## MERGE Statement Overview

The MERGE statement is a powerful SQL command used to perform INSERT, UPDATE, and DELETE operations in a single transaction. It compares two tables, a target table and a source table, and based on the condition provided, it can either:

* Insert new records,
* Update existing records, or
* Delete records from the target.

### Detailed Explanation

1. **Target and Source Tables**:
   * **Target**: The table where changes are being made.
   * **Source**: The table containing new or updated data.
2. **ON Clause**: This specifies the condition for matching rows between the Target and Source tables. Typically, it's based on the primary key or unique column (e.g., ProductID).

### Key Actions in the MERGE Statement

1. **INSERT**: If a row in the source does not have a corresponding match in the target, the WHEN NOT MATCHED BY Target condition inserts a new record from the source table into the target.
2. **UPDATE**: If a row in the source table matches a row in the target (based on the ON condition), the WHEN MATCHED condition updates the target table with the values from the source table.
3. **DELETE**: If a row in the target table does not have a matching row in the source table, the WHEN NOT MATCHED BY Source condition deletes that row from the target.
4. **OUTPUT Clause**: This clause provides a way to track the changes made by the MERGE operation, such as identifying which rows were inserted, updated, or deleted. It includes details from both the DELETED (for target) and INSERTED (for source) pseudo-tables.

### Important Considerations

1. **Semicolon Requirement**:
   * Every MERGE statement must end with a semicolon (;). If it's omitted, SQL Server will throw an error.
   * Example: MERGE ... ;
2. **RowCount**:
   * After executing the MERGE statement, you can use SELECT @@ROWCOUNT to check the number of rows affected by the operation.
   * Example:

MERGE ...;

SELECT @@ROWCOUNT;

1. **Mandatory MATCHED Clause**:
   * It's essential that at least one of the MATCHED clauses (WHEN MATCHED or WHEN NOT MATCHED BY Target or Source) is provided for the MERGE to operate. This defines the actions to take when the data matches or does not match.
2. **Ordering**:
   * The MERGE statement processes the data in a single scan, making it more efficient than executing separate INSERT, UPDATE, or DELETE statements. All actions are processed in one step.
3. **Performance Considerations**:
   * While MERGE can be very efficient, it may lead to performance issues if not used carefully, especially on large datasets. It's important to index the tables on the key column used in the ON condition to ensure optimal performance.
4. **Transaction Handling**:
   * Since MERGE is an all-in-one operation, it's implicitly wrapped in a transaction. Any error encountered will roll back the entire transaction, maintaining data integrity.
5. **Triggers and Constraints**:
   * Be mindful of triggers and foreign key constraints on the target table. They may affect the behavior of the MERGE statement. For instance, if an update causes a violation of a foreign key constraint, it may lead to an error.

### Example with Output

MERGE TargetProducts AS Target

USING SourceProducts AS Source

ON Source.ProductID = Target.ProductID

-- For Inserts

WHEN NOT MATCHED BY Target THEN

INSERT (ProductID, ProductName, Price)

VALUES (Source.ProductID, Source.ProductName, Source.Price)

-- For Updates

WHEN MATCHED THEN

UPDATE SET

Target.ProductName = Source.ProductName,

Target.Price = Source.Price

-- For Deletes

WHEN NOT MATCHED BY Source THEN

DELETE;

-- Output to see the action performed

OUTPUT $action,

DELETED.ProductID AS TargetProductID,

DELETED.ProductName AS TargetProductName,

DELETED.Price AS TargetPrice,

INSERTED.ProductID AS SourceProductID,

INSERTED.ProductName AS SourceProductName,

INSERTED.Price AS SourcePrice;

### Conclusion

The MERGE statement provides an efficient and convenient way to synchronize two tables with various data modifications (insert, update, delete). It ensures that all changes are made in a single operation, optimizing both performance and data integrity. However, proper attention should be paid to the clause conditions, output handling, and potential side effects (e.g., triggers, constraints).

## **Stored procedure**

A **stored procedure** is a precompiled collection of SQL statements that are executed together when called. Stored procedures help improve performance, ensure reusability, and provide encapsulation of business logic. They can accept parameters, execute multiple SQL statements, and return data or status information.

Here is an organized explanation of the concepts related to stored procedures, including examples, functionality, and usage.

### Creating a Stored Procedure

A stored procedure is created using the CREATE PROCEDURE statement. It can accept parameters (input, output) and return values.

#### Example of Creating a Stored Procedure with Output Parameter:

CREATE PROCEDURE outputparam

@paramout VARCHAR(20) OUT

AS

BEGIN

SELECT @paramout = 'Hello world'

END

#### Executing the Stored Procedure and Using the Output Parameter:

DECLARE @message VARCHAR(20)

EXEC outputparam @paramout = @message OUT

SELECT @message AS regards

SELECT CONCAT(@message, ', welcome to sqlshack')

In this example:

* The outputparam procedure sets the value of @paramout to 'Hello world'.
* The value of @message is fetched from the procedure, and concatenated with a string.

### Executing Stored Procedures with EXEC Statement

You can execute a stored procedure using the EXEC statement, which can also accept parameters.

#### Example of EXEC with INSERT:

INSERT INTO Person.Address2

EXEC tablexample -- This is assuming tablexample provides a SELECT statement for Address2.

#### Example of EXEC that Does Not Work:

EXEC tablexample INTO Person.Address3 -- This will not work

### WITH RESULT SETS in SQL Server

The **WITH RESULT SETS** clause allows you to alter the result set's structure (such as changing column names or data types) when executing a stored procedure.

#### Example of Using WITH RESULT SETS:

EXEC GetSalesOrder

WITH RESULT SETS

(

(OrderId INT, CustCode VARCHAR(20), OrderDate VARCHAR(10), Amount FLOAT)

)

This redefines the result set’s column names and data types to match the specified ones.

**Important points:**

1. The number of columns cannot be fewer or greater than the result set returned by the procedure; otherwise, an exception will occur.
2. Data types must be compatible. For instance, changing a column's data type to an incompatible one will throw an exception.

#### Example of Procedure with Multiple Result Sets:

CREATE PROCEDURE GetSalesOrderTwoResultSet

AS

BEGIN

SELECT OrderId, CustomerCode, OrderDate, TotalAmount FROM SALESORDER

SELECT OrderId, OrderDate, TAXAmount FROM SALESORDER

END

You can execute this stored procedure and redefine the result sets:

EXEC GetSalesOrderTwoResultSet

WITH RESULT SETS

(

(OrderId INT, CustCode VARCHAR(20), OrderDate VARCHAR(10), Amount FLOAT),

(OrderId INT, OrderDate VARCHAR(10), TAX FLOAT)

)

You can also specify WITH RESULT SETS NONE to indicate that no result sets will be returned from a procedure that does not provide any result set, like one with only PRINT statements.

EXEC WithNoResultSet

WITH RESULT SETS **NONE** -- does not return any result set(even though result have data)

If the stored procedure has a result set but no records, using WITH RESULT SETS **UNDEFINED** will result in an error. (Undefined --🡪 allow SQL Server to handle it dynamically at runtime.)

### Return Parameter vs OUTPUT Parameter

* **Return Parameter**:
  + A **return parameter** is an integer value (typically INT) that is returned by the stored procedure to indicate success or failure.
  + You can only return one RETURN value from a stored procedure.
  + It is mainly used for status codes, e.g., 0 for success, 1 for failure.

CREATE PROCEDURE TestReturnParam

AS

BEGIN

RETURN 0 -- Success

END

* + To capture the return value:

DECLARE @result INT

EXEC @result = TestReturnParam

SELECT @result AS ReturnValue

* **OUTPUT Parameter**:
  + **OUTPUT parameters** are used to return multiple values from a procedure. These parameters can be of various data types (e.g., VARCHAR, INT).
  + A stored procedure can have multiple OUTPUT parameters, unlike the RETURN parameter, which can only return one value.

CREATE PROCEDURE TestOutputParam

@outParam VARCHAR(50) OUTPUT

AS

BEGIN

SET @outParam = 'Hello, SQL!'

END

* + To capture the OUTPUT parameter:

DECLARE @message VARCHAR(50)

EXEC TestOutputParam @outParam = @message OUTPUT

SELECT @message AS OutputMessage

**Limitations of OUTPUT Parameters**:

* You can use multiple OUTPUT parameters, but they have limitations on the data types supported (e.g., VARCHAR, INT, etc.).

### Why SELECT INTO Doesn't Work with Stored Procedures:

Stored procedures, unlike table-valued functions, don't directly return a result set in a manner that can be used in a SELECT INTO statement. They execute a set of SQL statements and may return multiple result sets or modify data, but they don't directly return a **table** that can be referenced in the FROM clause like a TVF.

Summary of Differences:

* **User-Defined Function (UDF) - Table-Valued Function (TVF)**:
  + You can use the result directly in a SELECT INTO statement.
  + TVFs return a **table** that can be queried like any other table.
* **Stored Procedure (SP)**:
  + Stored procedures do not directly return tables.
  + You must use INSERT INTO combined with EXEC to capture the result of the stored procedure and insert it into a table.

### Miscellaneous Features

1. **Statistics for Query Performance**:
   * You can use SET STATISTICS to gather details about query performance, such as I/O and time statistics:

SET STATISTICS IO ON

SET STATISTICS TIME ON

* + After running the query or stored procedure, you will get output related to IO (reads and writes) and time (execution time).

1. **Clearing the Buffer**:
   * To reset the buffer between tests, you can use DBCC DROPCLEANBUFFERS:

DBCC DROPCLEANBUFFERS WITH NO\_INFOMSGS;

* + **SET NOCOUNT ON**: This prevents the message that shows the number of rows affected from being returned. It's useful for reducing unnecessary output.

SET NOCOUNT ON;

1. **Execution Phases**: The typical phases during SQL execution are:
   * **Parsing**: The SQL query is parsed to ensure correct syntax.
   * **Binding**: The parser binds variables and resolves names to objects.
   * **Optimization**: The query is optimized for the best execution plan.
   * **Execution**: The optimized plan is executed against the database.

### Conclusion

Stored procedures are a vital feature in SQL Server for modularizing and reusing SQL code. They support input, output, and return parameters, and can handle multiple result sets. The WITH RESULT SETS clause provides flexibility to modify the result set structure while executing a stored procedure. Understanding the differences between return and output parameters, as well as performance tuning features like SET STATISTICS, is crucial for efficient stored procedure development.

### Function vs Stored Procedure

| **Aspect** | **Function** | **Stored Procedure** |
| --- | --- | --- |
| **Parameters** | Functions can only have **input parameters**. | Stored procedures can have **both input and output parameters**. |
| **DML Operations** | Functions cannot use **DML** statements (INSERT/UPDATE/DELETE). | Stored procedures can use **DML** statements (INSERT/UPDATE/DELETE). |
| **Usage in SELECT Statement** | A function can be used **within a SELECT statement**. | Stored procedures **cannot** be used directly in a SELECT statement. |
| **Try-Catch Block** | Functions cannot use a **TRY-CATCH** block. | Stored procedures can use **TRY-CATCH** blocks for error handling. |
| **Transactions** | Functions **cannot** contain transactions. | Stored procedures can **contain transactions** (BEGIN TRANSACTION, COMMIT, ROLLBACK). |
| **Return Values** | Functions **must return a value** (scalar, table, etc.). | Stored procedures **can return 0 or multiple values** (via output parameters or result sets). |
| **Rowset Operations** | Functions that return tables (Inline Table-Valued Functions) can be used in **JOINs** with other tables. | Stored procedures cannot be directly used in **JOINs**. |
| **SQL Statement Usage** | Functions can be used in SQL statements, such as **WHERE**, **HAVING**, or **SELECT** clauses. | Stored procedures **cannot** be used directly in the WHERE/HAVING/SELECT clauses. |
| **Return Type** | Functions can **only return a single value** or a table. | Procedures can return **multiple values** or result sets. |
| **Calling from Other Functions** | A function **cannot call** a stored procedure. | A stored procedure **can call** a function. |
| **Compilation** | Functions are **compiled every time** they are executed. | Stored procedures **are not recompiled** unless altered or forced. |
| **Input Parameters Limit** | Functions can have up to **1023 input parameters**. | Stored procedures can have up to **2100 input parameters**. |
| **Use Case for Inline Functions** | Inline functions are similar to **views** that can take parameters and be used in **JOINs** and other rowset operations. | Stored procedures **cannot** be used in the same way as functions in JOINs or rowset operations. |

### Best for Parameter Sniffing

**Best local variables** and **sp\_recompile**

 **OPTION (RECOMPILE)**: Use when you have **one-off queries** where **parameter variations** can drastically change the optimal plan (e.g., queries with highly varying parameter values that are not predictable).

 **OPTIMIZE FOR**: Use when you want to **optimize** for a **specific parameter** value (e.g., if one value is much more common or representative of typical use). However, it's generally less flexible than the other approaches and is more of a workaround than a universal solution.

## Functions

### Types of User-Defined Functions in SQL Server

In SQL Server, user-defined functions (UDFs) allow you to encapsulate logic that can be reused within queries. There are three types of UDFs:

1. **Scalar Functions**: These functions return a single value (such as INT, VARCHAR, etc.).
2. **Inline Table-Valued Functions (TVFs)**: These functions contain a single T-SQL statement and return a table.
3. **Multi-Statement Table-Valued Functions (TVFs)**: These functions contain multiple T-SQL statements and return a table.

### Points to Remember:

* **Schema name is mandatory**: When invoking a function, you must always specify the schema name (e.g., dbo.FunctionName()).

### 1. ****Scalar Function Example****

A **scalar function** returns a single value. Below is an example of a function that converts dollars into pesos.

CREATE FUNCTION dbo.priceinpesos(@dollar REAL)

RETURNS REAL

AS

BEGIN

RETURN @dollar \* 20.33

END

This function takes a parameter @dollar (a real number) and returns the dollar amount converted into pesos (multiplied by a fixed exchange rate of 20.33).

To use this function:

SELECT dbo.priceinpesos(100); -- Converts 100 dollars into pesos

### 2. ****Inline Table-Valued Function (TVF) Example****

An **inline table-valued function** returns a table as a result of a single SELECT statement. Here is an example of an inline TVF:

CREATE FUNCTION dbo.functiontable()

RETURNS TABLE

AS

RETURN

(

SELECT [AddressID], [AddressLine1], [AddressLine2], City

FROM [Person].[Address]

)

This function returns a table with AddressID, AddressLine1, AddressLine2, and City columns from the Address table in the Person schema.

To use this function:

SELECT \* FROM dbo.functiontable(); -- Selects all columns returned by the function

SELECT AddressID FROM dbo.functiontable(); -- Selects only AddressID from the function

SELECT AddressID FROM dbo.functiontable() WHERE AddressID = 502; -- Filters results by AddressID

You can also insert the results from this function into another table:

SELECT \*

INTO mytable

FROM dbo.functiontable(); -- Inserts all the returned rows into a new table

### 3. ****Multi-Statement Table-Valued Function (TVF) Example****

A **multi-statement table-valued function** allows multiple T-SQL statements to be executed, returning a table as a result. Here is an example:

CREATE FUNCTION dbo.functioninsidefunction()

RETURNS varchar(50)

AS

BEGIN

RETURN dbo.helloworldfunction()

END

In this example, dbo.functioninsidefunction() calls another scalar function (dbo.helloworldfunction()), which returns a greeting string. Here's the definition of dbo.helloworldfunction():

CREATE FUNCTION dbo.helloworldfunction()

RETURNS varchar(20)

AS

BEGIN

RETURN 'Hello world'

END

You can now invoke the nested function like so:

SELECT dbo.functioninsidefunction(); -- Returns 'Hello world'

### Key Points:

* **Scalar Functions** return a single value.
* **Inline TVFs** return a table based on a single SELECT statement.
* **Multi-Statement TVFs** return a table based on multiple T-SQL statements.

### Usage of Functions in Queries:

* Functions can be invoked in the SELECT clause, WHERE clause, or even in JOIN conditions.
* For example, dbo.helloworldfunction() can be used as part of a SELECT statement or concatenated with other text:

SELECT CONCAT(dbo.helloworldfunction(), ', welcome to SQL Shack') AS Greeting;

### Conclusion:

User-defined functions in SQL Server are powerful tools for encapsulating reusable logic. You can define scalar functions for returning single values, inline table-valued functions for returning tables in a single query, and multi-statement table-valued functions for more complex operations that return tables. Each type of function has its specific use cases depending on the scenario.

## Error Handling

<https://www.sqlshack.com/how-to-implement-error-handling-in-sql-server/>

### Errors That Cannot Be Handled by the CATCH Block

* Compile-time errors (syntax or object resolution errors).
* Fatal system errors (severity level 20 and higher).
* Informational messages (severity level 10 or lower).
* Client connection issues.
* Recompilation errors (e.g., invalid table or column references).

#### 2. **Errors with Severity Level 20 or Higher**

**Fatal errors** (such as hardware failures, memory issues, or internal SQL Server errors) with a severity level of 20 or higher cannot be caught by TRY...CATCH. These errors typically cause the session to terminate immediately.

#### 3. **Errors with Severity Level 10 or Lower**

Informational **messages** with a severity level of 10 or lower (which are not actual errors) are not captured by the CATCH block. These messages generally do not impact the query execution and are typically warnings.

#### 4. **Client Connection or Session Issues**

Errors caused by **client connection interruptions**, broken client connections, or **killed sessions** are not caught by the TRY...CATCH block. These issues typically cause the client to disconnect, and SQL Server cannot handle these errors within the session.

#### 5. **Recompilation Errors**

**Statement-level recompilation errors**, such as **object name resolution errors** (e.g., invalid table or column reference), will prevent the batch from running and will not pass control to the CATCH block.

### Key Differences XACT\_ABORT ON vs OFF:

#### SET XACT\_ABORT ON:

SQL Server automatically rolls back the transaction when any runtime error occurs.

This reduces the need for manual error handling in the CATCH block, making the code simpler and less error-prone.

BEGIN TRY

BEGIN TRANSACTION;

-- Some operation that causes an error

INSERT INTO MyTable (Column1, Column2) VALUES ('Value1', 'Value2');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Manually handling the rollback

IF XACT\_STATE() = -1

BEGIN

-- If the transaction is uncommittable, roll it back

ROLLBACK TRANSACTION;

END

THROW;

END CATCH

| **XACT\_STATE() Value** | **Meaning** |
| --- | --- |

|  |  |
| --- | --- |
| **1** | The transaction is active and can be committed. |

|  |  |
| --- | --- |
| **-1** | The transaction is uncommittable (error occurred). |

|  |  |
| --- | --- |
| **0** | No active transaction exists (already committed or rolled back). |

#### SET XACT\_ABORT is OFF

SQL Server will not automatically roll back the transaction for most errors;

it will continue processing, and you must handle the rollback explicitly in the CATCH block.

(which is the default), only certain errors (like those that are fatal or result in termination) will cause the transaction to be rolled back automatically.

many errors (like constraint violations, deadlocks, etc.), SQL Server will not automatically roll back the transaction

You will need to check the transaction state (XACT\_STATE()) to decide whether or not to roll back.

SET XACT\_ABORT ON;

BEGIN TRY

BEGIN TRANSACTION;

-- Some operation that causes an error

INSERT INTO MyTable (Column1, Column2) VALUES ('Value1', 'Value2');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- No need to manually check XACT\_STATE() or call ROLLBACK

THROW;

END CATCH

## Trigger

Triggers in SQL Server are special types of stored procedures that automatically execute when certain events occur in the database, such as INSERT, UPDATE, or DELETE statements. Triggers are defined to respond to Data Manipulation Language (DML) events and can also respond to Data Definition Language (DDL) events. They are commonly used to enforce business rules, maintain referential integrity, audit changes, and perform cascading operations.

### Types of Triggers

1. **DML Triggers (Data Manipulation Language)**:
   * **Triggered by**: INSERT, UPDATE, or DELETE statements on a table or view.
   * **Execution**: Can be executed before or after the event occurs.
   * **Use cases**: Maintaining data integrity, auditing, cascading changes.
   * **Example**: A trigger can be used to log all updates to a specific table.

**Magic Tables**: In DML triggers, there are two special tables, inserted and deleted, which contain the new and old values of the modified rows, respectively. These tables allow access to data before and after modification.

Example:

CREATE TRIGGER trg\_AfterInsert

ON Employees

AFTER INSERT

AS

BEGIN

DECLARE @EmployeeID INT;

SELECT @EmployeeID = EmployeeID FROM inserted;

PRINT 'New Employee Added: ' + CAST(@EmployeeID AS VARCHAR(10));

END

1. **DDL Triggers (Data Definition Language)**:
   * **Triggered by**: DDL events such as CREATE, ALTER, DROP operations.
   * **Execution**: Only executed after the event occurs (i.e., when an object is created, altered, or dropped).
   * **Use cases**: Preventing schema changes, auditing DDL operations, preventing the dropping of critical objects.
   * **Example**: Preventing the drop of a critical table.

### Event Types in DDL Triggers

* There are different **event scopes** where a trigger can be applied: **on database**, **on schema**, and **on object**.

1. EVENTDATA(): This function returns an XML document containing the event's data.

Example:

CREATE TRIGGER TR\_ALTERTABLE

ON DATABASE

FOR ALTER\_TABLE

AS

BEGIN

INSERT INTO TableSchemaChanges

SELECT EVENTDATA(), GETDATE();

END

### How Many Triggers Can You Have on a Table?

You can have multiple **AFTER triggers** for each type of DML operation (INSERT, UPDATE, DELETE) on a table. However, only **one INSTEAD OF trigger** can be defined for each DML operation (INSERT, UPDATE, DELETE) on a table.

* **AFTER Triggers**: Multiple AFTER triggers can be created for INSERT, UPDATE, or DELETE on a table. These triggers execute after the DML event completes.
* **INSTEAD OF Triggers**: You can define only one INSTEAD OF trigger for each DML operation on a table. These triggers are executed in place of the DML operation and can modify or prevent the default behavior.

##### **Summary: Why the Difference? INSTEAD OF** vs **AFTER** Triggers

* **Multiple AFTER triggers**: They are **executed after the operation**, and each trigger can perform a separate action without interfering with the others. SQL Server runs them in sequence. The ability to create multiple AFTER triggers allows for modular design and separation of logic.

**Create trigger triggername1 on person AFTER INSERT**

**Create trigger triggername2 on person AFTER INSERT**

* **One INSTEAD OF trigger per DML operation**: An INSTEAD OF trigger **replaces** the default DML operation. If there were multiple INSTEAD OF triggers for the same operation, SQL Server would not know which one to run, leading to potential conflicts or unexpected results. Therefore, only **one INSTEAD OF trigger per DML operation** is allowed.

**Create trigger triggername1 on person** INSTEAD of **INSERT**

### Trigger vs Stored Procedure

* **Stored Procedures**:
  + Can be called explicitly by the user.
  + Can be executed on demand using EXEC or CALL statements.
  + Stored procedures can perform any type of operation, including SELECT, INSERT, UPDATE, DELETE, and more.
* **Triggers**:
  + Cannot be called explicitly by the user.
  + Automatically invoked by certain events (INSERT, UPDATE, DELETE, or DDL events).
  + They are typically used for enforcing data integrity, auditing changes, or cascading changes to related data.

### Invoking a Trigger on Demand

Triggers **cannot** be invoked manually on demand. They are event-driven and only execute in response to specific database events like data modifications (DML) or schema changes (DDL). The only way to cause a trigger to fire is by performing an action that triggers the event (e.g., inserting a record into a table to invoke an INSERT trigger).

### Nested Triggers

A **nested trigger** occurs when a trigger performs data modification actions (INSERT, UPDATE, DELETE) within itself, which can, in turn, trigger another trigger. This can create a chain of triggers being executed in sequence. For example, a trigger on an UPDATE statement can insert data into another table, which might fire another trigger.

In SQL Server, **nested triggers** can be controlled using the nested triggers setting (SET RECURSION), which defines how many levels of nested triggers are allowed.

Example of nested trigger:

CREATE TRIGGER trg\_AfterUpdate

ON Orders

AFTER UPDATE

AS

BEGIN

-- This update will fire another trigger

UPDATE Inventory SET Quantity = Quantity - 1 WHERE ProductID = (SELECT ProductID FROM inserted);

END

**Example of an INSTEAD OF trigger on a view**:

CREATE TRIGGER trg\_InsteadOfInsert

ON EmployeesView

INSTEAD OF INSERT

AS

BEGIN

INSERT INTO Employees (Name, Age)

SELECT Name, Age FROM inserted;

END

### ****Triggers and Constraints****

* A trigger (INSTEAD OF UPDATE) cannot be created when a FOREIGN KEY constraint with ON UPDATE CASCADE is in place.
* For example, you cannot create an INSTEAD OF DELETE trigger on a table with a DELETE CASCADE foreign key constraint.

### Conclusion

* **Triggers** are event-based stored procedures that automatically execute in response to certain database events like DML (INSERT, UPDATE, DELETE) or DDL (CREATE, ALTER, DROP) actions.
* **DML triggers** can be executed before or after a data modification, whereas **DDL triggers** are executed after database schema changes.
* Triggers are used to enforce rules, maintain data integrity, and perform auditing, but they cannot be invoked on demand like stored procedures.
* **Nested triggers** can occur when triggers themselves perform data changes that trigger additional triggers.

## Cursor

A **cursor** is a database object used to process individual rows from a result set, one row at a time. While SQL operates on the result set as a whole, cursors allow you to work with data row-by-row, which can be useful in certain scenarios.

DECLARE @EmployeeID INT, @EmployeeName VARCHAR(50);

DECLARE EmployeeCursor CURSOR FOR

SELECT EmployeeID, Name FROM Employees WHERE Department = 'Sales';

OPEN EmployeeCursor;

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @EmployeeName;

WHILE @@FETCH\_STATUS = 0

BEGIN

PRINT 'Employee ID: ' + CAST(@EmployeeID AS VARCHAR) + ' Name: ' + @EmployeeName;

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @EmployeeName;

END

CLOSE EmployeeCursor;

DEALLOCATE EmployeeCursor;

## ****Joins****

Joins are used to combine rows from two or more tables based on a common field between them.

### ****Types of Joins****

1. **INNER JOIN (Physical Join)**
   * Returns all rows when there is at least one match in both tables.
   * Only the rows that have a matching key in both tables are returned.

Example:

SELECT ColumnName\_1, ColumnName\_2

FROM Table\_1

INNER JOIN Table\_2 ON Table\_1.key = Table\_2.key;

1. **OUTER JOINS**
   * Returns rows even if there is no match between the two tables.

**a. LEFT OUTER JOIN**

* + Returns all rows from the left table and the matched rows from the right table. If there is no match, NULLs are returned for the right table.

**b. RIGHT OUTER JOIN**

* + Returns all rows from the right table and the matched rows from the left table. If there is no match, NULLs are returned for the left table.

**c. FULL OUTER JOIN**

* + Returns all rows when there is a match in either the left or right table. If there is no match, NULLs are returned for the non-matching table.

Example:

SELECT ColumnName\_1, ColumnName\_2

FROM Table\_1

LEFT OUTER JOIN Table\_2 ON Table\_1.key = Table\_2.key;

1. **CROSS JOIN**
   * Returns the Cartesian product of the two tables. Every row in the first table is combined with every row in the second table.

Example:

SELECT ColumnName\_1, ColumnName\_2

FROM Table\_1

CROSS JOIN Table\_2;

### ****Logical Joins****

Different types of logical joins used in query optimization:

1. **Nested Loop Join**: Used when one table is small (e.g., 5 rows) and the other is large (e.g., 50,000 rows), and when join columns are indexed.
2. **Merge Join**: Used when both tables are of similar size (e.g., 400,000 rows and 500,000 rows) and the join columns are indexed.
3. **Hash Join**: Used when both tables are large, but one is much larger than the other (e.g., 100,000 rows vs. 1 million rows), and the join columns are not indexed.

### ****Self-Join****

A self-join is a query used to join a table to itself.

Example: Consider an Employees table with employee IDs and manager IDs. A self-join can be used to find the managers of each employee.

CREATE TABLE emp (

empid INT,

mgrid INT,

empname CHAR(10)

);

INSERT INTO emp VALUES (1, 2, 'Vyas');

INSERT INTO emp VALUES (2, 3, 'Mohan');

INSERT INTO emp VALUES (3, NULL, 'Shobha');

INSERT INTO emp VALUES (4, 2, 'Shridhar');

INSERT INTO emp VALUES (5, 2, 'Sourabh');

-- Query to find employee and their manager

SELECT t1.empname AS Employee, t2.empname AS Manager

FROM emp t1

JOIN emp t2 ON t1.mgrid = t2.empid;

To handle employees without managers (e.g., super bosses), use a LEFT OUTER JOIN:

SELECT t1.empname AS Employee, COALESCE(t2.empname, 'No manager') AS Manager

FROM emp t1

LEFT OUTER JOIN emp t2 ON t1.mgrid = t2.empid;

### ****Subquery vs Joins****

Both subqueries and joins combine data from multiple tables, but they differ in approach and performance.

### ****Types of Subqueries****

1. **Single-row Subquery**: Returns a single value (e.g., =, <, <=).
2. **Multiple-row Subquery**: Returns multiple values (e.g., IN, EXISTS).
3. **Correlated Subquery**: The inner query depends on the outer query for its values.

### ****Subqueries vs. Correlated Subqueries****

* **Subquery (Nested Subquery)**: Executes once and returns results that the outer query can use.
* **Correlated Subquery**: Executes for each row processed by the outer query.

**Note** (**Correlated Subquery**:)

If you have **Table1** with 10 rows and **Table2** with 2 rows, and you're using a **correlated subquery**, the inner query (which operates on **Table2**) will execute once for each row in **Table1**.

**Performance Consideration**: Correlated subqueries are generally slower because they execute for each row of the outer query, whereas regular subqueries execute only once.

### Example of Correlated Subquery:

SELECT name,

(SELECT SUM(outsold) FROM productsales WHERE productid = products.id) AS total\_sales

FROM products;

### ****Set Operators (Union, Union All, Except, Intersect)****

Set operators combine results from two or more queries. Key rules:

* Expressions in each query must have the same number of columns with compatible data types.
* UNION removes duplicates, while UNION ALL retains all duplicates.

 EXCEPT it gives you the **rows from the first set that have no match in the second set**.

* INTERSECT returns records common to both queries.

Example:

SELECT product\_id FROM products

UNION

SELECT product\_id FROM sales;

#### **Join vs. Union**

##### **JOIN**

* Combines columns from multiple related tables.
* Increases the result set horizontally.

##### **UNION**

* Combines rows from multiple queries.
* Increases the result set vertically.

##### Key Differences:

* **Join**: Combines columns.
* **Union**: Combines rows, removes duplicates by default.

**Why is it called vertical?**

* The **vertical** expansion refers to how the **number of rows increases** when you stack multiple query results on top of each other, but the columns stay the same.

**Why is it called horizontal?**

* The **horizontal** expansion refers to how the **number of columns increases** when you add more data side by side, from multiple tables.

#### **Union vs. Union All**

* **UNION**: Removes duplicate records.
* **UNION ALL**: Retains duplicates, often faster than UNION.

#### **IN vs EXISTS**

**exists -- it stops early when it finds a match**

**in -- need to check through the entire dataset to create a list before comparing.**

Example of IN vs EXISTS:

SELECT product\_id FROM products WHERE product\_id IN (SELECT product\_id FROM sales);

SELECT product\_id FROM products WHERE EXISTS (SELECT 1 FROM sales WHERE products.product\_id = sales.product\_id);

#### **NOT IN vs EXCEPT**

* **NOT IN**: Can cause issues when the subquery results contain NULL values, as comparisons with NULL always return false.
* **EXCEPT**: Returns distinct records from the first result set that are not present in the second result set.

### ****CROSS APPLY vs OUTER APPLY****

**CROSS APPLY** is similar to an inner join.

It returns only the rows from the outer table (left table) that produce matching rows from the applied table (right side).

**OUTER APPLY** is similar to a left outer join.

It returns all the rows from the outer table (left table), even if the applied table (right side) does not return any matching rows.

CREATE FUNCTION GetProductReviews(@ProductID INT)

RETURNS TABLE

AS

RETURN (

SELECT Review

FROM Reviews

WHERE ProductID = @ProductID

);

SELECT p.ProductName, r.Review

FROM Products p

CROSS APPLY GetProductReviews(p.ProductID) r;

### ****Conclusion****

Joins, subqueries, set operators, and apply operators all serve distinct purposes in SQL queries. Joins are typically used to combine related tables, while set operators combine entire result sets. Understanding the nuances between INNER JOIN, OUTER JOIN, CROSS JOIN, UNION, and other operators helps optimize SQL performance and clarity.

## ****SQL Constraints****

* **NOT NULL**: Ensures a column cannot have a NULL value.
* **CHECK**: Validates data for a column at the time of insertion.
* **UNIQUE**: Ensures all values in a column are unique.
* **PRIMARY KEY**: Uniquely identifies rows in a table and does not allow NULL values.
* **FOREIGN KEY**: Ensures referential integrity between two tables.
* **DEFAULT**: Specifies a default value for a column if none is provided during insertion.

### ****Types of Data Integrity****

1. **Entity Integrity**: Ensured by primary keys and unique constraints.
2. **Domain Integrity**: Ensured through constraints like CHECK, NOT NULL, DEFAULT, and FOREIGN KEY.
3. **Referential Integrity**: Managed using FOREIGN KEY constraints.
4. **User-Defined Integrity**: Enforced through custom logic such as triggers, functions, and stored procedures.

### ****Primary Key vs Unique Key vs Foreign Key****

* **Primary Key**:
  + Uniquely identifies rows and does not allow NULL values.
  + Only one per table.
* **Unique Key**:
  + Allows NULLs (but only one NULL value per column).
  + Multiple unique keys are allowed in a table.
* **Foreign Key**:
  + Enforces referential integrity by linking two tables.
  + Multiple foreign keys can exist in a table.

### ****Cascade Referential Integrity Constraints****

The ON DELETE CASCADE and ON UPDATE CASCADE clauses ensure that when a row in the parent table is deleted or updated, the corresponding rows in the child tables are also deleted or updated. This ensures referential integrity between tables.

-- Example:

ALTER TABLE States ADD CONSTRAINT FK\_States\_Countries FOREIGN KEY (CountryID)

REFERENCES Countries (CountryID)

ON DELETE CASCADE;

### ****Wildcards (Pattern Matching) in SQL****

* The LIKE operator is used for pattern matching.
* To improve query performance, avoid using wildcards at the beginning of the search pattern (e.g., %value), as this forces a full table scan.

### ****CAST vs CONVERT****

* CAST is a standard SQL function, whereas CONVERT is specific to SQL Server.
* CONVERT supports formatting options for date types, while CAST does not.

### ****CASE Statement****

* The CASE statement is used for conditional logic within queries.
* It can be used in SELECT, WHERE, HAVING, ORDER BY, and INSERT statements.

### ****WHERE vs HAVING****

* **WHERE** filters rows before aggregation, whereas **HAVING** filters rows after aggregation (e.g., COUNT, SUM).
* **WHERE** cannot use aggregate functions, but **HAVING** can.

### ****DDL vs DML vs DCL****

* **DDL** (Data Definition Language) includes commands like CREATE, ALTER, and DROP.
* **DML** (Data Manipulation Language) includes commands like INSERT, UPDATE, DELETE.
* **DCL** (Data Control Language) includes commands like GRANT and REVOKE.

### ****Normalization vs Denormalization****

* **Normalization** reduces redundancy by organizing data into multiple tables, ensuring efficiency and minimizing anomalies.
* **Denormalization** combines data to improve read performance, often at the expense of redundancy.

### ****Local vs Global Variables****

* **Local Variables** are scoped to the current session or batch and are declared with DECLARE.
* **Global Variables** are predefined system variables with names beginning with @@ (e.g., @@SERVERNAME, @@ERROR).

### ****Pagination (SQL Server 2012)****

* The OFFSET-FETCH clause allows for pagination in SQL queries.

SELECT \* FROM DummyTable

ORDER BY DummyID

OFFSET @start ROWS

FETCH NEXT 10 ROWS ONLY;

### ****NULL Replacement Functions****

* **ISNULL()**: Replaces NULL with a specified value.
* **COALESCE()**: Returns the first non-NULL value in a list of expressions.
* **CASE**: Provides conditional replacement of NULL values based on custom conditions.

### ****Data Types in SQL****

* **Decimal**: Stores numeric data with precision (p) and scale (s).
* **GUID**: A globally unique identifier represented by UNIQUEIDENTIFIER. Generated using the NEWID() function.

### ****Clustered vs Non-Clustered Indexes****

* **Clustered Index**: Defines the physical storage order of data in the table. Each table can have only one clustered index.
* **Non-Clustered Index**: A separate structure that contains pointers to the data rows. A table can have multiple non-clustered indexes.

### ****DELETE vs TRUNCATE****

* **DELETE**: A DML command that removes rows based on a condition and can be rolled back.
* **TRUNCATE**: A DDL command that removes all rows and cannot be rolled back in certain situations (e.g., if the session is closed).

### ****Identity vs Sequence****

* **Identity**: Automatically generates sequential numbers and is tied to a specific table.
* **Sequence**: A standalone object used to generate sequential numbers that can be shared across multiple tables.

### ****Table Variables****

* Table variables are temporary structures stored in tempdb. They allow for primary keys, unique constraints, and checks but not foreign keys.

### ****Temporary Tables****

* **Local Temporary Tables**: Available only to the current session, prefixed with #.
* **Global Temporary Tables**: Available to all sessions, prefixed with ##.

### ****Common Table Expressions (CTEs) vs Temp Tables****

* **CTEs** are used for temporary result sets within a single query.
* **Temp Tables** are physical structures that exist for the duration of a session and can be used across multiple queries.

## ****Views****

A **view** in SQL Server is a virtual table based on the result set of a stored SQL query. Views do not store data physically, but rather provide a way to simplify queries, enhance security, and abstract complex data structures. A view can represent a subset of data from a table or a combination of data from multiple tables, and users can query it as if it were a table.

### ****Uses of Views****

1. **Subset of Data:** Views allow users to see only specific columns or rows of data without exposing the entire base table.
2. **Join and Simplify Multiple Tables:** Views can combine multiple tables, using joins, into a single virtual table, making it easier to interact with related data.
3. **Aggregation:** Views can include aggregate functions (e.g., SUM, AVG) to present summarized data, such as total sales or average ratings, making reporting simpler.
4. **Hiding Data Complexity:** Views can provide a simplified interface to complex queries, such as presenting partitioned data (e.g., by year or region), while hiding the complexity of underlying table structures.
5. **Security:** Views can restrict access to sensitive data by allowing users to query the view without having direct access to the underlying tables.

### ****Advantages of Views****

1. **No Physical Storage:** Views don't take up storage space like tables because they are virtual.
2. **Data Abstraction:** Views allow you to hide certain columns or rows from the underlying tables, offering a cleaner and more focused view of the data.
3. **Access Control:** Views can enforce restrictions, preventing users from performing data-modifying operations (insert, update, delete) directly on the view.

### ****Disadvantages of Views****

1. **Dependence on Base Tables:** If the base table(s) are dropped or altered, the view can become invalid or irrelevant.
2. **Performance Issues:** Views are recalculated whenever they are queried, which can result in slower performance, especially for complex views.
3. **Memory Usage:** Large or complex views can consume significant memory when they aggregate large volumes of data.

### ****Creating and Using Views****

A **view** is created using the CREATE VIEW statement, and it can be queried just like a regular table:

CREATE VIEW MyView AS

SELECT column1, column2

FROM MyTable

WHERE condition;

#### **Refreshing Views:**

In case the underlying tables have changed (such as schema changes), you can refresh the metadata of a view using the following command:

EXEC sp\_refreshview 'MyView';

#### **SCHEMABINDING:**

The WITH SCHEMABINDING option locks the underlying table's schema, preventing changes that could affect the view's structure. This is particularly useful when creating indexed views (discussed below).

CREATE VIEW MyView

WITH SCHEMABINDING

AS

SELECT column1, column2

FROM dbo.MyTable;

#### **Using** WITH CHECK OPTION**:**

The WITH CHECK OPTION ensures that the data modified through the view complies with the view's WHERE clause. For example, if the view only returns records where the status = 'Active', the WITH CHECK OPTION will prevent users from inserting or updating rows that do not meet this condition.

CREATE VIEW ActiveEmployees

AS

SELECT \* FROM Employees

WHERE status = 'Active'

WITH CHECK OPTION;

### ****Indexed Views (Materialized Views)****

**Indexed views** in SQL Server improve the performance of complex queries by creating a unique clustered index on the view. This allows the data in the view to be physically stored in the database, just like a regular table, with sorted and indexed data.

#### **Benefits of Indexed Views:**

1. **Improved Query Performance:** Since indexed views are materialized, queries that use the view can benefit from the pre-sorted and indexed data, reducing the need for expensive joins and aggregations.
2. **Used in OLAP Systems:** Indexed views are particularly useful in Online Analytical Processing (OLAP) environments where the data doesn't change frequently, but complex queries are run frequently.
3. **Data Integrity:** Changes to the base tables are automatically reflected in the indexed view, as it maintains an up-to-date version of the view data.

#### **Requirements for Indexed Views:**

1. **SCHEMABINDING:** The underlying tables must be bound to the view using WITH SCHEMABINDING to prevent changes to the schema that would affect the view.
2. **No Non-Deterministic Functions:** Functions like GETDATE() cannot be used in indexed views because they produce different results each time they are called.
3. **Limitations in View Definition:**
   * You cannot use EXISTS, NOT EXISTS, OUTER JOIN, COUNT(\*), MIN, MAX, subqueries, or UNION in indexed views.
   * You must use COUNT\_BIG(\*) instead of COUNT(\*) in views with GROUP BY.

#### **Creating Indexed Views:**

CREATE VIEW MyIndexedView

WITH SCHEMABINDING

AS

SELECT column1, column2, COUNT\_BIG(\*)

FROM dbo.MyTable

GROUP BY column1, column2;

CREATE UNIQUE CLUSTERED INDEX MyIndex ON MyIndexedView (column1, column2);

#### **Using the Indexed View:**

Once the indexed view is created and indexed, queries referencing the view can use it directly to benefit from improved performance:

SELECT \* FROM MyIndexedView WITH (NOEXPAND) WHERE column1 = 'value';

The WITH (NOEXPAND) hint forces SQL Server to use the clustered index of the view directly rather than expanding the view into its underlying queries.

Sure! Here's an organized and clear explanation comparing **regular views** and **indexed views**, as well as the concept of **view expansion** in SQL Server:

### ****Regular View vs. Indexed View****

#### 1. **Regular View:**

* **Definition**: A **regular view** is a virtual table defined by a SELECT query. It **does not store data** but only stores the query definition.
* **Data Handling**: Every time you query a regular view, SQL Server **executes the underlying query**. If the query involves complex operations such as joins, filters, or aggregations, these calculations are **repeated** every time the view is queried.
* **Performance Impact**: For **simple views**, the performance impact may be minimal. However, for **complex views** with joins, aggregations, or filters, performance can be suboptimal because SQL Server **recalculates** the results each time the view is accessed.

**Example**:

CREATE VIEW SalesSummary AS

SELECT ProductID, SUM(SaleAmount) AS TotalSales

FROM Sales

GROUP BY ProductID;

* Querying this view:

SELECT \* FROM SalesSummary;

* **What happens**: SQL Server recalculates the SUM(SaleAmount) and performs the GROUP BY operation every time you query the view.

#### 2. **Indexed View:**

* **Definition**: An **indexed view** (also called a **materialized view**) is a view with a **unique clustered index**. It **physically stores** the result of the query defined in the view.
* **Data Handling**: The results of the view's query, including any aggregation, are **precomputed** and stored. SQL Server automatically keeps this data up-to-date when the underlying tables are modified (insert, update, or delete operations).
* **Performance Impact**: Because the data is pre-aggregated and stored, querying an indexed view is **much faster**. There is no need to recalculate the aggregation or perform joins every time the view is accessed.

**Example**:

CREATE VIEW SalesSummaryIndexed

WITH SCHEMABINDING

AS

SELECT ProductID, SUM(SaleAmount) AS TotalSales

FROM Sales

GROUP BY ProductID;

-- Creating an indexed view with a clustered index

CREATE UNIQUE CLUSTERED INDEX IX\_SalesSummary

ON SalesSummaryIndexed (ProductID);

* Querying this indexed view:

SELECT \* FROM SalesSummaryIndexed;

* **What happens**: SQL Server retrieves the precomputed TotalSales from the indexed view, without recalculating the SUM(SaleAmount) or performing the GROUP BY.
* SQL Server **doesn’t need to recompute the sum**. It simply retrieves the precomputed values from the indexed view.

### Can I create a non-clustered index on the view?

Yes, you can create **non-clustered indexes** on the indexed view, but **you must first create a unique clustered index**. After that, non-clustered indexes can be added to improve query performance.

### ****View Expansion in SQL Server:****

**View**: When SQL Server expands a view, it executes the base query defined in the view.

SQL Server has to recalculate everything in the view every time it is queried.

**Indexed View**: An indexed view stores precomputed results and can greatly improve performance by eliminating the need to recalculate aggregations or join operations each time the view is queried.

Using **WITH (NOEXPAND):** If you have an indexed view, you can use the WITH (NOEXPAND) hint to force SQL Server to use the indexed view directly, avoiding the need to expand it into the underlying query and benefiting from faster performance. (some case index view can expansion that’s why we are forsing index view )

### ****The Difference: (view vs indexed view)****

| **Aspect** | **Regular View** | **Indexed View** |
| --- | --- | --- |
| **Data Storage** | No data is stored; only the query is stored. | Data is **physically stored** after being computed. |
| **Performance** | Needs to **recalculate** data each time queried. | Precomputed data is stored; **faster query performance**. |
| **Use Case** | Useful for simple queries or for logical views. | Useful for complex queries or aggregations where performance is critical. |
| **Update Behavior** | No updates to the view data; it reflects real-time data from the underlying tables. | Automatically updates when the underlying data changes. |
| **Data Calculation** | Data is calculated at query time (no preaggregation). | Data is pre-aggregated and stored in the indexed view. |
| **Use of WITH (NOEXPAND)** | Not needed. | Use WITH (NOEXPAND) to prevent view expansion and use indexed view directly. |

### ****Limitations and Considerations for Indexed Views****

* **Performance Trade-offs:** While indexed views can significantly improve read performance, they incur overhead during data modification (INSERT, UPDATE, DELETE) operations, as the indexed view must be updated to reflect the changes.
* **Use in OLAP Systems:** Indexed views are best suited for environments where data is not frequently updated, such as data warehouses and reporting databases.
* **SQL Server Editions:** Some optimizations related to indexed views may vary depending on the SQL Server edition, with Enterprise Edition benefiting more from this feature.

### ****Indexed Views vs. Regular Views in Context of**** sp\_refreshview****:****

* **Indexed Views**: You don't need to use sp\_refreshview for indexed views, because SQL Server **automatically updates** them when the underlying data changes (insert, update, delete) and when structural changes occur in the underlying tables.
* **Regular Views**: You need sp\_refreshview when the **schema of the underlying tables changes** (e.g., adding or removing columns) to keep the view's metadata in sync. However, the data in the view is **not stored**, so there’s no need for the procedure to handle data updates—just the metadata.

## Index

### ****Definition of Indexes:****

* An index is a database object that helps speed up the retrieval of data from a table, improving query performance.
* It provides a quick way to look up rows in a table without having to search through every row.

### ****Types of Indexes:****

1. **Clustered Index:**
   * Defines the physical order of data within the table.
   * A table can have only **one clustered index**.
   * The data rows are stored in the order of the clustered index key.
   * SQL Server creates a clustered index automatically on the primary key.
2. **Non-Clustered Index:**
   * Does not define the physical order of data.
   * Contains a copy of the indexed columns and a pointer to the actual data.
   * A table can have **multiple non-clustered indexes**.

### ****Features****

#### **2.2. Composite Index:**

* An index involving multiple columns.
* The order of columns is important for query optimization.
* Example: CREATE INDEX IX\_Name\_Age ON Users(LastName, FirstName);

#### **2.3. Covering Index (INCLUDE Clause):**

* Includes additional columns in the index to avoid lookup operations.
* Helps optimize queries by including all columns needed in the index.
* Example: CREATE NONCLUSTERED INDEX IX\_CoveringIndex ON Users(LastName) INCLUDE(FirstName, Age);

#### **2.4. Filtered Index:**

* Indexes only a subset of rows based on a WHERE condition.
* Optimizes performance for queries that target specific data.
* Example: CREATE INDEX IX\_FilteredIndex ON Users(LastName) WHERE Active = 1;

#### **2.5. Unique Index:**

* Ensures that all values in the indexed column are unique.

#### **2.6. Spatial Index:**

* Optimizes queries on spatial data types like geometry and geography.
* Example: CREATE SPATIAL INDEX IX\_SpatialIndex ON Locations(GeoData);

#### **2.7. XML Index:**

* Used to optimize queries on XML data types.
* Includes **Primary XML Index** and **Secondary XML Index**.
* Example: CREATE PRIMARY XML INDEX PXML\_Index ON Users(AddressXML);

### ****3. Index Maintenance****

#### **3.1. Online Index Operations:**

* Allows index creation, rebuilding, or dropping without locking the table, making it suitable for high-traffic tables.

#### **3.2. Fillfactor:**

* The percentage of space to leave free on each index page to minimize page splits.
* Helps manage fragmentation.
* Example: CREATE NONCLUSTERED INDEX IX\_IndexName ON Users(LastName) WITH (FILLFACTOR = 90);

#### **3.3. Statistics Update:**

* Keeps the SQL Server query optimizer informed of the data distribution within the tables.
* Can be updated using UPDATE STATISTICS.
* Example: UPDATE STATISTICS Users;

### ****4. Query Performance and Index Efficiency****

#### **4.1. Logical Reads vs Physical Reads:**

* **Logical Reads**: Data read from memory.
* **Physical Reads**: Data read from disk.

#### **4.2. RID Lookup vs Key Lookup:**

* **RID Lookup**: Occurs with heaps when SQL Server uses a non-clustered index to find data.
* **Key Lookup**: Occurs when SQL Server uses a non-clustered index on a clustered table to retrieve data.

#### **4.3. Indexes on Heaps vs Clustered Tables:**

* **Heap**: A table without a clustered index; data is stored unordered.
* **Clustered Table**: Has a clustered index and data rows are ordered according to the clustered index.

### **Lookup diff**

| **Scenario** | **Lookup Type** | **Description** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **1: Heap Table, Non-Clustered Index** | **RID Lookup** | The **non-clustered index** stores the **RID** of the rows in the heap. SQL Server performs a **RID lookup** to retrieve the full row from the heap. |

|  |  |  |
| --- | --- | --- |
| **2: Clustered Table, Clustered Index, Non-Clustered Index** | **Key Lookup** | The **non-clustered index** stores the **clustered index key (e.g., ID)**. SQL Server uses this **clustered index key** to perform a **Key Lookup** on the clustered index. **RID Lookup is not needed** here. |

|  |  |  |
| --- | --- | --- |
| **3.Heap Table, No Index** | **Full Table Scan** | Without any indexes, SQL Server performs a **full table scan**, and no lookups (RID or Key) are used. |

|  |  |  |
| --- | --- | --- |
| **4.Table Having Only Clustered Index** | **No Lookup** | The **clustered index** organizes the rows physically, so SQL Server can directly retrieve the rows using the clustered index key. No **Key Lookup** or **RID Lookup** is needed. |

### ****5. Clustered Index Design Considerations****

#### **Best Practices for Clustered Index Key:**

* **Short**: Narrow keys minimize index size.
* **Static**: Prefer columns that do not change often (e.g., IDENTITY).
* **Increasing**: Use auto-incrementing values.
* **Unique**: Ensure the key is unique to avoid the creation of hidden uniqueifiers.

### ****6. Clustered vs Non-Clustered Indexes****

* **Clustered Index**:
  + Defines the order of data.
  + Only one per table.
* **Non-Clustered Index**:
  + Separate from data storage.
  + Multiple indexes can exist on a table.

### ****7. Index Constraints and Restrictions****

#### **Primary Key Constraints:**

* Automatically creates a clustered index, unless one already exists.

#### **Non-Clustered Index Constraints:**

* Can be dropped or altered using T-SQL commands.
* Example: ALTER TABLE TableName DROP CONSTRAINT PK\_ConstraintName;

### ****8. Filtered Index vs Indexed Views****

| **Feature** | **Filtered Index** | **Indexed View** |
| --- | --- | --- |
| **Data Scope** | Indexes a subset of rows based on a condition | Can span multiple tables. |
| **Performance** | Lower overhead due to smaller data set. | Can significantly improve complex queries. |
| **Updateable** | Yes | Yes, but requires more resources. |

### ****9. Best Practices for Index Key Selection****

#### **GUID vs Character Columns:**

* **GUID**: Use cautiously as clustered index keys due to potential fragmentation.
* **Character columns**: Should be avoided as clustered index keys due to their size and collation.

#### **Appropriate Data Types:**

* **Best Data Types for Indexes**: Small, unique, and static data types like INT, BIGINT, or CHAR(20).
* Avoid large or random data types (e.g., VARCHAR(MAX), GUID).

### ****10. Non-Clustered Index on Heap vs Clustered Table****

#### **Heap**:

* A table without a clustered index.
* Non-clustered indexes will use a **RID Lookup** to retrieve data.

#### **Clustered Table**:

* Has a clustered index that defines the order of the data.
* Non-clustered indexes perform a **Key Lookup** to access data.

### ****11. INCLUDE Clause in Non-Clustered Indexes****

* **Purpose**: Allows additional columns to be included in the index without affecting the index key size.
* **Benefits**: Reduces the number of lookups and provides a covering index for queries.
* **Example**: CREATE NONCLUSTERED INDEX IX\_Employee\_LastName ON Employees(LastName) INCLUDE (FirstName, Age);

### ****12. Tables Without Indexes (Heaps)****

* **Heaps**: Tables without any indexes, where data is stored in unordered pages.
* SQL Server uses **Index Allocation Map (IAM)** pages to manage heap storage.

### ****13. Extents in SQL Server****

* **Uniform Extents**: Allocated to a single object (more efficient for large tables).
* **Mixed Extents**: Shared between multiple objects (efficient for smaller tables).

### ****14. Managing Index Fragmentation****

#### **Fragmentation Types:**

* **Internal Fragmentation**: Wasted space inside index pages due to incomplete fills.
* **External Fragmentation**: The physical order of data is not aligned with the index.

#### **Index Maintenance**:

* **Rebuilding**: Drops and recreates the index (recommended for fragmentation >30%).
* **Reorganizing**: Reorganizes the index pages without dropping the index (recommended for fragmentation between 5-30%).

### 14.1 How to Address Fragmentation from Page Splits?

#### 1.fill factor

The default fill factor in SQL Server is 100 (i.e., index pages are completely filled).

This is typically suitable for read-heavy environments where index pages are unlikely to undergo frequent changes.

A lower fill factor can help reduce index fragmentation caused by page splits

To manage fragmentation, you might consider adjusting the fill factor along with regular index maintenance strategies like rebuilding or reorganizing indexes.

CREATE INDEX idx\_order\_date ON orders(order\_date)

WITH (FILLFACTOR = 70);

#### 2.Index Maintenance:

Rebuild or Reorganize indexes regularly to reduce fragmentation.

ALTER INDEX idx\_customer\_name ON customers

REBUILD WITH (FILLFACTOR = 80);

ALTER INDEX idx\_order\_date

ON orders

REBUILD;

#### 3.Monitor Fragmentation:

select \*,avg\_fragmentation\_in\_percent from sys.dm\_db\_index\_physical\_stats(NULL, NULL, NULL, NULL, 'DETAILED')

**Internal vs external fragmentation:**

Internal fragmentation occurs when there is unused space within a page, usually caused by deletes or updates.

External fragmentation occurs when index pages are not stored contiguously on disk, often due to page splits or inefficient storage allocation.

### ****15. Fill Factor in Indexes****

* **Fill Factor**: Determines how much space to leave free on each index page.
  + **Lower Fill Factors** (e.g., 70%) help reduce fragmentation in write-heavy workloads.
  + **Higher Fill Factors** (e.g., 100%) fill pages to capacity but may increase fragmentation over time.
  + Example: CREATE NONCLUSTERED INDEX IX\_Employee\_LastName ON Employees(LastName) WITH (FILLFACTOR = 80);

### ****16. Duplicate Indexes in SQL Server****

* **Issue**: Duplicate indexes waste storage and increase maintenance costs.
* **Solution**: Identify and remove duplicate indexes.
* Query to identify duplicate indexes:

SELECT OBJECT\_NAME(IX.OBJECT\_ID) AS Table\_Name,

IX.name AS Index\_Name,

COL.name AS Column\_Name

FROM sys.indexes IX

INNER JOIN sys.index\_columns IC

ON IX.object\_id = IC.object\_id AND IX.index\_id = IC.index\_id

INNER JOIN sys.columns COL

ON IC.object\_id = COL.object\_id AND IC.column\_id = COL.column\_id

WHERE OBJECTPROPERTY(IX.OBJECT\_ID, 'IsUnique') = 0

ORDER BY Table\_Name, Index\_Name;

This version ensures that all major aspects of SQL Server indexing are covered comprehensively and is organized into clearly defined sections. Each section is now focused on a specific aspect of index types, maintenance, and best practices, ensuring the topics are fully addressed.

## Transaction

### ****What is a Transaction and What Are ACID Properties?****

* **Transaction**: A transaction in a database management system (DBMS) is a logical unit of work that contains one or more operations (e.g., insert, update, delete) executed as a single unit. A transaction is considered successful only if all operations within it succeed; if one operation fails, the entire transaction is rolled back.
* **ACID Properties**:
  + **Atomicity**: Ensures that all operations within a transaction are treated as a single, indivisible unit. Either all operations are successfully committed, or if any operation fails, the transaction is completely rolled back (all or nothing).
  + **Consistency**: Ensures that a transaction brings the database from one valid state to another. It preserves data integrity by ensuring that all database constraints, triggers, and rules are respected.
  + **Isolation**: Guarantees that the operations of a transaction are isolated from other transactions. The intermediate state of a transaction is not visible to other transactions until the transaction is completed (committed).
  + **Durability**: Once a transaction is committed, its effects are permanent and are not lost, even in the case of system failures.

### ****Best Practices for SQL Server Transactions****

1. **Narrow the scope of the transaction**: Limit the duration of the transaction to avoid holding locks for long periods.
2. **Retrieve data before starting the transaction**: If possible, gather all required data before initiating the transaction to minimize the transaction's workload.
3. **Access only necessary data**: Ensure that only the essential data is accessed during the transaction.
4. **Avoid waiting for user input**: Don’t prompt for user input inside a transaction, as this can introduce delays and increase the risk of deadlocks.
5. **Choose the appropriate transaction mode and isolation level**: Ensure the transaction mode (Autocommit, Implicit, or Explicit) and isolation level (Read Uncommitted, Read Committed, Repeatable Read, etc.) match the specific needs of the transaction.

### ****Modes of Transactions in SQL Server****

1. **Autocommit Mode** (Default mode): Each individual SQL statement is considered a separate transaction. If a statement succeeds, it is committed automatically; if it fails, it is rolled back.
2. **Implicit Transaction Mode**: In this mode, every data manipulation language (DML) operation implicitly starts a transaction. The programmer must explicitly commit or roll back the transaction at the end.
3. **Explicit Transaction Mode**: The transaction is explicitly started and ended by the user. This mode provides full control over the transaction's scope and handling.

### ****Savepoints in Transactions****

* **Savepoints** allow you to set intermediate points within a transaction. If necessary, the transaction can be rolled back to a specific savepoint instead of rolling back the entire transaction.

Example:

BEGIN TRANSACTION

INSERT INTO Person VALUES('Mickey', 'Mouse')

SAVE TRANSACTION InsertStatement

DELETE FROM Person WHERE PersonID = 3

ROLLBACK TRANSACTION InsertStatement

COMMIT

### ****Tracking Active Transactions****

* **@@TRANCOUNT**: This function shows the number of open transactions for the current session. It increments when a new transaction is started and decrements when a transaction is committed or rolled back.
* **XACT\_STATE()**: This function provides the state of the transaction. It returns:
  + **1**: Active, commit-able transaction.
  + **0**: No active transaction.
  + **-1**: Active transaction, but it is uncommittable due to an error.
* **DBCC OPENTRAN**: This command shows information about active transactions in the current database.

### ****Locking and Concurrency Control****

* **Locks**: Locks are mechanisms used to ensure data integrity during concurrent access. SQL Server uses different types of locks for various operations, including:
  + **Shared (S)**: Multiple transactions can read the data.
  + **Exclusive (X)**: Only one transaction can modify the data.
  + **Update (U)**: A special type of lock to prevent deadlocks, allowing a transaction to acquire a lock for updates.
  + **Schema Locks** (Sch-M and Sch-S): Used for schema changes like creating or altering indexes.
* **Lock Escalation**: This is the process of converting many row-level or page-level locks into a table-level lock to reduce memory usage. It occurs when the number of locks exceeds a certain threshold.
* **TablockX Hint**: The TABLOCKX hint forces an exclusive lock on the entire table during a transaction, which prevents other transactions from modifying the table until the transaction is committed.

### ****Deadlock and Its Management****

* **Deadlock**: Occurs when two or more transactions hold locks on resources and each waits for the other to release its lock. SQL Server detects deadlocks and resolves them by terminating one of the transactions, called the **deadlock victim**.
* **Deadlock Priority**: SQL Server uses deadlock priority to decide which transaction to terminate. The transaction with the lowest deadlock priority is chosen as the victim. Deadlock priority can be set using SET DEADLOCK\_PRIORITY.
* **Deadlock Detection**: SQL Server uses a lock monitor that runs periodically (every 5 seconds by default) to check for deadlocks.

### ****Isolation Levels****

[Understanding SQL Server Isolation Levels through Examples | Interface Technical Training (interfacett.com)](https://www.interfacett.com/blogs/understanding-isolation-levels-sql-server-2008-r2-2012-examples/?unapproved=118825%20HYPERLINK%20%22https://www.interfacett.com/blogs/understanding-isolation-levels-sql-server-2008-r2-2012-examples/?unapproved=118825&moderation-hash=431598806d2a7c4d734e17b1b3a37276%22%20HYPERLINK)

The **isolation level** determines how transaction integrity is visible to other transactions and how data is locked during the transaction. SQL Server supports the following isolation levels:

1. **Read Uncommitted**: Allows dirty reads, meaning a transaction can read data that has been modified but not yet committed.
2. **Read Committed**: The default level; ensures that a transaction reads only committed data.
3. **Repeatable Read**: Ensures that if a transaction reads a value, no other transaction can modify or insert new rows that would affect the value during the transaction.
4. **Serializable**: The strictest isolation level. It prevents other transactions from inserting, updating, or deleting rows that would affect the current transaction.
5. **Snapshot**: Provides a consistent view of the data as it was at the start of the transaction, regardless of changes made by other transactions.
6. **Read Committed Snapshot**: Combines Read Committed with the ability to use row versions for a consistent view without blocking reads.

### ****Write-Ahead Logging (WAL)****

* **WAL**: This concept ensures that before any data is written to the database, a log record of the change is written to the transaction log. This helps ensure data consistency and durability.

### ****Transaction Log Sequence Numbers (LSN)****

* Every transaction in SQL Server is associated with a unique Log Sequence Number (LSN) that helps track changes in the transaction log. This is used for ensuring transaction durability and recovery.

### ****Latch vs Lock****

* **Latch**: A synchronization mechanism used internally by SQL Server to protect access to memory structures (e.g., buffer cache). Latches are not controlled by the user or DBA.
* **Lock**: A mechanism used for controlling access to database objects and data during transactions. Locks are managed by the DBMS and can be influenced by developers and DBAs.

### ****Buffer Pool and Memory Management****

* SQL Server's buffer pool caches data pages to optimize disk I/O. Pages in the buffer pool can either be "clean" (not modified) or "dirty" (modified but not yet written to disk). Checkpoints ensure that dirty pages are written to disk periodically to facilitate crash recovery.

### ****Batch Mode on Rowstore****

* Introduced in SQL Server 2019, **Batch Mode on Rowstore** improves query performance by processing large sets of rows in batches rather than one row at a time. This is particularly useful for aggregation and sorting operations.

## SQL Architecture

<https://www.guru99.com/sql-server-architecture.html>

<https://www.c-sharpcorner.com/article/learn-about-sql-server-architecture/>

## Partitioning

**My requirement •**

Partition 1: Covers all data before 2015 .

Partition 2: Covers 2015 data, with the fg\_2015 filegroup.

Partition 3: Covers 2016 data, with the fg\_2016 filegroup.

Partition 4: Covers 2017 and onward (and assigned to the fg\_2017 filegroup).,

### **1.With PRIMARY Filegroup -** Creating New partition table

**-- Step 1: Create Filegroups for each year**

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2015;

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2016;

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2017;

**-- Step 2: Create Partition Function**

CREATE PARTITION FUNCTION partitionfunctionname (DATE)

AS RANGE RIGHT FOR VALUES

(

'2015-01-01', -- Boundary for 2015 (Partition 2 starts here)

'2016-01-01', -- Boundary for 2016 (Partition 3 starts here)

'2017-01-01' -- Boundary for 2017 (Partition 4 starts here)

);

**-- Step 3: Create Partition Scheme**

CREATE PARTITION SCHEME partitionschemaname

AS PARTITION partitionfunctionname

TO

(PRIMARY, -- Partition 1: Data before 2015 (using PRIMARY filegroup)

fg\_2015, -- Partition 2: Data for 2015

fg\_2016, -- Partition 3: Data for 2016

fg\_2017); -- Partition 4: Data for 2017 and onward

**-- Step 4: Create Partitioned Table**

CREATE TABLE tablename

(

saleid INT IDENTITY(1,1) PRIMARY KEY,

productid INT,

quantityid INT,

saleamount DECIMAL(9,2),

saledate DATETIME

)

ON partitionschemaname (saledate); -- Partitioning by SaleDate

**-- Step 5: Add Files to Filegroups**

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2015, FILENAME = 'C:\sql\_data\sales\_fg\_2015.ndf') TO FILEGROUP fg\_2015;

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2016, FILENAME = 'C:\sql\_data\sales\_fg\_2016.ndf') TO FILEGROUP fg\_2016;

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2017, FILENAME = 'C:\sql\_data\sales\_fg\_2017.ndf') TO FILEGROUP fg\_2017;

### 2. **Without PRIMARY Filegroup (Using Dedicated Filegroup for Pre-2015 Data)**

**-- Step 1: Create Filegroups for each year**

ALTER DATABASE databasename01

ADD FILEGROUP fg\_before\_2015; -- Dedicated filegroup for data before 2015

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2015;

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2016;

ALTER DATABASE databasename01

ADD FILEGROUP fg\_2017;

**-- Step 2: Create Partition Function**

CREATE PARTITION FUNCTION partitionfunctionname (DATE)

AS RANGE RIGHT FOR VALUES

(

'2015-01-01', -- Boundary for 2015 (Partition 2 starts here)

'2016-01-01', -- Boundary for 2016 (Partition 3 starts here)

'2017-01-01' -- Boundary for 2017 (Partition 4 starts here)

);

**-- Step 3: Create Partition Scheme**

CREATE PARTITION SCHEME partitionschemaname

AS PARTITION partitionfunctionname

TO

(fg\_before\_2015, -- Partition 1: Data before 2015 (using dedicated filegroup)

fg\_2015, -- Partition 2: Data for 2015

fg\_2016, -- Partition 3: Data for 2016

fg\_2017); -- Partition 4: Data for 2017 and onward

**-- Step 4: Create Partitioned Table**

CREATE TABLE tablename

(

saleid INT IDENTITY(1,1) PRIMARY KEY,

productid INT,

quantityid INT,

saleamount DECIMAL(9,2),

saledate DATETIME

)

ON partitionschemaname (saledate); -- Partitioning by SaleDate

**-- Step 5: Add Files to Filegroups**

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_before\_2015, FILENAME = 'C:\sql\_data\sales\_fg\_before\_2015.ndf') TO FILEGROUP fg\_before\_2015;

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2015, FILENAME = 'C:\sql\_data\sales\_fg\_2015.ndf') TO FILEGROUP fg\_2015;

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2016, FILENAME = 'C:\sql\_data\sales\_fg\_2016.ndf') TO FILEGROUP fg\_2016;

ALTER DATABASE databasename01

ADD FILE (NAME = sales\_fg\_2017, FILENAME = 'C:\sql\_data\sales\_fg\_2017.ndf') TO FILEGROUP fg\_2017;

### ****When to Use Each Approach****

* **Use the PRIMARY filegroup (Approach 1)** if:
  + Your dataset before 2015 is **small** and you don’t need any special performance optimizations.
  + The **data before 2015 is frequently queried** or accessed. In this case, managing everything in the PRIMARY filegroup simplifies management.
  + You want to **minimize complexity** and avoid managing multiple filegroups.
* **Use a separate filegroup for older data (Approach 2)** if:
  + You have **large amounts of historical data** (before 2015) that is **rarely accessed**.
  + You want to optimize storage, for example, storing older data on cheaper disks.
  + You need **better I/O performance** for current data and want to isolate old data to avoid impacting performance.
  + You want **granular backups and restores**, especially if you plan to exclude or back up older data less frequently.

### ****Understanding the SWITCH Operation****

 **Partitioned Tables Only**:

* ALTER TABLE SWITCH works **only** between **partitioned tables**.
* Both the **source** and **target tables** must use the **same partition scheme** and **partition function**.
* The **columns** must be in the **same order**, with **identical data types** in both tables.
* Both tables must be **partitioned by the same column** (e.g., SalesDate).

 **No Row-by-Row Movement**:

* The **SWITCH operation** doesn't move data **row by row**.
* It simply **switches the metadata**, making the data from the source table part of the target partitioned table.
* This is much **more efficient** than using INSERT INTO as it avoids copying rows.

 **Efficient Data Transfer**:

* Using SWITCH is **faster** and more efficient, as it **avoids the overhead** of physically moving the data.

### Move Data from a Non-Partitioned Table to a Partitioned Table

**-- Step 1: Create the partitioned table (target table)**

CREATE TABLE new\_partitioned\_table

(

saleid INT IDENTITY(1,1) PRIMARY KEY,

productid INT,

quantityid INT,

saleamount DECIMAL(9,2),

saledate DATETIME

)

ON SalesPartitionScheme (saledate); -- Partitioning by saledate

**-- Step 2: Create a temporary partitioned table**

CREATE TABLE temp\_partitioned\_table

(

saleid INT IDENTITY(1,1) PRIMARY KEY,

productid INT,

quantityid INT,

saleamount DECIMAL(9,2),

saledate DATETIME

)

ON SalesPartitionScheme (saledate); -- Same partition scheme and function

**-- Step 3: Insert data into the temporary partitioned table**

INSERT INTO temp\_partitioned\_table (productid, quantityid, saleamount, saledate)

SELECT productid, quantityid, saleamount, saledate

FROM old\_table;

**-- Step 4: Use SWITCH to move the data to the final partitioned table**

ALTER TABLE temp\_partitioned\_table

SWITCH TO new\_partitioned\_table;

-- Optional: Drop the temporary partitioned table (if no longer needed)

DROP TABLE temp\_partitioned\_table;

### the MDF and LDF files serve specific purposes:

1. **MDF (Primary Data File)**: This is the primary data file for a database, which contains the schema (tables, indexes, etc.) and actual data for the database. It is required to have exactly one **MDF** file per database.
2. **LDF (Log File)**: This file contains the transaction log information for the database, which helps in recovering the database and maintaining its ACID properties (Atomicity, Consistency, Isolation, Durability). The **LDF** file is also required to have at least one file per database.

#### **Why Use** .NDF **Files for Filegroups (and not** .MDF**/**.LDF**)**:

* **Filegroups**: A filegroup is a logical grouping of database files used to physically store data. Filegroups can contain multiple data files (i.e., .MDF, .NDF) but not transaction log files (.LDF).
* **Adding .NDF Files**:
  + When you create additional **filegroups** (like fg\_2015, fg\_2016, fg\_2017 in your case), you **add new data files** for each of them, but these files are of type **.NDF** (Secondary Data Files).
  + These **.NDF** files are used to **store data** in the new partitions associated with those filegroups. SQL Server automatically uses the **PRIMARY filegroup**'s **MDF file** for the default storage, and additional filegroups use their respective **.NDF files**.
  + **.NDF** files are specifically used to **extend storage** for partitioned tables or tables with large datasets, whereas **.MDF** holds the primary data and **.LDF** holds the transaction log.

## Execution Plan

### Reusing Execution Plans in SQL Server

Reusing execution plans is crucial for performance when frequently executing stored procedures with varying parameters. SQL Server stores the execution plans in memory for re-use, minimizing the need to generate new plans each time a query is executed. This is beneficial because plan generation is a costly process. SQL Server uses the plan cache to store and reuse the execution plans when queries are executed again, which can significantly improve performance.

### Reading the SQL Execution Plan

The correct way to read an execution plan is to start from the **right side to the left side** and from the **top to the bottom**. The **left-most SELECT operator** represents the final result of the query, while the other operators (to the right) indicate how the data is processed and manipulated at each step.

1. **Right to Left**: You should follow the flow from the right-most part of the plan, as this is often where data is first accessed (e.g., via table scans or index scans).
2. **Top to Bottom**: Each row represents a step in the query execution, starting from scanning data or applying filters to the final result set.

### Can the Estimated Execution Plan be Trusted?

The **Estimated Execution Plan** is an approximation of how SQL Server will execute the query, based on the current statistics. While it can be trusted under normal conditions, the accuracy depends on whether the **statistics** (which guide the optimizer) are up to date. If statistics are not updated, the execution plan may not reflect the actual execution path. The **Actual Execution Plan**, which is generated after executing the query, will provide a more accurate picture of how SQL Server executed the query.

### Fat and Thin Arrows in Execution Plans

In an execution plan, **fat arrows** indicate a large number of rows being processed. These are followed by **thin arrows**, which indicate the flow of a smaller number of rows to the next operator. This often suggests that data is being filtered down at each step. If you see many fat arrows followed by thin arrows, it could indicate:

* **Missing indexes**, leading to inefficient data access.
* **Data multiplication issues** or inefficiencies in joins, such as when multiple rows from one table are being unnecessarily matched to many rows in another.

### Types of Execution Plans

1. **Estimated Execution Plan**: This plan is generated without actually executing the query. It predicts the execution path based on the optimizer’s estimates and is useful for quick checks.
2. **Actual Execution Plan**: This plan is generated after the query is executed and contains actual runtime statistics, such as the number of rows processed and time taken for each operation.

### How Execution Plans Work

The **execution plan** is generated after a SQL query is submitted, and it is created in the following phases:

1. **SQL Server Relational Engine**:
   * **Parser**: Validates the query syntax.
   * **Algebrizer**: Ensures the query is semantically correct, checking if tables, columns, and other objects exist.
   * **Optimizer**: The query optimizer analyzes the query, using statistics to choose the most efficient execution plan. The optimizer considers various strategies and their associated costs before choosing the optimal plan.
2. **SQL Server Storage Engine**:
   * The **execution engine** executes the plan, fetching and processing the data as determined by the plan.
   * Once executed, the result is returned to the client.

### Live Execution Statistics

Introduced in SQL Server 2016, **Live Execution Statistics** allows real-time monitoring of query performance while it is running. This is particularly useful for long-running queries, as it provides visibility into how data flows between operators in the execution plan. It can help identify the source of performance bottlenecks without waiting for the query to complete. Permissions required to use this feature include:

* **SHOWPLAN**: To generate the execution plan.
* **VIEW SERVER STATE**: To see live statistics.
* **Query execution permissions**.

### Operators in Execution Plans

SQL Server execution plans include various operators, each responsible for different aspects of query processing:

1. **Table Scan**: Scans an entire table without using an index.
2. **Index Scan**: Scans an index to retrieve data.
3. **Index Seek**: Uses an index to find specific rows, more efficient than a scan.
4. **RID Lookup**: Looks up the row identifier in a heap table when using a non-clustered index.
5. **Key Lookup**: Retrieves data from the clustered index when using a non-clustered index.
6. **Sort**: Sorts the data based on a specified order.
7. **Stream Aggregate**: Aggregates data streams, often with GROUP BY.
8. **Compute Scalar**: Performs a computation on each row, such as a mathematical operation or concatenation.
9. **Concatenation**: Combines multiple result sets into one.
10. **Hash Match Join**: Efficiently joins two large datasets by dividing data into "buckets" using a hash algorithm.
11. **Merge Join**: Uses sorted data for a fast join operation.
12. **Nested Loops Join**: Efficient for small outer tables with an index on the inner table.
13. **Filter**: Filters data based on a predicate expression.

### Operators Explanation:

1. **RID Lookup**: If a table lacks a clustered index, it is considered a heap, and data is accessed via row identifiers (RIDs).
2. **Key Lookup**: In a clustered index, data retrieval involves looking up the primary key or clustered index to fetch data.
3. **Stream Aggregate**: Efficient for aggregated queries if the data is already sorted.
4. **Compute Scalar**: Used for computations like calculations or string manipulations in SELECT statements.
5. **Concatenation**: Often seen in queries with UNION ALL to combine result sets.
6. **Hash Match Join**: Creates a hash table for joining two tables and then probes the hash table to find matching records.
7. **Hash Match Aggregate**: Used for aggregations on unsorted data, building a hash table to process records efficiently.

### Conclusion

Understanding execution plans in SQL Server is crucial for identifying performance bottlenecks and optimizing queries. By interpreting the various operators and understanding the flow of data through the plan, you can troubleshoot issues such as missing indexes, inefficient joins, and poorly optimized queries. The **Actual Execution Plan** is the most reliable for post-query analysis, while the **Estimated Execution Plan** is useful for pre-query assessments.

## Slow Running Query/Query Optimization

### Identify slow-running queries

SELECT r.session\_id, r.status, r.total\_elapsed\_time, r.cpu\_time, r.wait\_time, r.command

FROM sys.dm\_exec\_requests r

CROSS APPLY sys.dm\_exec\_sql\_text(r.sql\_handle)

WHERE status != 'sleeping'

### Key Concepts for Query Optimization:

1. **Activity Monitor**:
   * Use to identify expensive queries by sorting them based on CPU time, logical reads, or elapsed time. This helps pinpoint which queries are consuming excessive resources and need optimization.
2. **Query Store**:
   * Enables capturing and analyzing query performance over time. By enabling it, you can monitor top resource-consuming queries, filter them by metrics like duration, CPU, or logical reads, and even track execution statistics (like minimum, maximum, or average metrics) to find performance bottlenecks.
3. **Query Statistics**:
   * Enable SET STATISTICS TIME, IO ON to capture CPU and I/O statistics for each table involved in a query. This provides detailed insight into resource consumption during execution.
4. **Execution Plan Analysis**:
   * Look for "Key Lookup" operations in execution plans, which are often a sign of inefficient queries. Adding relevant columns to indexes can help remove unnecessary lookups.
   * Differences between estimated and actual row counts indicate potential issues like outdated statistics, which can be resolved by updating statistics.
5. **TSQL Optimization Techniques**:
   * **OR in Join Predicate/WHERE Clause**: Avoid using OR in multiple column conditions as it might lead to inefficient query plans. This can often be optimized by restructuring the query or using UNION.
   * **Wildcard Searches**: Use more selective patterns (e.g., FOR% instead of %FOR%) to improve performance by reducing the search space.
   * **Large Write Operations**: For bulk data loads or writes, use batch processing to minimize locking and I/O overhead.
   * **Missing Indexes**: Ensure proper indexes are in place. Missing indexes can cause unnecessary table scans and slow down operations like UPDATE, DELETE, and INSERT.
   * **Use of Temporary Tables**: Move metadata or lookup data into temporary tables to optimize large joins or complex queries.
6. **Query Hints**:
   * **NOLOCK**: Use with caution to avoid dirty reads.
   * **RECOMPILE**: Forces the query to recompile each time, useful when dealing with parameter sniffing issues where a single execution plan is inefficient for different input parameters.
   * **Merge/Hash/Loop Join Hints**: Use these to force the query optimizer to choose a specific join method when you know a particular one will perform better.
   * **OPTIMIZE FOR**: Helps in situations where the query optimizer makes bad decisions due to parameter sniffing. By forcing the optimizer to use a specific value for a parameter, it can generate a more optimal plan.
7. **SQL Injection Prevention**:
   * Using **sp\_executesql** with parameters helps mitigate SQL injection attacks. By parameterizing queries, input values are treated as data rather than executable code, preventing malicious SQL injection.
8. **Index Types**:
   * **Columnstore Index**: Excellent for analytical workloads as it stores data in columns, allowing for better compression and performance when accessing only certain columns. It works by fetching entire segments of data, improving query performance for large datasets.
   * **Row-based Index**: Uses traditional row-based storage and requires fetching full data pages into memory. While useful for transactional queries, it can be less efficient for analytical queries.
9. **Parameter Sniffing**:
   * When a query plan is compiled based on a parameter's value, subsequent executions with different parameters can result in suboptimal performance if the execution plan is not suitable for all possible parameter values. Solutions include:
     + **OPTION (RECOMPILE)**: Forces the query to recompile every time, ensuring the plan is optimized for the current parameter value.
     + **OPTION (OPTIMIZE FOR)**: Allows you to provide a specific value for a parameter to optimize the plan for that scenario.

### Additional Considerations:

* **Database and Table Design**: Ensure the tables are normalized appropriately and that indexes are used effectively. Poor schema design can lead to slow queries and complex execution plans.
* **Monitoring and Logging**: Continuously monitor queries with tools like SQL Server Profiler, Extended Events, or third-party tools to identify slow-running queries and optimize them proactively.
* **Recompiling Stored Procedures**: Sometimes recompiling stored procedures helps optimize performance, especially if outdated statistics or parameter sniffing is causing issues. However, use this sparingly as it can introduce additional overhead.

By combining the above optimization techniques with continuous monitoring and proactive tuning, you can significantly improve the performance of SQL queries in production environments.

## Windows Function (Aggregate/Rank/Value)

### ****Window Functions Overview****:

Window functions allow you to perform calculations across a set of table rows related to the current row within the result set, without collapsing the rows into a single output row. They retain the identities of individual rows while allowing you to compute cumulative or other grouped values.

### ****Types of Window Functions****:

1. **Aggregate Window Functions**:
   * These functions operate over a set of rows within a window and return an aggregated value for each row.
   * Common examples: SUM(), MAX(), MIN(), AVG(), COUNT().

**Example of Aggregate Window Function**:

SELECT order\_id, order\_date, customer\_name, city, order\_amount,

SUM(order\_amount) OVER(PARTITION BY city) AS grand\_total

FROM [dbo].[Orders]

Here, the SUM(order\_amount) will calculate the total order amount for each city (partitioned by the city) without collapsing the result set.

**Important Note**:

* + COUNT() in a window function doesn't support the DISTINCT keyword. So, to count distinct values in a window function, use a workaround (like COUNT(DISTINCT <column>) in a GROUP BY query).

1. **Ranking Window Functions**: These assign a rank to rows within a partition based on an order.
   * ROW\_NUMBER(): Sequential number for rows based on the ORDER BY clause.
   * RANK(): Ranks rows and skips rank numbers for ties.
   * DENSE\_RANK(): Ranks rows without skipping numbers for ties.
   * NTILE(n): Divides the result set into n approximately equal parts and assigns a bucket number to each row.

**Example of Ranking Function**:

SELECT StudentName, Subject, Marks,

RANK() OVER(PARTITION BY StudentName ORDER BY Marks DESC) AS Rank

FROM ExamResult

ORDER BY StudentName, Rank;

This query ranks students based on their marks, with ties being given the same rank (using RANK()).

1. **Value Window Functions**: These functions allow you to access data from different rows (either previous or next rows) in the result set.
   * LAG(): Provides the value of a previous row (used for comparing current and previous values).
   * LEAD(): Provides the value of a subsequent row (used for comparing current and next values).
   * FIRST\_VALUE(), LAST\_VALUE(): Access the first and last value in the partition.

**Example of LAG() Function**:

SELECT order\_id, customer\_name, city, order\_amount, order\_date,

LAG(order\_date, 1) OVER(ORDER BY order\_date) AS prev\_order\_date

FROM [dbo].[Orders]

This query retrieves the order\_date from the previous row for each row in the result set, based on the order of order\_date.

1. **Percentage Calculation**: In SQL, percentage calculations can be done by dividing the value of interest by the total and multiplying by 100.

**Simple Percentage Calculation**:

SELECT val,

val \* 100 / (SELECT SUM(val) FROM Scores) AS Percentage\_of\_Total

FROM Scores

**Example with Common Table Expression (CTE)**:

WITH ProductCTE AS (

SELECT SupplierID, SUM(UnitPrice) AS TotalPriceProduct,

(SUM(UnitPrice) \* 100) / SUM(SUM(UnitPrice)) OVER () AS Percentage\_of\_Total\_Price

FROM Products

GROUP BY SupplierID

)

SELECT \*

FROM ProductCTE;

This example calculates each supplier’s percentage of total price from the entire Products table.

### ****Main Advantage of Window Functions Over Regular Aggregate Functions****:

* **No Grouping**: The primary advantage of window functions is that they allow calculations like aggregates, ranking, or value comparisons without grouping the rows. Each row retains its identity, and you can add aggregated values alongside each individual row.
* **More Flexibility**: Window functions provide more flexibility by allowing you to compute running totals, row comparisons, and ranks, while still retaining the individual data row structure.

### ****Practical Use Cases for Window Functions****:

* **Running Totals**: Compute running totals over a partitioned dataset (e.g., calculating cumulative sales for each salesperson).
* **Time-Based Calculations**: Use LAG() or LEAD() to analyze data trends over time (e.g., comparing current and previous day’s sales).
* **Ranking**: Rank results within partitions, like finding top performers in each department or ranking exam results within different subjects.
* **Windowed Percentages**: Calculate percentages of totals, like finding a product’s share of total sales.

### ****Considerations****:

* Window functions require a **PARTITION BY** clause (optional) to define the partitioning of data. If omitted, the entire dataset is considered as a single partition.
* The **ORDER BY** clause inside window functions determines the order of calculations (like ranking and time-based functions).

### Conclusion:

SQL window functions are extremely powerful for performing calculations across rows without collapsing them, making them more efficient and flexible compared to regular aggregate functions. By using window functions, you can calculate things like running totals, ranks, and value comparisons more efficiently and intuitively within your queries.

## Delete Duplicate rows in Different ways:

CREATE TABLE Employee

(

[ID] INT identity(1,1),

[FirstName] Varchar(100),

[LastName] Varchar(100),

[Country] Varchar(100),

)

SELECT [FirstName],

[LastName],

[Country],

COUNT(\*) AS CNT

FROM [SampleDB].[dbo].[Employee]

GROUP BY [FirstName],

[LastName],

[Country]

HAVING COUNT(\*) > 1;

Ex 1:

DELETE FROM [SampleDB].[dbo].[Employee] -- SELECT \* FROM [SampleDB].[dbo].[Employee]

WHERE ID NOT IN

(

SELECT MAX(ID) AS MaxRecordID

FROM [SampleDB].[dbo].[Employee]

GROUP BY [FirstName],

[LastName],

[Country]

);

Ex 2:

WITH CTE([firstname],

[lastname],

[country],

duplicatecount)

AS

(SELECT [firstname],

[lastname],

[country],

ROW\_NUMBER() OVER(PARTITION BY [firstname], [lastname],[country] ORDER BY id) AS DuplicateCount

FROM [SampleDB].[dbo].[employee]

)

DELETE FROM CTE WHERE DuplicateCount > 1; -- SELECT \* FROM CTE;

Ex 3:

SELECT E.ID,

E.firstname,

E.lastname,

E.country,

T.rank

FROM [SampleDB].[dbo].[Employee] E

INNER JOIN

(

SELECT \*,

RANK() OVER(PARTITION BY firstname,

lastname,

country

ORDER BY id) rank

FROM [SampleDB].[dbo].[Employee]

) T ON E.ID = t.ID;

DELETE E

FROM [SampleDB].[dbo].[Employee] E

INNER JOIN

(

SELECT \*,

RANK() OVER(PARTITION BY firstname,

lastname,

country

ORDER BY id) rank

FROM [SampleDB].[dbo].[Employee]

) T ON E.ID = t.ID

WHERE rank > 1;

## New Features in SQL Server 2016, 2017, 2019, and 2022

Here is a comprehensive list of new features and improvements introduced in SQL Server versions 2016, 2017, 2019, and 2022, with a focus on developer-relevant features like string functions, performance improvements, and data management.

### ****SQL Server 2016 New Features****:

1. **DROP IF EXISTS**:
   * Simplifies dropping objects if they already exist.
   * Example:

DROP SCHEMA IF EXISTS TMP;

DROP VIEW IF EXISTS dbo.SQLShack;

1. **STRING\_SPLIT**:
   * A function for splitting a string into rows based on a delimiter.
   * Example:

DECLARE @string VARCHAR(800) = '001,002,003,004';

SELECT \* FROM STRING\_SPLIT(@string, ',');

1. **DATEDIFF\_BIG**:
   * Used for calculating differences between dates with larger values than INT can handle (e.g., differences in milliseconds).
   * Example:

SELECT DATEDIFF\_BIG(MILLISECOND, GETDATE()-100, GETDATE());

1. **AT TIME ZONE**:
   * Converts datetime or datetime2 to datetimeoffset with a specific time zone.
   * Example:

SELECT GETDATE() AS [Current Time],

CONVERT(datetimeoffset, GETDATE()) AT TIME ZONE 'Central European Standard Time';

1. **Always Encrypted**:
   * Encrypts sensitive data, ensuring encryption at both rest and in transit.
   * Uses two types of keys: Column Encryption Keys and Master Key.
2. **JSON Support**:
   * SQL Server 2016 introduces native support for JSON, making it easy to parse, store, and query JSON data.
   * Example:

SELECT column, expression

FROM table

FOR JSON AUTO;

1. **Dynamic Data Masking**:
   * Protects sensitive data by masking values in query results.
   * Example:

ALTER TABLE MyTable ALTER COLUMN MyColumn MASKED WITH (FUNCTION = 'default()');

1. **PolyBase**:
   * Allows integration with external data sources like Hadoop and Azure Blob Storage, enabling querying across different data systems.
2. **Row-Level Security**:
   * Restricts access to rows in a database based on user context.
   * Example:

CREATE SECURITY POLICY fn\_security

ADD FILTER PREDICATE MyFunction ON MyTable;

1. **Temporal Tables**:
   * Helps store historical data and track changes over time (Audit trail).
   * Example:

SELECT \* FROM MyTable FOR SYSTEM\_TIME BETWEEN '2022-01-01' AND '2022-12-31';

### ****SQL Server 2017 New Features****:

1. **SQL Server on Linux**:
   * SQL Server 2017 brings native support for Linux, allowing SQL Server to run on Linux-based environments.
2. **Automatic Plan Correction**:
   * SQL Server can now automatically detect and correct issues with query plans using the Query Store.
3. **Adaptive Query Processing**:
   * Introduces new features like interleaved execution and batch mode memory grant feedback to enhance query processing.
4. **Graph Database Features**:
   * Introduces graph database features for modeling data as nodes and edges.
   * Example:

CREATE TABLE Person (ID INT PRIMARY KEY, Name NVARCHAR(100));

CREATE TABLE Knows (FromPersonID INT, ToPersonID INT, RelationshipType NVARCHAR(100));

1. **STRING\_AGG**:
   * A new aggregation function for concatenating row values into a single string.
   * Example:

SELECT STRING\_AGG(column\_name, ',') AS concatenated\_values

FROM MyTable;

1. **Resumable Index Rebuild**:
   * Allows resumption of index rebuild operations, reducing downtime.
2. **Scalar UDF Inlining**:
   * Improves performance by automatically inlining scalar UDFs into the main query.
3. **Enhanced JSON Support**:
   * New JSON functions like JSON\_MODIFY() and OPENJSON() allow for easier manipulation of JSON data.

### ****SQL Server 2019 New Features****:

1. **Big Data Clusters**:
   * SQL Server 2019 introduces Big Data Clusters, enabling SQL Server to integrate with Hadoop, Spark, and Kubernetes for big data processing.
2. **Intelligent Query Processing**:
   * Enhances performance with features like scalar UDF inlining, batch mode on rowstore, and adaptive joins.
3. **Always Encrypted with Secure Enclaves**:
   * Secure Enclaves extend the functionality of Always Encrypted to allow computations on encrypted data without exposing it.
4. **UTF-8 Support**:
   * SQL Server 2019 supports UTF-8 encoded data, enabling better internationalization and data interoperability.
5. **Accelerated Database Recovery (ADR)**:
   * Improves database recovery speed by introducing a new recovery process that minimizes rollback times.
6. **PolyBase Enhancements**:
   * Additional connectors to allow integration with external data sources like Oracle, MongoDB, and others.
7. **Query Store Enhancements**:
   * The Query Store is enhanced with a wait stats feature, improved execution statistics, and improved performance.
8. **STRING\_AGG Enhancements**:
   * Improves the aggregation functionality for string concatenation.
9. **Memory-Optimized TempDB Metadata**:
   * TempDB is optimized to use memory for storing metadata, improving performance for workloads that heavily use TempDB.

### ****SQL Server 2022 New Features****:

1. **Azure Integration Enhancements**:
   * Deeper integration with Azure for hybrid cloud workloads, enabling seamless querying of cloud-based data sources using PolyBase.
2. **Ledger**:
   * A blockchain-like feature to store tamper-evident data in SQL Server, ideal for financial and auditing use cases.
3. **Query Store Enhancements**:
   * New features such as automatic switching to read-only mode, advanced troubleshooting capabilities, and better historical query tracking.
4. **Resilient Memory-Optimized Tables**:
   * Enhanced durability for memory-optimized tables, ensuring better performance and reliability for transactional systems.
5. **SQL Server on Kubernetes**:
   * Full support for running SQL Server in a Kubernetes containerized environment, enhancing flexibility in modern DevOps pipelines.
6. **Built-in SQL Diagnostics**:
   * SQL Server 2022 introduces new diagnostic tools to help analyze system health and diagnose performance issues.
7. **Improved In-Memory OLTP**:
   * Enhancements to In-Memory OLTP (Hekaton) to improve performance for transactional workloads, particularly with large volumes of data.
8. **STRING\_SPLIT Enhancements**:
   * Enhancements to STRING\_SPLIT in SQL Server 2022 for better performance and new options.

### ****New String Functions Across SQL Server Versions****:

1. **SQL Server 2016**:
   * **STRING\_SPLIT**: Splits a string into rows by a specified delimiter.
   * Example:

SELECT value

FROM STRING\_SPLIT('a,b,c,d', ',');

1. **SQL Server 2017**:
   * **STRING\_AGG**: Concatenates values from multiple rows into a single string.
   * Example:

SELECT STRING\_AGG(column\_name, ', ') AS concatenated\_values

FROM MyTable;

1. **SQL Server 2019**:
   * **STRING\_AGG**: Enhanced with options like ORDER BY to control the order of concatenated strings.
   * Example:

SELECT STRING\_AGG(column\_name, ', ') WITHIN GROUP (ORDER BY column\_name)

FROM MyTable;

1. **SQL Server 2022**:
   * **STRING\_SPLIT**: Further enhancements to STRING\_SPLIT for better performance and functionality.

### ****Conclusion****:

SQL Server versions from 2016 through 2022 have introduced numerous powerful features for developers, including enhancements to string manipulation functions, improved performance, and new ways to handle data. Features like **STRING\_AGG**, **Always Encrypted**, **PolyBase**, and **Query Store** are especially valuable for modern data-driven applications, while **Temporal Tables**, **Row-Level Security**, and **Ledger** provide robust data governance and security capabilities. With each version, SQL Server has become a more capable platform for both traditional and big data applications.

## String Function

Ltrim -- removes blanks on the left hand side of the given character expression

Rtrim -- remove blanks on the right hand side of the given character expression

Lower – Convert the character into Lower case

Upper -- Convert the character into Upper case

Reverse -- reverse all the character in the given string expression

Len -- returns the count of total character in the given string expression and excluding the blanks at end of the expression

Left -- returns the specified number of character from the left hand side of the given character expression

Right --

SubString -- returns the part of the string from the given spression ex. Substing( ‘ragul,2,3) o/p agu

Replicate -- replicate returns the repeates the given string for the specified number of times

Space -- returns number of spaces , specified by the no of spaces arguments

Replace -- replaces all occurances of a specified string value with another string values

Stuff -- delete the specified length of character and insert another set of character at a specified starting point

Ex. Stuff(‘ragul’,2,3,’\*\*\*’) o/p R\*\*\*l

CharIndex – returns the string position of the specified expression in a character string. Ex charindex(‘@’,’rag@gamil.com’ o/p: 4

PatIndex -- returns the string position of the First Occurance of a pattern in a specified expression

Ascii

Char

### STUFF vs REPLACE function

**STUFF** function is used to overwrite existing characters.

syntax, STUFF (string\_expression, start, length, replacement\_characters),

string\_expression is the string that will have characters substituted, start is the starting position,

length is the number of characters in the string that are substituted,

and replacement\_characters are the new characters interjected into the string.

**REPLACE** function to replace existing characters of all occurrences.

syntax REPLACE (string\_expression, search\_string, replacement\_string),

where every incidence of search\_string found in the string\_expression will be replaced with replacement\_string.

## Date/date Part/time function

**Date Part**

Date Parts an integer which represents the specified part of a date.

DATENAME (*date part*, *value*)

SELECT DATEPART (YEAR, GETDATE ()) as Year, -- 2005

       DATEPART (WEEK, GETDATE ()) as Week, -- 33

       DATEPART (DAYOFYEAR, GETDATE ()) as DayOfYear, -- 222

       DATEPART (MONTH, GETDATE ()) as Month, -- 8

       DATEPART (DAY, GETDATE ()) as Day, -- 10

       DATEPART (WEEKDAY, GETDATE ()) as WEEKDAY -- 2

**DATENAME**

DateName is a very useful function that is used to return various parts of a date such as the name of the month, or day of the week corresponding to a particular date.

DATENAME (year, GETDATE ()) as Year, --2005

DATENAME (week, GETDATE ()) as Week, -- 33

       DATENAME (dayofyear, GETDATE ()) as DayOfYear, -- 222

       DATENAME (month, GETDATE ()) as Month, -- August

       DATENAME (day, GETDATE ()) as Day, -- 10

       DATENAME (weekday, GETDATE ()) as WEEKDAY – Monday

**DATEADD**

SELECT DATEADD (DAY, 30, GETDATE ())

**DATEDIFF**

SELECT DATEDIFF (DAY, '01/01/2009', GETDATE ())

SELECT **ISDATE** ('07/44/09')

SELECT **MONTH** (getdate ()),

SELECT **DAY** (getdate ()),

SELECT **YEAR** (getdate ())

------

DECLARE @dates TABLE (date DATE);

-- setting the first and the last date in the month given by date

SET @date = '2020/05/12';

SET @start\_date = DATEFROMPARTS(YEAR(@date ), MONTH(@date ), '01');

SET @end\_date = EOMONTH(@date);

INSERT INTO @dates(date) VALUES (@loop\_date);

SET @loop\_date = DATEADD(DAY, 1, @loop\_date);

----

IF DATEPART(DW,GETDATE())= 2

## ****Dynamic SQL****:

Dynamic SQL allows you to build and execute SQL queries dynamically at runtime. It is especially useful when the structure of the query or parameters needs to be determined dynamically. This is done using the EXEC or sp\_executesql statement.

* **Example**: Suppose you want to dynamically query a table based on different conditions, such as selecting columns or filtering data based on user input.

DECLARE @sql NVARCHAR(MAX)

DECLARE @TableName NVARCHAR(50) = 'Employees'

DECLARE @ColumnName NVARCHAR(50) = 'Salary'

SET @sql = N'SELECT ' + @ColumnName + ' FROM ' + @TableName

EXEC sp\_executesql @sql

In this example, the SQL query is built dynamically based on table and column names, which are provided at runtime.

## ****SQL Agent****:

SQL Server Agent is a component of SQL Server used for automating tasks, like running queries, executing jobs, or scheduling maintenance tasks. SQL Agent can manage and schedule:

* Jobs
* Alerts
* Operators
* Schedules

SQL Server Agent is typically used for recurring tasks, such as backups, report generation, or batch processing.

**Example**: To create a SQL Server Agent job:

1. Open SQL Server Management Studio (SSMS).
2. In Object Explorer, expand the SQL Server Agent node, right-click **Jobs**, and select **New Job**.
3. Define job steps, schedules, and alerts.

**Example Job to execute a stored procedure**:

EXEC msdb.dbo.sp\_add\_job

@job\_name = N'JobName';

EXEC msdb.dbo.sp\_add\_jobstep

@job\_name = N'JobName',

@step\_name = N'Step1',

@subsystem = N'TSQL',

@command = N'EXEC dbo.MyProcedure;',

@database\_name = N'MyDatabase';

EXEC msdb.dbo.sp\_add\_schedule

@schedule\_name = N'DailySchedule',

@freq\_type = 4, -- daily frequency

@active\_start\_time = 090000; -- 9 AM

EXEC msdb.dbo.sp\_attach\_schedule

@job\_name = N'JobName',

@schedule\_name = N'DailySchedule';

EXEC msdb.dbo.sp\_add\_jobserver

@job\_name = N'JobName',

@server\_name = N'(local)';

## ****Pivot****:

Pivoting in SQL is used to convert data from rows to columns. This can be especially useful when you want to summarize or aggregate data, such as turning monthly sales data into columns for each month.

**Example**: Here’s an example of pivoting a table of sales data to show sales by month:

SELECT \*

FROM (

SELECT Month, SalesAmount

FROM Sales

) AS SourceTable

PIVOT (

SUM(SalesAmount)

FOR Month IN ([January], [February], [March], [April])

) AS PivotTable;

This will create a table where each month (January, February, March, April) is a column and the SUM(SalesAmount) is displayed for each month.

## ****In-Memory OLTP (Online Transaction Processing)****:

In-Memory OLTP, also known as **Hekaton**, is a feature in SQL Server that optimizes data processing by keeping tables and indexes in memory. This can drastically speed up transactions for high-performance workloads, such as real-time analytics or financial applications.

* **Example**: To create an In-Memory table, you must first enable the **In-Memory OLTP** feature and create memory-optimized tables.

-- Create a memory-optimized filegroup

ALTER DATABASE [YourDatabase]

ADD FILEGROUP [MemoryOptimized] CONTAINS MEMORY\_OPTIMIZED\_DATA;

-- Create a memory-optimized table

CREATE TABLE dbo.MyMemoryOptimizedTable

(

ID INT PRIMARY KEY NONCLUSTERED,

Name NVARCHAR(100),

CreatedDate DATETIME

) WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

## ****Job to Email Status****:

SQL Server Agent can be used to send email notifications when jobs complete. This is useful for monitoring job execution, errors, or completion status.

**Steps to configure email notification**:

1. **Enable Database Mail**: Set up Database Mail in SQL Server to send email notifications.
   * Go to **Object Explorer** → **Management** → **Database Mail**.
   * Right-click and choose **Configure Database Mail** to set up an SMTP profile.
2. **Configure SQL Server Agent** to send email notifications:
   * Right-click **SQL Server Agent** in SSMS → **Properties** → **Alert System** → Enable mail profile.
3. **Set up the Job to send an email**:
   * Open the **Job Properties** window, and under the **Notifications** tab, you can set it to email on success, failure, or completion.
   * **Example**:

EXEC msdb.dbo.sp\_add\_jobserver

@job\_name = 'JobName';

1. **Sending Email on Job Completion**:
   * You can configure a step in the job to send an email when it finishes:

EXEC msdb.dbo.sp\_notify\_operator

@profile\_name = 'YourProfile', -- Your mail profile

@email\_address = 'recipient@example.com',

@subject = 'Job Execution Status',

@body = 'The job has completed successfully.';

### Key Takeaways:

* **Dynamic SQL** helps execute SQL queries based on runtime conditions.
* **SQL Server Agent** automates tasks and job scheduling.
* **Pivot** transforms row data into columnar form to summarize and analyze data.
* **In-Memory OLTP** improves performance by storing tables in memory for fast transactions.
* **Job to Email Status** allows you to send notifications about job success, failure, or completion.

## Questions

### Replication Types

* **Snapshot Replication**: Used when data changes infrequently. It takes periodic snapshots of data and replicates them from a main server to branch servers.
* **Merge Replication**: Allows bidirectional updates between a central database and various remote sources, useful when multiple systems need to simultaneously update information.
* **Transactional Replication**: Ideal for frequently changing data, as it replicates real-time updates from the publisher to subscribers.

### DBCC Commands

* **DBCC CHECKALLOC**: Verifies that pages are correctly allocated.
* **DBCC CHECKDB**: Ensures the integrity of database objects and their indexes.
* **DBCC CHECKFILEGROUP**: Checks the physical integrity of file groups.

### Functions

* **SIGN()**: Returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.
* **UPDATE STATISTICS**: Updates the statistics for an object to help the SQL Server query optimizer choose the most efficient execution plan.

### Concepts

* **Entities**: Group similar data in a structured form, like tables.
* **Collation**: Defines rules for comparing and sorting character data.
* **NoSQL Databases**: Typically used for complex queries like graph and geospatial analysis, which are less feasible in SQL databases.

### DMV (Dynamic Management Views)

* **sys.dm\_exec\_query\_stats (qs)**: Provides statistics about cached query execution, such as elapsed time and execution count.
* **sys.dm\_exec\_procedure\_stats (ps)**: Gives details on stored procedures, including their execution times and worker times, useful for diagnosing performance bottlenecks.
* **sys.dm\_exec\_sql\_text**: Returns the SQL text of a query from its handle.
* **sys.dm\_exec\_query\_plan**: Shows the execution plan for a cached query.

### SSRS (SQL Server Reporting Services) Functions

* **LEFT() and RIGHT()**: Used for extracting parts of strings, like masking part of a phone number or email address. Examples:
  + Masking part of a phone number: =Left(Fields!Mobile\_number.Value,2) & StrDup(5,"\*") & RIGHT(Fields!Mobile\_number.Value,3)
  + Masking part of an email address:

=Replace(Fields!email\_id.Value,

Mid(Fields!email\_id.Value,3,Instr(Fields!email\_id.Value,"@")-3),

StrDup(LEN(Mid(Fields!email\_id.Value,3,Instr(Fields!email\_id.Value,"@")-3)),"\*"))

* **Rownumber()**: Used to create row numbers within a report.
* **Caching**: Stores a temporary copy of the report, helpful for reducing the load on the server when refreshing frequently accessed reports.

### Common SQL Statements and Procedures

* **sp\_rename**: Renames database objects like tables or views.

EXEC sp\_rename 'OldViewName', 'NewViewName';

* **sp\_helptext**: Retrieves the text definition of a stored procedure or view.

EXEC sp\_helptext 'Employee\_View4';

* **WITH SCHEMABINDING, WITH ENCRYPTION, WITH CHECK OPTION**: Various options to enforce schema binding, encrypt views, and restrict changes (e.g., preventing modification of views by non-admin users).

### WHILE Loop Example

A simple WHILE loop to demonstrate conditional logic:

DECLARE @Counter INT;

SET @Counter = 1;

WHILE (@Counter <= 20)

BEGIN

IF @Counter % 2 = 1

BEGIN

SET @Counter = @Counter + 1;

CONTINUE;

END

PRINT 'The counter value is = ' + CONVERT(VARCHAR, @Counter);

SET @Counter = @Counter + 1;

END

This loop increments a counter and prints its value, skipping odd numbers using CONTINUE.

### Key Takeaways:

* **Stored procedures** are useful for encapsulating multiple SQL statements and reducing network traffic by executing them in one batch.
* **Replication** types in SQL Server allow different configurations for data distribution, ensuring consistency across systems.
* **DBCC** commands are essential for checking and maintaining database integrity.
* **DMVs** provide deep insights into server performance and query execution, helping DBAs troubleshoot issues.
* **SSRS functions** are helpful for string manipulation in reports, such as masking sensitive data or controlling row numbers.
* **WHILE loops** allow for iterative processing, often used in batch operations or for controlling logic flow.

### General Syntax for a SELECT Statement (covering all options)

SELECT select\_list

[INTO new\_table] -- Creates a new table with the result set

FROM table\_source

[WHERE search\_condition] -- Filters records based on condition

[GROUP BY group\_by\_expression] -- Groups records by specified columns

[HAVING search\_condition] -- Filters grouped records based on condition

[ORDER BY order\_expression [ASC | DESC]] -- Sorts the result by specified columns (optional, default is ASC)

### Authentication Modes in SQL Server

SQL Server supports **two authentication modes**:

1. **Windows Authentication Mode**:
   * Only Windows accounts are used for authentication.
   * Users are authenticated using their Windows credentials.
2. **Mixed Mode Authentication** (SQL Server and Windows Authentication):
   * Allows both Windows accounts and SQL Server logins to authenticate users.
   * This is typically used in environments where users need to authenticate using SQL Server logins as well as Windows accounts.

#### Changing Authentication Mode:

* The authentication mode can be changed through SQL Server Management Studio (SSMS) or by using the sp\_configure system stored procedure.

Steps to change the authentication mode:

1. Open SSMS.
2. Right-click on the server instance in the **Object Explorer**.
3. Click **Properties**.
4. In the **Security** tab, select either **Windows Authentication Mode** or **SQL Server and Windows Authentication Mode**.
5. Click **OK**, and restart the SQL Server service for changes to take effect.

Alternatively, you can change it using the sp\_configure stored procedure:

EXEC sp\_configure 'authentication mode', 1; -- 1 for mixed mode, 0 for Windows authentication

RECONFIGURE;

### Software Development Life Cycle (SDLC)

SDLC is a methodology used for creating high-quality software, following a series of phases:

1. **Requirement Analysis**: Understanding and documenting the requirements of the system.
2. **Planning**: Defining the project scope, schedule, resources, and cost estimation.
3. **Software Design**: Creating the architecture and detailed design for the system.
4. **Software Development**: Writing the actual code for the system.
5. **Testing**: Verifying the system through various testing methods to ensure it works as intended.
6. **Deployment**: Deploying the system into the production environment.

### Waterfall vs. Agile Methodologies

* **Waterfall Model**:
  1. A linear, sequential approach.
  2. Changes are difficult once the project is underway.
  3. Testing is performed only after the build phase.
* **Agile Model**:
  1. Focuses on continuous iterations and feedback.
  2. Offers flexibility to make changes during development.
  3. Testing is integrated into each development cycle.

### Basic Functions of System Databases in SQL Server:

1. **Master**: Holds information about all databases on the SQL Server instance and is essential for SQL Server startup.
2. **msdb**: Stores backup information, SQL Server Agent jobs, alerts, and other administrative data.
3. **model**: Template database used when creating new user databases.
4. **tempdb**: Stores temporary objects such as temporary tables, global/local temporary tables, and stored procedures.
5. **resource**: A read-only database containing all system objects that come with SQL Server. It is not user-accessible directly.

### Where Are SQL Server Usernames and Passwords Stored?

SQL Server stores usernames and passwords in the system catalog views:

* **sys.server\_principals**: Contains information about server-level principals.
* **sys.sql\_logins**: Contains information about SQL logins.

### What is Replication and Database Mirroring?

* **Replication**: Replication involves copying and distributing data from one database (publisher) to other databases (subscribers).
* **Database Mirroring**: Involves two copies of a single database, one as the principal (active) and the other as the mirror (passive). Changes made to the principal are mirrored to the secondary copy for high availability.

### SQL Server Agent:

SQL Server Agent is a component used to automate and schedule administrative tasks such as backups, database maintenance, and running queries or jobs.

### OLTP vs. OLAP Workloads:

* **OLTP (Online Transaction Processing)**:
  + Focuses on transactional operations such as insert, update, and delete.
  + Typically used for operational databases where data integrity is crucial.
  + **Indexes**: OLTP systems use indexes to speed up transactional queries (e.g., primary keys and foreign keys).
* **OLAP (Online Analytical Processing)**:
  + Used for data analysis and reporting (e.g., in data warehouses).
  + Queries often involve aggregations, filtering, and joining large datasets.
  + **Indexes**: OLAP systems benefit from using many indexes on read-heavy tables, such as star schema designs (fact and dimension tables).

### Fact Tables vs. Dimension Tables:

* **Fact Tables**: Contain quantitative data for analysis, often with numeric values that can be aggregated (e.g., sales amounts).
* **Dimension Tables**: Contain descriptive data that provides context to the fact tables (e.g., customer names, product categories).

### Data Warehouse Characteristics:

* **Subject-Oriented**: Data is organized by subject areas like sales or finance.
* **Integrated**: Data is collected from multiple sources and stored in a unified format.
* **Time-Variant**: Stores historical data to allow trend analysis over time.
* **Non-Volatile**: Once data is loaded into a data warehouse, it does not change (historical data remains unchanged).

### ETL Process in Data Warehousing:

* **Extraction**: Pulling data from various source systems.
* **Transformation**: Cleaning and transforming the data to fit the target schema.
* **Loading**: Loading the transformed data into the data warehouse.

### Data Mining:

Data mining involves analyzing data from different perspectives to extract patterns, trends, and valuable insights from large datasets.

### RDBMS (Relational Database Management System):

An RDBMS is a system for managing relational databases where data is stored in tables with predefined relationships. Key features include:

* **Data Integrity**: Ensures data accuracy and consistency.
* **ACID Properties**: Ensures transactions are processed reliably.
* **SQL Support**: Uses SQL for querying and managing the data.

### Entity-Relationship Diagram (ERD):

An **Entity-Relationship Diagram (ERD)** is a visual representation of entities (tables) in a database and their relationships. It helps in database design.

### Linked Server:

A **Linked Server** allows querying data from a remote SQL Server or other databases as though it were a local database. It is configured using sp\_addlinkedserver and sp\_addlinkedserverlogin.

### Collation and Types:

Collation defines the rules for sorting and comparing string data. It affects case sensitivity, accent sensitivity, and other aspects of character comparison.

* **Types of Collation Sensitivity**:
  1. **Case Sensitivity**: Differentiates between uppercase and lowercase characters (e.g., 'A' vs. 'a').
  2. **Accent Sensitivity**: Differentiates characters based on accents (e.g., 'e' vs. 'é').
  3. **Kana Sensitivity**: Differentiates between Hiragana and Katakana in Japanese.
  4. **Width Sensitivity**: Differentiates between half-width and full-width characters (e.g., 'a' vs. 'ａ').

### TCP/IP Port for SQL Server:

SQL Server typically runs on port **1433** for TCP/IP connections. This can be changed in SQL Server Configuration Manager under the **TCP/IP Properties**.

### Index Configurations in SQL Server:

A table in SQL Server can have the following index configurations:

1. **No Indexes**: The table has no indexes (less efficient for large datasets).
2. **Clustered Index**: Data is physically stored in the order of the clustered index (there can only be one clustered index per table).
3. **Clustered Index + Nonclustered Indexes**: A table can have one clustered index and multiple nonclustered indexes.
4. **Nonclustered Index**: A separate structure that holds pointers to the actual data (more efficient for read-heavy operations).
5. **Multiple Nonclustered Indexes**: A table can have multiple nonclustered indexes to improve performance on various queries.

### ****Log Shipping:****

* **Log shipping** is an automated process for backing up database transaction logs and restoring them on a standby server.
* It ensures disaster recovery by synchronizing logs between the primary server and a secondary server.
* It’s supported only in **Enterprise Editions** of SQL Server.
* The transaction log from the primary server is backed up and restored on the secondary server at defined intervals.

### ****Count Records in a Table:****

* To get an accurate count of records in a table:
  1. SELECT COUNT(\*) FROM table1; – Direct count of all rows.
  2. SELECT \* FROM table1; – Returns all rows (but inefficient for large tables).
  3. SELECT rows FROM sysindexes WHERE id = OBJECT\_ID('table1') AND indid < 2; – Provides row count from system tables, more efficient in some cases.

### ****Replication Types in SQL Server:****

* **Replication** is used to copy data between different databases or servers.
  1. **Snapshot Replication:** Takes a point-in-time snapshot of the data and applies it to subscribers.
  2. **Transactional Replication:** Replicates changes as they occur in real-time, with options for immediate or queued updates.
  3. **Merge Replication:** Allows updates to be made at both the publisher and subscriber ends, then merges the changes.

### ****Difference Between Functions and Stored Procedures:****

* **Functions:**
  + Used to return a value and can be embedded in SELECT, WHERE, HAVING, etc.
  + Cannot perform DML operations (INSERT, UPDATE, DELETE).
* **Stored Procedures:**
  + Can perform DML operations, and can return results or perform other actions.
  + Cannot be used directly in SELECT statements like functions.

### ****Database Relationships:****

* **One-to-One:** One row in Table A corresponds to exactly one row in Table B (e.g., a person’s passport).
* **One-to-Many:** One row in Table A can correspond to multiple rows in Table B (e.g., a customer can have multiple orders).
* **Many-to-Many:** Multiple rows in Table A correspond to multiple rows in Table B (e.g., students and courses, where each student can enroll in many courses and vice versa).

### ****Denormalization:****

* Denormalization is the process of combining tables and introducing redundancy to improve query performance, especially in OLAP systems. It reduces the need for complex joins and enhances performance at the cost of increased storage and potential data integrity risks.

### ****Transaction Logs and Cursors:****

* **Transaction Log** tracks all changes made in the database, providing the ability to undo or redo operations.
* **Cursors** allow row-by-row processing, but are inefficient with large datasets. Alternatives like **temporary tables** or **CASE statements** should be considered when possible to improve performance.

### ****User-Defined Functions (UDF):****

* **Scalar UDFs:** Return a single value.
* **Inline UDFs:** Return a table and can be used in joins or subqueries.
* **Multi-statement UDFs:** Allow multiple SQL statements and return a table.

### ****Key Concepts:****

* **Candidate Key:** A key that uniquely identifies a row in a table.
* **Alternate Key:** A candidate key that was not chosen as the primary key.
* **Composite Key:** A key that combines multiple columns to uniquely identify a row.

### ****SQL Server Tips for Performance:****

* **Stored Procedures** are preferred over ad-hoc queries for better performance and security. They are precompiled and stored in the database.
* **Optimizing Queries:** Techniques like using the right indexes, reducing the transaction area, and recompiling stored procedures after changes in indexes can significantly improve performance.

### ****CTE (Common Table Expressions):****

* A CTE is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE query.
* It simplifies complex queries, especially those that require recursion.

### ****Deadlocks:****

* A **deadlock** occurs when two or more transactions are blocked forever because they are each waiting for the other to release a lock.

### ****SQL Server Indexes:****

* **Clustered Index:** Sorts and stores the data rows based on the index key. Only one clustered index can exist per table.
* **Non-clustered Index:** A separate structure from the table that contains a pointer to the data.
* **In-Memory Tables:** Have specialized indexes (hash and range) and can improve performance significantly for certain workloads.

### group by cube vs group by rollup n?

SELECT Region, Product, SUM(Amount) AS TotalSales

FROM Sales

GROUP BY CUBE(Region, Product);

### Surrogate Key

 **Surrogate Key** (customerkey): In OLAP, you use the surrogate key (customerkey) in dimension tables and fact tables for efficient processing, historical tracking, and data integrity. It’s typically an **auto-incrementing integer**.

 **Business Key** (customerid): The business key (customerid) is used to link the dimension table to the operational system, but it is **not the primary key** in the dimension table.

 **Fact Table**: The **fact table** typically stores the **surrogate key** (customerkey) from the **dimension table** as a foreign key, not the business key (customerid).

#### **Summary of SCD in OLTP vs OLAP**

| **Aspect** | **OLTP System (Transactional)** | **OLAP System (Data Warehouse)** |
| --- | --- | --- |
| **Purpose** | Real-time transactions and operations | Historical data analysis and reporting |
| **Normalization** | Highly normalized (avoids redundancy) | Denormalized (easier for querying) |
| **Handling Changes** | Overwrite (Type 1) | Preserve history (Type 2, Type 3) |
| **SCD Type** | Type 1 (Overwrite) | Type 1, Type 2 (Tracking History), Type 3 |
| **Focus** | Current state of data | Historical state and trends over time |
| **Example of Data** | Latest customer address only | All versions of customer address |
| **Handling of Historical Data** | Not tracked; only the current data | All changes are tracked over time (Type 2) |
| **Surrogate Key** | Usually not used | Surrogate keys used for historical records (Type 2) |
| **Purpose of Key** | Identifies unique records for transactions | Identifies records for historical tracking and efficient querying |
| **Type of Key** | Primary key, often business-related (e.g., CustomerID) | Surrogate key, artificial (e.g., CustomerKey) |
| **Use of Business Keys** | Business keys (e.g., CustomerID) are used directly | Surrogate keys are used for linking to fact tables |
| **Key Change** | Business keys remain stable (e.g., CustomerID is fixed) | Surrogate keys remain fixed, even if business keys change |
| **Tracking Changes** | Not typically used for tracking changes in history | Allows tracking of historical changes (e.g., slowly changing dimensions) |
| **Example of Key** | CustomerID = 'C001' (Primary Key) | CustomerKey = 1 (Surrogate Key) |