions, Triggers, Indices

1.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. T-SQL extends standard SQL by adding additional features and functionality to support complex querying, procedural programming, and administrative tasks.

2.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;
```

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

| Feature | Functions | Procedures |
|---|---|---|
| **Purpose** | Perform calculations or return values | Execute a series of SQL statements or operations |
| **Usage** | Can be used within SQL queries and expressions | Executed using EXEC or EXECUTE commands |
| **Return Type** | Returns a single value (scalar) or a table (table-valued) | Does not return a value directly; can return result sets, output parameters, or status codes |
| **Parameters** | Supports input parameters; no output parameters | Supports input parameters, output parameters, and return values |
| **Transaction Control** | Cannot manage transactions; no BEGIN TRANSACTION, COMMIT, or ROLLBACK | Can manage transactions using BEGIN TRANSACTION, COMMIT, and ROLLBACK |
| **Error Handling** | No direct support for TRY...CATCH within the function | Supports TRY...CATCH for error handling |
| **Execution** | Can be used in SELECT, WHERE, ORDER BY, and other SQL clauses | Executed as a separate command; cannot be used directly in SQL expressions |

4.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.

## Comparison input on TSQL with PL/SQL

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

- **Database System**:
    - **T-SQL**: Used with Microsoft SQL Server and Azure SQL Database.
    - **PL/SQL**: Used with Oracle Database.
- **Syntax and Conventions**:
    - **T-SQL**: Follows SQL Server conventions. For example, variable declaration is done using @ (e.g., DECLARE @VariableName INT).
    - **PL/SQL**: Follows Oracle conventions. For example, variables are declared without @ (e.g., DECLARE VariableName NUMBER;).
- **Error Handling**:
    - **T-SQL**: Uses TRY...CATCH blocks for error handling.
    - **PL/SQL**: Uses EXCEPTION blocks for error handling.
- **Dynamic SQL**:
    - **T-SQL**: Executes dynamic SQL using EXEC or sp_executesql.
    - **PL/SQL**: Uses EXECUTE IMMEDIATE for dynamic SQL execution.
- **Cursors**:
    - **T-SQL**: Cursors are used with DECLARE CURSOR, OPEN, FETCH, CLOSE.

    - **PL/SQL**: Cursors are used with similar syntax but also support cursor attributes and different types of cursors (explicit and implicit).

- **Exception Handling**:
    - **T-SQL**: Uses TRY...CATCH for handling exceptions.
    - **PL/SQL**: Uses EXCEPTION blocks for handling exceptions.

- **Built-in Functions**:
    - **T-SQL**: Provides a set of functions for various operations, including string manipulation, date handling, and more.
    - **PL/SQL**: Also provides extensive built-in functions, with some advanced features not available in T-SQL.
- **Stored Procedures**:
    - **T-SQL**: Stored procedures are defined using CREATE PROCEDURE and can be altered or dropped using ALTER PROCEDURE and DROP PROCEDURE.
    - **PL/SQL**: Stored procedures are defined similarly, but can also be part of packages.

- **Support for Packages**:
    - **T-SQL**: Does not have a direct equivalent to Oracle packages; similar functionality can be achieved through schemas and stored procedures.
    - **PL/SQL**: Supports packages, which allow grouping related procedures and functions together.

- **Data Types**:
    - **T-SQL**: Uses data types like NVARCHAR, DATETIME, and DECIMAL.
    - **PL/SQL**: Uses data types such as VARCHAR2, DATE, and NUMBER.

## Aggregate and Atomicity

1.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
```

2.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;
```

## Security and Accessibility

1.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?

**1. Define Access Control Requirements**

- **Identify Sensitive Data**: Determine which data is considered sensitive (e.g., personal information, payment details).
- **Determine User Roles**: Define roles and responsibilities for users (e.g., administrators, support staff, analysts) and what data each role should access.

## 2. Implement Database Security Measures

### 1. Use Database Roles and Permissions

```
CREATE ROLE AdminRole;
CREATE ROLE SupportRole;
CREATE ROLE AnalystRole;
GRANT SELECT, INSERT, UPDATE, DELETE ON Customers TO AdminRole;
GRANT SELECT ON Customers TO SupportRole;
GRANT SELECT ON Orders TO AnalystRole;
ALTER SERVER ROLE AdminRole ADD MEMBER [AdminUser];
ALTER SERVER ROLE SupportRole ADD MEMBER [SupportUser];
ALTER SERVER ROLE AnalystRole ADD MEMBER [AnalystUser];
```

**2. Use Column-Level Security**

- **Restrict Access to Sensitive Columns**: Limit access to sensitive columns using views or column-level permissions.

```
-- Create a view that excludes sensitive columns

CREATE VIEW vw_CustomerBasicInfo AS
SELECT CustomerID, Name, Email
FROM Customers;
-- Grant access to the view
GRANT SELECT ON vw_CustomerBasicInfo TO SupportRole;
```

**3. Implement Row-Level Security**

- **Filter Data Based on User Context**: Use row-level security to restrict access to rows based on user roles or attributes.

```
CREATE FUNCTION dbo.FilterCustomers(@UserRole NVARCHAR(50))

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN

SELECT CustomerID, Name, Email

FROM Customers

WHERE
```

```
    (@UserRole = 'Admin') OR
    (@UserRole = 'Support' AND Region = 'NorthAmerica'); -- Example condition

CREATE SECURITY POLICY CustomerSecurityPolicy
ADD FILTER PREDICATE dbo.FilterCustomers(UserRole) ON dbo.Customers;
```

**4. Use Encryption**

- **Encrypt Sensitive Data**: Use encryption to protect data at rest and in transit. This includes encrypting sensitive columns and using secure connections (e.g., SSL/TLS).

```
-- Example of column encryption
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'YourStrongPassword';
CREATE CERTIFICATE MyCertificate WITH SUBJECT = 'Database Encryption';
CREATE SYMMETRIC KEY MySymmetricKey WITH ALGORITHM = AES_256 ENCRYPTION BY
CERTIFICATE MyCertificate;

OPEN SYMMETRIC KEY MySymmetricKey DECRYPTION BY CERTIFICATE MyCertificate;
-- Encrypt sensitive column
UPDATE Customers SET CreditCardNumber =
ENCRYPTBYKEY(KEY_GUID('MySymmetricKey'), CreditCardNumber);
```

**5. Implement Authentication and Authorization**

- **Use Strong Authentication Methods**: Implement multi-factor authentication (MFA) and strong password policies for accessing the database.
- **Enforce Least Privilege Principle**: Ensure users have the minimum level of access necessary to perform their duties.

**6. Regularly Audit and Monitor Access**

- **Audit Database Activity**: Enable and review auditing to track access and modifications to sensitive data.

```
-- Example: Enabling SQL Server Audit
    CREATE SERVER AUDIT Audit_YourDB
    TO FILE ( FILEPATH = 'C:\AuditLogs\' )
    WITH ( QUEUE_DELAY = 1000, ON_FAILURE = CONTINUE );
    CREATE DATABASE AUDIT SPECIFICATION Audit_Spec
    FOR SERVER AUDIT Audit_YourDB
    ADD (SELECT ON dbo.Customers BY [public]) WITH (STATE = ON);
```

## MSSQL Install and Configure, OLAP and OLTP

1.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.

## Data Encryption and Storage

1.What is authentication vs autorization.

| Aspect | Authentication | Authorization |
|---|---|---|
| **Definition** | Verifying the identity of a user or system. | Determining what an authenticated user or system can access or do. |
| **Purpose** | To confirm that the entity is who they claim to be. | To control access to resources based on permissions or roles. |
| **Process** | Involves checking credentials (e.g., username/password, biometrics). | Involves checking permissions or roles to grant or deny access. |
| **Key Focus** | Identity verification. | Access control and permissions. |
| **Examples** | - Username and password<br>- Two-factor authentication (2FA)<br>- Biometric verification (e.g., fingerprint) | - Role-Based Access Control (RBAC)<br>- Access Control Lists (ACLs)<br>- Permissions and policies |
| **Occurs First** | Yes, authentication occurs before authorization. | No, it occurs after authentication. |
| **What It Ensures** | That the entity is legitimate. | That the entity has the right permissions. |
| **Implementation** | - Login systems<br>- Security tokens<br>- Identity verification methods | - Defining roles and permissions<br>- Implementing access control policies |
| **Dependency** | Authorization relies on successful authentication. | Authentication provides the identity needed for authorization. |

## DDL

1.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

Query:

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

2. 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
```

3.What is Composite key?

A composite key is a type of primary key in a database that consists of two or more columns to uniquely identify a record in a table. Unlike a single-column primary key, which relies on one column for uniqueness, a composite key uses a combination of multiple columns to ensure that each record is unique.

**Key Characteristics of Composite Keys:**

1. **Uniqueness:** The combination of the values in the columns that make up the composite key must be unique for each row in the table. No two rows can have the same combination of values in these columns.
2. **Multi-Column:** A composite key involves more than one column. The uniqueness is enforced by the combination of all the specified columns.
3. **Primary Key Constraint:** Composite keys are used as primary keys to ensure that each row in a table can be uniquely identified.
4. **Relationships:** Composite keys are often used in tables that represent many-to-many relationships between entities or where a single column alone cannot guarantee uniqueness.

Example:

```
CREATE TABLE CourseEnrollments (
    StudentID INT NOT NULL,
    CourseID INT NOT NULL,
    EnrollmentDate DATE,
    PRIMARY KEY (StudentID, CourseID)
);
```

## DML

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

Query:

```
UPDATE Employees SET Salary = Salary * 1.10;
```

2.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.

Query:

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

## Aggregate Functions

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

Query:

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

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

```
WITH SalesTotals AS (
  SELECT
    e.EmployeeID,
    e.Name,
    SUM(s.SaleAmount) AS TotalSales
    FROM Employees e
    JOIN Sales s ON e.EmployeeID = s.EmployeeID
   WHERE s.SaleDate BETWEEN '2024-06-01' AND '2024-06-30'
    GROUP BY e.EmployeeID, e.Name
      ),
 RankedSales AS (
```

```
SELECT
EmployeeID,
Name,
TotalSales,
RANK() OVER (ORDER BY TotalSales DESC) AS SalesRank
 FROM SalesTotals
 )
SELECT * FROM RankedSales;
```

## Filters

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

Query:

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

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

Example Scenario:

Suppose you have a Products table with a Price column, and you want to categorize products into price ranges:

- **Low**: Price less than ₹50
- **Medium**: Price between ₹50 and ₹100
- **High**: Price greater than ₹100

Query:

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

## Multiple Tables: Normalization

1.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.

## Indexes & Constraints

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

Query:

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

## Joins

1.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;
```

2.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;
```

## Alias

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

Query:

```
SELECT
    o.OrderID AS OrderID,
    o.OrderDate AS OrderDate,
    o.Quantity AS Quantity,
    c.CustomerName AS CustomerName,
    p.ProductName AS ProductName
FROM Orders o
```

```
INNER JOIN Customers c ON o.CustomerID = c.CustomerID
INNER JOIN Products p ON o.ProductID = p.ProductID;
```

2.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;
```

## Joins vs Sub Queries

1.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
);
```

2.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.

## Types

1.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;
```

2.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.

## Correlation and Non-Correlation

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

Query:

```
SELECT e1.employee_id, e1.employee_name, e1.salary, e1.department_id
FROM Employees e1
WHERE e1.salary > (
    SELECT AVG(e2.salary)
    FROM Employees e2
    WHERE e2.department_id = e1.department_id
);
```

2.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.

## Introduction to TSQL, Procedures, Functions, Triggers, Indices

1.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');
```

2.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 ;
```

## Security and Accessibility

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

- **Authentication and Authorization**:

This involves configuring who can access the SQL Server and what they can do once they have access. SQL Server supports both Windows Authentication (using Windows accounts) and SQL Server Authentication (using SQL Server-specific accounts). Implementing strong, unique passwords and using roles and permissions to grant the least privileges necessary to users helps minimize security risks.

- **Encryption**:

This involves protecting data both at rest and in transit. For data at rest, SQL Server provides features like Transparent Data Encryption (TDE) to encrypt the database files. For data in transit, you can use SSL/TLS to encrypt the data as it travels between the client and the server, ensuring that sensitive information isn't exposed during transmission.

2.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];
```

3.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';
```