# IF, ELSE, ELSE IF

**IF condition**

**BEGIN**

**-- Statements to be executed if the condition is true**

**END**

***Example:***

***1.***

DECLARE @Number INT = 3;

IF @Number > 5

BEGIN

PRINT 'The number is greater than 5.';

END

ELSE

BEGIN

PRINT 'The number is not greater than 5.';

END

***2.***

DECLARE @Number INT = 3;

IF @Number > 5

BEGIN

PRINT 'The number is greater than 5.';

END

ELSE

BEGIN

PRINT 'The number is not greater than 5.';

END

***3.***

DECLARE @Number INT = 3;

IF @Number > 5

BEGIN

PRINT 'The number is greater than 5.';

END

ELSE IF @Number = 5

BEGIN

PRINT 'The number is equal to 5.';

END

ELSE

BEGIN

PRINT 'The number is less than 5.';

END

***Point : how to check more than one condition ?***

***1.***

DECLARE @Number INT = 10;

IF @Number > 5 AND @Number < 20

BEGIN

PRINT 'The number is between 5 and 20.';

END

***2.***

DECLARE @Number INT = 3;

IF @Number < 5 OR @Number > 10

BEGIN

PRINT 'The number is less than 5 or greater than 10.';

END

***3.***

DECLARE @Number INT = 15;

IF (@Number < 5 OR @Number > 10) AND @Number <> 12

BEGIN

PRINT 'The number is either less than 5 or greater than 10, but not equal to 12.';

END

# BEGIN ... END

In T-SQL, the BEGIN and END keywords are used to define a block of statements.

***Example:***

***1.***

IF @Condition = 1

BEGIN

-- Statements to be executed if the condition is true

END

***2.***

WHILE @Counter <= 10

BEGIN

-- Statements to be executed in each iteration

SET @Counter = @Counter + 1;

END

***3.***

BEGIN

PRINT 'Statement 1';

PRINT 'Statement 2';

END

In this example, the BEGIN and END keywords are used to group two PRINT statements together. The block of statements will be executed sequentially when the script is run.

***Point : It's important to note that if there is only one statement in the block, the BEGIN and END keywords are optional.***

# PRINT

***Example:***

***1.***

PRINT 'Your message here';

***2.***

DECLARE @Message NVARCHAR(100) = 'Hello, world!';

PRINT @Message;

***3.***

DECLARE @Number INT = 42;

PRINT 'The value of @Number is: ' + CAST(@Number AS NVARCHAR(10));

Output = "The value of @Number is: 42."

***4.***

PRINT 'Statement 1';

PRINT 'Statement 2';

Output =

Statement 1;

Statement 2; --These are in two specific lines not at same line

***Point : When executing scripts in SQL Server Management Studio, the messages generated by the PRINT statements will be displayed in the "Messages" tab in the results pane.***

# CAST and CONVERT

**CAST (expression AS data\_type)**

**CONVERT(data\_type, expression [, style]) [**آنچه این بین است، یعنی نوشتنش اختیاری است**]**

Here, expression is the value you want to convert, and data\_type is the target data type to which you want to convert the expression.

***Example:***

***1.***

DECLARE @Number INT = 42;

DECLARE @NumberAsString NVARCHAR(50);

-- Convert the integer to a string

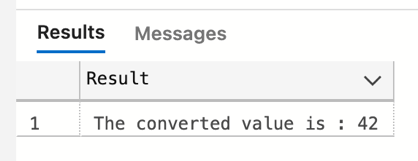
SET @NumberAsString = CAST(@Number AS NVARCHAR(50));

SELECT 'The converted value is: ' + @NumberAsString AS Result;

***2.***

DECLARE @Number INT = 42;

SELECT 'The converted value is: ' + CONVERT(VARCHAR, @Number) AS Result;

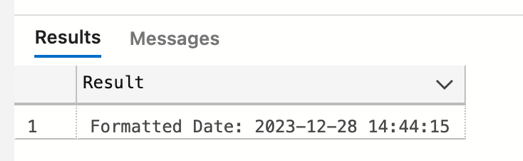


***Point : It's important to note that not all conversions are allowed, and you should use the appropriate conversion functions based on the data types involved.*** ***For example, if you're dealing with date and time values, you might use CONVERT or FORMAT functions instead of CAST for more flexible formatting and conversion options:***

DECLARE @CurrentDate DATETIME = GETDATE();

-- Convert the date to a string with a specific format

SELECT 'Formatted Date: ' + CONVERT(VARCHAR, @CurrentDate, 120) AS Result;

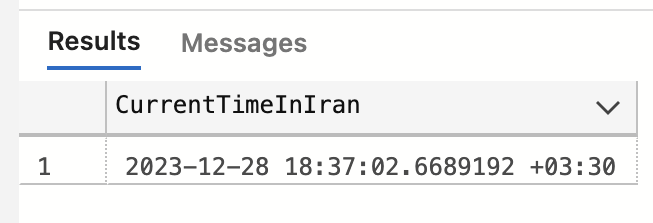
******

**In this case, CONVERT is used to format the @CurrentDate as a string using the 120 style code for the "YYYY-MM-DD HH:MI:SS" format.**

**In Microsoft SQL Server, you can get the current date and time in the local time zone of the server using the GETDATE() function. However, if you specifically need the current time in the Iran Standard Time (IRST) zone, you can use the AT TIME ZONE clause along with SYSDATETIMEOFFSET() or SYSUTCDATETIME().**

**Here's an example to get the current mean time in Iran:**

SELECT SYSDATETIMEOFFSET() AT TIME ZONE 'Iran Standard Time' AS CurrentTimeInIran;



### point : how to save days between two date variable in a int variable ?

DECLARE @StartDate DATETIME = '2023-01-01';

DECLARE @EndDate DATETIME = '2023-01-10';

DECLARE @DaysDifference INT;

-- Calculate the number of days between @StartDate and @EndDate

SET @DaysDifference = DATEDIFF(DAY, @StartDate, @EndDate);

-- Display the result

SELECT @DaysDifference AS DaysDifference;

# TRIGGER

a trigger is a set of instructions that automatically execute in response to an event on a particular table or view.

The basic syntax for creating a trigger is as follows:

**CREATE TRIGGER trigger\_name**

**ON** table\_name | view**\_name**

**[AFTER | INSTEAD OF] [INSERT | UPDATE | DELETE]**

**AS**

**BEGIN**

**-- Trigger logic here**

**END;**

## Special Tables:

These tables are automatically populated by SQL Server and are only available within the scope of the trigger. They are not explicitly created or managed by the user but are a feature provided by SQL Server.

### 1. INSERTED Table:

* The INSERTED table contains the newly inserted rows resulting from an INSERT or UPDATE operation.
* In the case of an INSERT trigger, the INSERTED table holds the newly inserted rows.
* In the case of an UPDATE trigger, the INSERTED table holds the updated values after the UPDATE operation.

### 2. DELETED Table:

* The DELETED table contains the rows that are being deleted or updated.
* In the case of a DELETE trigger, the DELETED table holds the rows that are being deleted.
* In the case of an UPDATE trigger, the DELETED table holds the values before the UPDATE operation.

Exampl***e:***

***1.***

***CREATE TRIGGER AuditTrailTrigger***

***ON YourTable***

***AFTER INSERT, UPDATE, DELETE***

***AS***

***BEGIN***

***INSERT INTO AuditTrail (TableName, Action, Timestamp)***

***SELECT 'YourTable',***

***CASE***

***WHEN EXISTS (SELECT \* FROM INSERTED) AND EXISTS (SELECT \* FROM DELETED) THEN 'UPDATE'***

***WHEN EXISTS (SELECT \* FROM INSERTED) THEN 'INSERT'***

***WHEN EXISTS (SELECT \* FROM DELETED) THEN 'DELETE'***

***END,***

***GETDATE();***

***END;***

***2.***

***CREATE TRIGGER ApplyDiscountTrigger***

***ON Orders***

***AFTER INSERT, UPDATE***

***AS***

***BEGIN***

***UPDATE Orders***

***SET Discount = 0.1 -- 10% discount***

***WHERE TotalAmount > 1000;***

***END;***

***3.***

***CREATE TRIGGER CascadeUpdateTrigger***

***ON Customers***

***AFTER UPDATE***

***AS***

***BEGIN***

***UPDATE Orders***

***SET CustomerName = i.CustomerName***

***FROM INSERTED i***

***WHERE Orders.CustomerID = i.CustomerID;***

***END;***

***4.***

***CREATE TRIGGER HistoryTrackingTrigger***

***ON YourTable***

***AFTER UPDATE***

***AS***

***BEGIN***

***INSERT INTO HistoryTable (ID, OldValue, NewValue, ModifiedDate)***

***SELECT i.ID, d.Column1, i.Column1, GETDATE()***

***FROM INSERTED i***

***JOIN DELETED d ON i.ID = d.ID***

***WHERE i.Column1 <> d.Column1;***

***END;***

### Point : How a trigger work ???

### In Microsoft SQL Server, the AFTER INSERT trigger fires after the INSERT operation has been completed but before the transaction is committed. The trigger is part of the same transaction as the statement that fired it. This means that the INSERT operation is done before the AFTER INSERT trigger logic is executed.

### Here's the sequence of events for an AFTER INSERT trigger:

### The INSERT statement is executed, and the new rows are added to the table.

### The AFTER INSERT trigger is invoked, and its logic is executed.

### If the trigger logic completes successfully and the overall transaction is committed, the INSERT operation becomes permanent.

### If there are any issues or errors within the trigger logic, it can potentially affect the success of the entire transaction. For example, if an error occurs in the trigger and the transaction is rolled back, the changes made by the INSERT operation and the trigger will be undone.

### Point : can a trigger execute after more than one event ???

In Microsoft SQL Server, a trigger can be associated with multiple events using a comma-separated list of events in the trigger definition. This allows a single trigger to respond to multiple types of events, such as INSERT, UPDATE, and DELETE.

Here's an example of how you can create a trigger that responds to more than one event:

### CREATE TRIGGER AfterMultipleEventsTrigger

### ON YourTable

### AFTER INSERT, UPDATE, DELETE

### AS

### BEGIN

### -- Trigger logic here

### -- This logic will be executed after INSERT, UPDATE, or DELETE operations on YourTable

### END;

In this example, the YourTrigger trigger will fire after an INSERT **or** after an UPDATE, **or** after a DELETE operation on the YourTable table.

### Point : assume that we want to add some rows by a query to a table which has a instesd of insert trigger . is it correct to say that this trigger runs for each row of insert query ?

Yes

### Point : is it correct to have a trigger with this condition , "after insert or instead of delete" ???

Yes, it is possible to have a trigger with the condition AFTER INSERT OR INSTEAD OF DELETE. This indicates that the trigger will be activated either after an INSERT operation or instead of a DELETE operation on the associated table.

Here's an example:

### CREATE TRIGGER YourTrigger

### ON YourTable

### AFTER INSERT, INSTEAD OF DELETE

### AS

### BEGIN

### -- Trigger logic here

### END;

***Point : what is " AS " (mssql format) ? to separate the header from its body. AS is necessary to write in create trigger an create procedure.***

***Point :*** An INSTEAD OF trigger is executed instead of the original event. **The original event is not done**; the INSTEAD OF trigger allows you to define custom logic to handle the event.

Point :To remove a trigger, use the DROP TRIGGER statement : DROP TRIGGER YourTrigger ON YourTable;

Point : Disabling/Enabling Triggers ???

You can temporarily disable a trigger using the DISABLE TRIGGER statement. Use the ENABLE TRIGGER statement to enable it again:

### DISABLE | ENABLE TRIGGER [ schema\_name . ] trigger\_name ON [ schema\_name . ] table\_name

### [ ,...n ] [ ; ]

### 1. DISABLE TRIGGER YourTrigger ON YourTable;

### 2. -- Disable the trigger DISABLE TRIGGER dbo.MyTrigger ON dbo.MyTable;

### 3. -- Enable the trigger ENABLE TRIGGER dbo.MyTrigger ON dbo.MyTable;

# EXISTS

In Microsoft SQL Server, the EXISTS is a Boolean operator that is used to determine whether a subquery returns any results. The EXISTS operator returns TRUE if the subquery returns one or more rows; otherwise, it returns FALSE.

-- Basic Syntax

IF/while/... EXISTS (subquery)

-- Statements to execute if the subquery returns one or more rows

Exampl***e:***

***1.***

IF EXISTS (

SELECT 1

FROM Orders o

WHERE o.CustomerID = 'ALFKI'

)

BEGIN

PRINT 'Orders exist for customer ALFKI.';

END

ELSE

BEGIN

PRINT 'No orders found for customer ALFKI.';

END;

#### *مقدار خاص انتخاب شده در پرس و جو برای هدف عملگر EXISTS اهمیتی ندارد. هدف اصلی این است که بررسی کنیم آیا هر ردیفی توسط subquery برگردانده شده است یا خیر.عملگر EXISTS به وجود ردیف ها علاقه مند است، نه داده های واقعی در آن ردیف ها. بنابراین، می‌توانید از هر مقدار یا عبارت ثابتی در دستور SELECT استفاده کنید، و همچنان هدف بررسی وجود ردیف‌ها را دارد.*

#### *SELECT 1 یک قرارداد رایج است که به عنوان نگهدارنده مکان استفاده می شود. همچنین می توانید از* SELECT 'X'، SELECT NULL *یا هر مقدار ثابت دیگری استفاده کنید. بخش مهم وجود عبارت* SELECT *در عبارت* EXISTS *است که نشان می دهد پرس و جو در حال بررسی وجود ردیف هایی است که شرایط مشخص شده را برآورده می کنند.*

### Point : this code is incorrect. Point : this code is correct.

### ~~IF (SELECT \* FROM mytable)~~ IF EXISTS (SELECT \* FROM mytable)

### ~~PRINT 'mymessage';~~ PRINT 'mymessage';

# DECLARE

## DECLARE @VariableName DataType [ = InitialValue ];

Exampl***e:***

***1.***

***DECLARE @Counter INT;***

***SET @Counter = 1;***

***2.***

***DECLARE @FirstName NVARCHAR(50) = 'John';***

***DECLARE @LastName NVARCHAR(50) = 'Doe';***

***DECLARE @Age INT = 30;***

***3.***

***DECLARE @ProductName NVARCHAR(100) = 'Widget';***

***SELECT ProductID, ProductName, Price***

***FROM Products***

***WHERE ProductName = @ProductName;***

***Point : how to save today's day in a varchar(2) by getdate() in sql ?***

### DECLARE @TodayDay VARCHAR(2);

### -- Set @TodayDay to today's day as a two-digit string

### SET @TodayDay = FORMAT(GETDATE(), 'dd');

### -- Display the result

### SELECT @TodayDay AS TodayDay;

### Point : what is @ ???

### In T-SQL (Transact-SQL), the @ symbol is used to declare and reference variables. When you see @ followed by a name (e.g., @VariableName) :

### Variable Declaration: --the @ symbol indicates that it is a variable

**DECLARE @MyVariable INT**;

1. Variable Assignment:

**SET @MyVariable = 42;**

1. Variable Usage:

**SELECT Column1, Column2**

**FROM MyTable**

**WHERE Column3 = @MyVariable;**

1. Stored Procedure Parameters: --@ is used to define input parameters

**CREATE PROCEDURE MyProcedure**

**@Parameter1 INT,**

**@Parameter2 NVARCHAR(50)**

**AS**

**BEGIN**

**-- Procedure logic using parameters**

**END;**

1. Output Parameters:

**CREATE PROCEDURE MyProcedure**

**@InputParameter INT,**

**@OutputParameter NVARCHAR(50) OUTPUT**

**AS**

**BEGIN**

**SET @OutputParameter = 'Output value';**

**END;**

# TRANSACTION

a transaction is a sequence of one or more SQL statements that are executed as a single unit of work. Transactions follow the principles of ACID properties: Atomicity (A), Consistency (C), Isolation (I), Durability (D)

Key Concepts and Commands Related to Transactions:

1. BEGIN TRANSACTION: Marks the beginning of a transaction. BEGIN TRANSACTION MyNamedTransaction;
2. COMMIT: Marks the successful end of a transaction, making all changes within the transaction permanent. COMMIT;
3. ROLLBACK: Undoes the changes made within a transaction, reverting the database to its state before the transaction started. ROLLBACK;
4. SAVEPOINT: Defines a point within a transaction to which you can later roll back. SAVE TRANSACTION SavepointName;
5. ROLLBACK TO SAVEPOINT: Rolls back a transaction to the specified savepoint. ROLLBACK TO SavepointName;
6. SET TRANSACTION ISOLATION LEVEL: Sets the isolation level for the duration of the transaction. SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

***Example:***

***1.*** ***Simple Money Transfer***

BEGIN TRANSACTION;

-- Deduct $100 from Account 1

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

-- Add $100 to Account 2

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

-- Commit the transaction if everything is successful

COMMIT;

***2.*** ***Money Transfer with Error Handling***

BEGIN TRANSACTION;

-- Deduct $100 from Account 1

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

-- Simulate an error (e.g., insufficient funds)

IF (SELECT Balance FROM Accounts WHERE AccountID = 1) < 0

BEGIN

-- Roll back the entire transaction if there are insufficient funds

ROLLBACK;

PRINT 'Transaction rolled back due to insufficient funds.';

END

ELSE

BEGIN

-- Add $100 to Account 2

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

-- If everything is successful, commit the transaction

COMMIT;

PRINT 'Transaction committed successfully.';

END;

***3.*** ***Transaction with Savepoint***

BEGIN TRANSACTION;

-- Deduct $50 from Account 1

UPDATE Accounts SET Balance = Balance - 50 WHERE AccountID = 1;

-- Save a savepoint

SAVE TRANSACTION Savepoint1;

-- Deduct $30 more from Account 1

UPDATE Accounts SET Balance = Balance - 30 WHERE AccountID = 1;

-- Check for an error condition (e.g., negative balance)

IF (SELECT Balance FROM Accounts WHERE AccountID = 1) < 0

BEGIN

-- Roll back to the savepoint

ROLLBACK TRANSACTION Savepoint1;

PRINT 'Transaction rolled back to savepoint due to negative balance.';

END

ELSE

BEGIN

-- If everything is successful, commit the transaction

COMMIT;

PRINT 'Transaction committed successfully.';

END;

***4.*** ***Simple Nested Transaction***

BEGIN TRANSACTION;

-- Outer transaction

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

-- Inner transaction (not truly nested)

BEGIN TRANSACTION;

-- Additional logic within the inner transaction

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

-- Commit the inner transaction (does not affect outer transaction)

COMMIT;

-- Commit the outer transaction

COMMIT;

# SET

## the SET statement is used to assign values to variables and to configure various session settings. Here are the primary uses of the SET statement:

1. Assigning Values to Variables:

### DECLARE @VariableName INT;

### SET @VariableName = 42;

### -- or directly in one line

### DECLARE @VariableName INT = 42;

2. Session Settings:

### SET ANSI\_NULLS ON; -- Enables ANSI NULL behavior

### SET ANSI\_NULLS OFF; -- Disables ANSI NULL behavior

### SET NOCOUNT ON; -- Suppresses the "xx rows affected" message

### SET NOCOUNT OFF; -- Displays the "xx rows affected" message

3. Options and Behavior:

### SET CONCAT\_NULL\_YIELDS\_NULL ON; -- Concatenating NULL with a string results in NULL

### SET CONCAT\_NULL\_YIELDS\_NULL OFF; -- Concatenating NULL with a string results in the string

### SET ARITHABORT ON; -- Arithmetic abort behavior

### SET ARITHABORT OFF; -- Arithmetic abort behavior

4. Setting Date and Time Formats:

### SET DATEFORMAT dmy; -- Sets the date format to day/month/year

### SET DATEFORMAT mdy; -- Sets the date format to month/day/year

### SET LANGUAGE 'German'; -- Sets the language for date and time formats

Exampl***e:***

***1.***

***DECLARE @Counter INT = 1;***

***SET NOCOUNT ON;***

***WHILE @Counter <= 5***

***BEGIN***

***PRINT 'Iteration ' + CAST(@Counter AS VARCHAR(2));***

***SET @Counter = @Counter + 1;***

***END;***

***SET NOCOUNT OFF;***

In this example, SET NOCOUNT ON is used to suppress the "xx rows affected" message for each iteration of the loop, and SET NOCOUNT OFF is used to revert to the default behavior after the loop.

### Point : how to use sum in sql server and put it in a variable ?

### DECLARE @totalSales INT;

### SELECT @totalSales = AVG(SalesAmount)

### FROM Sales;

# PROCEDURE

#### ***رویه ذخیره شده مجموعه ای از پیش کامپایل شده از یک یا چند دستور Transact-SQL است که به عنوان یک شی واحد در پایگاه داده ذخیره می شوند. رویه های ذخیره شده به شما این امکان را می دهند که دنباله ای از عبارات SQL را در یک واحد قابل استفاده مجدد و قابل اجرا کپسوله کنید و راهی برای سازماندهی و مدولار کردن منطق پایگاه داده شما فراهم می کند. رویه ها و توابع ذخیره شده در SQL Server هر دو شیء پایگاه داده هستند که مجموعه ای از دستورات T-SQL را محصور می کنند. در حالی که آنها شباهت هایی دارند، تفاوت های کلیدی بین رویه ها و توابع ذخیره شده وجود دارد.***

Exampl***e:***

### 1.

### CREATE PROCEDURE GetEmployeeCount

### AS

### BEGIN

### SELECT COUNT(\*) AS TotalEmployees

### FROM Employees;

### END;

### 2.

### CREATE PROCEDURE GetEmployeesByDepartment

### @departmentID INT

### AS

### BEGIN

### SELECT EmployeeID, FirstName, LastName

### FROM Employees

### WHERE DepartmentID = @departmentID;

### END;

### 3.

### CREATE PROCEDURE GetEmployeeFullName

### @employeeID INT,

### @fullName NVARCHAR(100) OUTPUT

### AS

### BEGIN

### SELECT @fullName = FirstName + ' ' + LastName

### FROM Employees

### WHERE EmployeeID = @employeeID;

### END;

### 4.

### CREATE PROCEDURE CheckEmployeeExistence

### @employeeID INT

### AS

### BEGIN

### IF EXISTS (SELECT 1 FROM Employees WHERE EmployeeID = @employeeID)

### RETURN 1; -- Employee exists

### ELSE

### RETURN 0; -- Employee does not exist

### END;

### 5.

### CREATE PROCEDURE ExampleProcedure

### @inputParameter1 INT,

### @inputParameter2 VARCHAR(50),

### @outputParameter1 INT OUTPUT,

### @outputParameter2 NVARCHAR(100) OUTPUT

### AS

### BEGIN

### -- Perform some logic using input parameters

### SET @outputParameter1 = @inputParameter1 \* 2;

### SET @outputParameter2 = 'Hello, ' + @inputParameter2;

### END;

***6.***

***-- Outer stored procedure***

***CREATE PROCEDURE OuterProcedure***

***@outerParam INT***

***AS***

***BEGIN***

***PRINT 'Outer Procedure Start';***

***-- Call the nested stored procedure***

***EXEC InnerProcedure @outerParam;***

***PRINT 'Outer Procedure End';***

***END;***

***-- Inner stored procedure***

***CREATE PROCEDURE InnerProcedure***

***@innerParam INT***

***AS***

***BEGIN***

***PRINT 'Inner Procedure Start';***

***PRINT 'Inner Parameter: ' + CAST(@innerParam AS VARCHAR(10));***

***PRINT 'Inner Procedure End';***

***END;***

***When you execute OuterProcedure, it will, in turn, execute InnerProcedure with the specified parameter:***

***EXEC OuterProcedure @outerParam = 42;***

***The result will be a series of print statements indicating the flow of execution:***

***Outer Procedure Start***

***Inner Procedure Start***

***Inner Parameter: 42***

***Inner Procedure End***

***Outer Procedure End***

***Point : If you use the RETURN statement in a stored procedure, it can only return a single integer value. This is a limitation of the RETURN statement. If you need to return multiple values or a result set, you should use other methods, such as output parameters or result sets.***

***Point : To execute a stored procedure, you can use the EXEC keyword or EXECUTE***

***-- Execute the stored procedure***

***1.***

***EXEC GetProductsByCategory @categoryID = 1;***

***2.***

***DECLARE @result1 INT, @result2 NVARCHAR(100);***

***EXEC ExampleProcedure***

***@inputParameter1 = 5,***

***@inputParameter2 = 'World',***

***@outputParameter1 = @result1 OUTPUT,***

***@outputParameter2 = @result2 OUTPUT;***

***-- Display the results***

***SELECT @result1 AS OutputParameter1, @result2 AS OutputParameter2;***

***point : can we write this in above example => @output1 = @outputParameter1 OUTPUT ???***

***No, the order of the assignment should be @outputParameter1 OUTPUT = @output1. The syntax for output parameters in a stored procedure call requires the output parameters to be specified before the variables that will capture their values.***

*Point : write a PROCEDURE with 3 input and 2 output ?*

*) نیازی به* declare *نوشتن برای تعریف ورودی ها نیست(*

*CREATE PROCEDURE CalculateProduct*

*@Input1 INT,*

*@Input2 INT,*

*@Input3 INT,*

*@OutputSum INT OUTPUT,*

*@OutputProduct INT OUTPUT*

*AS*

*BEGIN*

*-- Calculate the sum of the inputs*

*SET @OutputSum = @Input1 + @Input2 + @Input3;*

*-- Calculate the product of the inputs*

*SET @OutputProduct = @Input1 \* @Input2 \* @Input3;*

*END;*

*Point : To create a stored procedure with default parameter values, you can specify default values for the parameters in the procedure definition. Here's an example:*

*CREATE PROCEDURE ExampleProcedure*

*@param1 INT = 10,*

*@param2 VARCHAR(50) = 'DefaultString'*

*AS*

*BEGIN*

*-- Procedure logic using @param1 and @param2*

*PRINT 'Parameter 1: ' + CAST(@param1 AS VARCHAR(10));*

*PRINT 'Parameter 2: ' + @param2;*

*END;*

*both two below commands are true:*

*1. EXEC ExampleProcedure; -- Uses default values for @param1 and @param2*

*2. EXEC ExampleProcedure @param1 = 20, @param2 = 'NewString';*

*Point : how to delete a PROCEDURE ???* ***DROP PROCEDURE procedure\_name***

*Point : برای تغییر یک رویه ذخیر شده*

*هنگام استفاده از ALTER PROCEDURE، معمولاً نیازی به بازنویسی کل رویه ذخیره شده ندارید. هدف ALTER PROCEDURE این است که بدون نیاز به تعریف مجدد کامل، تغییرات یا اضافاتی در رویه ذخیره شده موجود ایجاد کند. شما بخش های اصلاح شده یا اضافی رویه ذخیره شده را ارائه می دهید و SQL Server این تغییرات را در تعریف موجود گنجانده است.*

ALTER PROCEDURE procedure\_name

AS

BEGIN

sql\_statement

END;

# CASE

#### عبارت CASE برای منطق شرطی در یک پرس و جو یا یک عبارت Transact-SQL استفاده می شود. به شما امکان می دهد ارزیابی های شرطی را انجام دهید و مقادیر مختلف را بر اساس شرایط مشخص شده برگردانید.

## CASE

## WHEN condition1 THEN result1

## WHEN condition2 THEN result2

## ...

## WHEN conditionN THEN resultN

## ELSE elseResult

## END

Exampl***e:***

***1.***

***SELECT***

***ProductName,***

***UnitPrice,***

***CASE***

***WHEN UnitPrice < 50 THEN 'Low Price'***

***WHEN UnitPrice >= 50 AND UnitPrice < 100 THEN 'Medium Price'***

***ELSE 'High Price'***

***END AS PriceCategory***

***FROM Products;***

#### در این مثال:

#### برای هر ردیف در جدول Products، عبارت CASE UnitPrice را ارزیابی می کند و یک PriceCategory را بر اساس شرایط مشخص شده اختصاص می دهد.

#### اگر UnitPrice کمتر از 50 باشد، دسته "قیمت پایین" است.

#### اگر UnitPrice بین 50 (شامل) و 100 (انحصاری) باشد، دسته "قیمت متوسط" است.

#### در غیر این صورت، دسته "قیمت بالا" است.

### 2.

### SELECT

### OrderID,

### OrderDate,

### CASE

### WHEN ShipCountry = 'USA' THEN 'Domestic'

### ELSE 'International'

### END AS OrderType

### FROM Orders

### WHERE

### CASE

### WHEN ShipCountry = 'USA' THEN 1

### ELSE 0

### END = 1;

Here, the CASE statement is used in both the SELECT clause and the WHERE clause. It categorizes orders as 'Domestic' or 'International' in the SELECT clause and filters only domestic orders in the WHERE clause.

### 3.

### SELECT

### CategoryID,

### AVG(UnitPrice) AS AveragePrice,

### CASE

### WHEN AVG(UnitPrice) < 50 THEN 'Low Average Price'

### WHEN AVG(UnitPrice) >= 50 AND AVG(UnitPrice) < 100 THEN 'Medium Average Price'

### ELSE 'High Average Price'

### END AS PriceCategory

### FROM Products

### GROUP BY CategoryID;

### 4.

### SELECT

### Grade,

### CASE

### WHEN Score >= 90 THEN 'A'

### WHEN Score >= 80 THEN 'B'

### WHEN Score >= 70 THEN 'C'

### WHEN Score >= 60 THEN 'D'

### ELSE 'F'

### END AS GradeCategory

### FROM Students;

***Point :*** دو سلکت زیر باهم برابر هستند

SELECT

OrderID,

OrderDate,

CASE

WHEN ShipCountry = 'USA' THEN 'Domestic' SELECT

ELSE 'International' OrderID,

END AS OrderType OrderDate,

FROM Orders CASE

WHERE WHEN ShipCountry = 'USA' THEN 'Domestic'

CASE ELSE 'International'

WHEN ShipCountry = 'USA' THEN 1 END AS OrderType

ELSE 0 FROM Orders

END = 1; WHERE ShipCountry = 'USA';

***Point :***

END: Marks the end of the CASE expression.

AS AliasName: Assigns an alias to the result of the entire CASE expression. This alias can be used to reference the result in the output.

#### *کلمه کلیدی AS، در این زمینه، برای خوانایی و برای ارائه یک نام معنادار (نام مستعار) برای نتیجه عبارت CASE استفاده می‌شود و در خروجی واضح‌تر می‌شود که هدف آن ستون محاسبه‌شده چیست.*

point : چگونه به یک ستون نامی مستعار بر اساس مطابق بودن مقدار آن ستون با شرطی نسبت داد ؟

1.

SELECT

ProductName,

UnitPrice,

CASE

WHEN UnitPrice < 50 THEN 'Low Price'

WHEN UnitPrice >= 50 AND UnitPrice < 100 THEN 'Medium Price'

ELSE 'High Price'

END AS PriceCategory

FROM Products;

2.

CREATE FUNCTION dbo.GetPriceCategory(@UnitPrice DECIMAL(18, 2)) RETURNS NVARCHAR(50)

AS

BEGIN

DECLARE @Category NVARCHAR(50)

IF @UnitPrice < 50

SET @Category = 'Low Price'

ELSE IF @UnitPrice >= 50 AND @UnitPrice < 100

SET @Category = 'Medium Price'

ELSE

SET @Category = 'High Price'

RETURN @Category

END;

SELECT

ProductName,

UnitPrice,

dbo.GetPriceCategory(UnitPrice) AS PriceCategory

FROM Products;

point : In SQL, **the WHEN and THEN keywords are specifically associated with the CASE statement**. They are used together to define conditions and the corresponding results in a CASE expression.

# WHILE

## The WHILE loop continues to execute a block of SQL statements as long as the specified condition is true.

Exampl***e:***

***1.***

***DECLARE @Counter INT = 1;***

***WHILE @Counter <= 10***

***BEGIN***

***-- Your statements here***

***PRINT 'Iteration ' + CAST(@Counter AS VARCHAR(2));***

***SET @Counter = @Counter + 1;***

***END;***

***Point : BREAK and CONTINUE***

* ***BREAK: Exits the loop prematurely.***
* ***CONTINUE: Skips the rest of the code in the loop and goes to the next iteration.***

DECLARE @Counter INT = 1;

WHILE @Counter <= 10

BEGIN

IF @Counter = 5

BREAK; -- Exit the loop when Counter is 5

-- Your statements here

PRINT 'Iteration ' + CAST(@Counter AS VARCHAR(2));

SET @Counter = @Counter + 1;

END;

**تنها حلقه ای که sqlاز آن پشتیبانی می‌کند، whileاست *Point :***

# Dynamic SQL

## Building queries based on user input or dynamic conditions (at rumtime) ...

In SQL Server, EXEC and sp\_executesql are both used for executing dynamic SQL, but there are some differences between them. In general, when working with dynamic SQL, it's recommended to use sp\_executesql due to its support for parameters, better performance optimization, and enhanced security features.

**EXEC**: The EXEC statement is used for executing dynamic SQL without supporting parameters directly. If you want to include parameters in your dynamic SQL, you need to concatenate them into the SQL string.

Example using EXEC without parameters:

**DECLARE @sql NVARCHAR(MAX);**

**SET @sql = 'SELECT \* FROM MyTable WHERE Column1 = ''someValue''';**

**EXEC(@sql);**

**sp\_executesql**: sp\_executesql is a system stored procedure that supports parameterized dynamic SQL. It allows you to define and pass parameters to the dynamic SQL, providing a safer and more efficient way to handle dynamic queries.

Exampl***e:***

### 1.

### DECLARE @sql NVARCHAR(MAX);

### DECLARE @param NVARCHAR(50);

### SET @sql = 'SELECT \* FROM MyTable WHERE Column1 = @param';

### EXEC sp\_executesql @sql, N'@param NVARCHAR(50)', @param = 'someValue';

### 2.

### DECLARE @sql NVARCHAR(MAX);

### DECLARE @condition NVARCHAR(50) = 'someValue';

### SET @sql = 'SELECT \* FROM MyTable';

### IF @condition IS NOT NULL

### SET @sql = @sql + ' WHERE Column1 = @condition';

### -- Execute the dynamically constructed SQL statement

### EXEC sp\_executesql @sql, N'@condition NVARCHAR(50)', @condition = @condition;

point :

#### دستور سلکت جدولی با ستون هایی با نامی برابر نام ستون انتخابی از جدول یا نامی مستعار که با as اختصاص می‌دهیم، درست می‌کند که در آن جدول، پاسخ کوئری هایی که در آن سلکت صدق می‌کنند را قرار می‌دهد.

# IN , IS NULL , NOT IN , IS NOT NULL

## The IN operator in SQL is used to specify a range for a WHERE clause. It is used to reduce the need for multiple OR conditions in a SELECT, INSERT, UPDATE, or DELETE statement. Here's the basic syntax of the IN operator:

## SELECT column\_name(s)

## FROM table\_name

## WHERE column\_name IN (value1, value2, ...);

Exampl***e:***

**1.**

-- Select rows where the 'City' column is either 'New York' or 'London'

SELECT \*

FROM Customers

WHERE City IN ('New York'**,** 'London'); // city = 'New York' **OR** city = 'London'

**2.**

-- Update the 'Status' column to 'Approved' for orders with specific IDs

UPDATE Orders

SET Status = 'Approved'

WHERE OrderID IN (1001**,** 1002**,** 1003); // OrderID = 1001 **OR** OrderID = 1002 **OR** OrderID = 1003

### point : is it correct => " IS NULL " equal with " = " ???

#### هنگام برخورد با مقادیر NULL در SQL، نمی توانید از عملگر برابری (=) استفاده کنید زیرا NULL به عنوان یک مقدار ناشناخته در نظر گرفته می شود و نتیجه هر مقایسه برابری با NULL همیشه ناشناخته است. بنابراین، باید از "IS NULL" یا "IS NOT NULL" استفاده کنید تا به طور خاص مقادیر NULL را بررسی کنید.

#### ترتیب پردازش منطقی دستور SELECT به شرح زیر است point :

1. **FROM**: This is the first clause to be evaluated. It specifies the source tables or views from which the data will be retrieved.
2. **WHERE**: This is the second clause to be evaluated. It filters rows based on the specified conditions.
3. **GROUP BY**: If present, the data is then grouped based on the specified columns.
4. **HAVING**: This is similar to WHERE but is used with GROUP BY. It filters groups based on the specified conditions.
5. **SELECT**: This is the fifth clause to be evaluated. It specifies the columns to be retrieved.
6. **ORDER BY**: This is the last clause to be evaluated. It sorts the result set based on the specified columns.Top of Form

Bottom of Form

It's important to note that SQL Server has a query optimizer that attempts to find the most efficient execution plan for a query, and it may choose different strategies based on the specifics of your database and the query being executed. You can view the execution plan for a query using tools like SQL Server Management Studio (SSMS) to see how SQL Server plans to execute the query.

### point :

### -- Select customers who have placed orders

### SELECT \*

### FROM Customers

### WHERE CustomerID IN (SELECT DISTINCT CustomerID FROM Orders);

&

### -- Select customers who have placed orders using EXISTS

### SELECT \*

### FROM Customers

### WHERE EXISTS (SELECT 1 FROM Orders WHERE Orders.CustomerID = Customers.CustomerID);

you can generally think of the subquery in the IN clause or the EXISTS clause as being executed before the outer query. The subquery is evaluated, and its result is used in the evaluation of the outer query.

#### *هر دو IN و EXISTS برای اهداف مشابهی استفاده می شوند، اما معنایی متفاوتی دارند. IN بررسی می کند که آیا یک مقدار با هر مقدار در نتیجه پرس و جو مطابقت دارد یا خیر، در حالی که EXISTS بررسی می کند که آیا درخواست فرعی هر ردیفی را برمی گرداند یا خیر. بسته به نیازهای خاص خود ، ممکن است تصمیم بگیرید که از یکی یا دیگری استفاده کنید .نکته بالا را چک کن (It 's important...)*

# SUBQUERY

## یک پرس و جو فرعی که به عنوان پرس و جوی تودرتو یا پرس و جو داخلی نیز شناخته می شود، پرس و جوی است که در پرس و جو دیگری تعبیه شده است. برای بازیابی داده هایی استفاده می شود که در پرس و جو اصلی به عنوان شرطی برای محدود کردن بیشتر داده هایی که باید بازیابی شوند استفاده می شود. به عبارت دیگر، یک پرس و جو فرعی یک پرس و جو در داخل پرس و جوی دیگر است.

#### پرسش‌های فرعی را می‌توان در بخش‌های مختلف یک دستور SQL، از جمله عبارت SELECT، بند FROM، عبارت WHERE و موارد دیگر، بسته به نیاز خاص، استفاده کرد. انواع اصلی سوالات فرعی عبارتند از:

**1. Scalar Subquery: Returns a single value**

A scalar subquery is a subquery that returns a single value, and it is typically used in places where a single value is expected, such as in the SELECT clause or the WHERE clause.

*SELECT CustomerName, (SELECT MAX(OrderDate) FROM Orders WHERE Customers.CustomerID = Orders.CustomerID) AS LastOrderDate*

*FROM Customers;*

In this example, the scalar subquery (SELECT MAX(OrderDate) FROM Orders WHERE Customers.CustomerID = Orders.CustomerID) returns the maximum order date for each customer.

**2. Row Subquery: Returns a single row**

### A row subquery is a subquery that returns a single row. It is often used in contexts where a single row of data is expected, such as in comparison.

*SELECT CustomerName*

*FROM Customers*

*WHERE (CustomerID, City) = (SELECT CustomerID, City FROM Orders WHERE OrderID = 123);*

In this example, the row subquery (SELECT CustomerID, City FROM Orders WHERE OrderID = 123) returns a single row with the customer ID and city associated with the order ID 123.

**3. Table Subquery: Returns a table**

A table subquery is a subquery that returns a table. It can be used in places where a table is expected, such as in the FROM clause.

*SELECT \**

*FROM (SELECT CustomerID, COUNT(\*) AS OrderCount FROM Orders GROUP BY CustomerID) AS OrderSummary;*

In this example, the table subquery (SELECT CustomerID, COUNT(\*) AS OrderCount FROM Orders GROUP BY CustomerID) returns a table with the count of orders for each customer. The outer query then selects all columns from this derived table.

Exampl***e:***

**1.** **Correlated Subquery**

Find products with a price higher than the average price in their category.

SELECT ProductName, CategoryID, UnitPrice

FROM Products p

WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products WHERE CategoryID = p.CategoryID);

**2. Exists Subquery**

Retrieve orders that have at least one associated order detail.

SELECT \*

FROM Orders o

WHERE EXISTS (SELECT 1 FROM OrderDetails WHERE OrderID = o.OrderID);

**3.** **Subquery in the FROM Clause (Derived Table)**

Calculate the total sales for each customer.

SELECT CustomerID, CustomerName, TotalSales

FROM Customers c

JOIN (SELECT CustomerID, SUM(UnitPrice \* Quantity) AS TotalSales FROM Orders GROUP BY CustomerID) o

ON c.CustomerID = o.CustomerID;

# SELECT TOP

## SELECT TOP expression|PERCENT [WITH TIES]

## column1, column2, ...

## FROM

## table

## ORDER BY

## column1 [ASC | DESC], column2 [ASC | DESC], ...;

Exampl***e:***

1.

-- Retrieve the top 5 students with their scores, including ties

SELECT TOP (5) WITH TIES

StudentID, Score

FROM

Students

ORDER BY

Score DESC;

In this example, if there are students with the same highest score, the WITH TIES option ensures that all those students are included in the result set.

point : The SELECT TOP WITH TIES clause is used to retrieve additional rows with the same sort key values **as the last row** in the specified result set

point : what is DESC and ASC ???

In SQL, DESC and ASC are keywords used with the ORDER BY clause to specify the sort order of the result set. the default is ASC.

**ASC (Ascending)** => Rows are sorted from the lowest value to the highest value.

**DESC (Descending)** => It sorts rows in descending order, from the highest value to the lowest value.

# OFFSET , FETCH

## [ORDER BY column\_list [ASC | DESC]]

## OFFSET offset\_row\_count {ROW | ROWS}

## FETCH {FIRST | NEXT} fetch\_row\_count {ROW | ROWS} [ONLY]

#### *کلمه کلیدی ONLY اختیاری است و می توانید آن را بدون تأثیر بر عملکرد پرس و جو حذف کنید*point : *.*

Exampl***e:***

1. رد کردن 10 دانشجوی اول رشته کامپیوتر و برگرداندن اطلاعات سایر دانشجویان این رشته

SELECT \* FROM STT

WHERE STMJR = 'COMPUTER'

ORDER BY STID

OFFSET 10 ROWS

#### رد کردن ۱۰دانشجوی اول رشته کامپیوتر و برگرداندن اطلاعات 10دانشجوی بعدی این رشته2.

SELECT \* FROM STT

WHERE STMJR = 'COMPUTER'

ORDER BY STID

OFFSET 10 ROWS

FETCH NEXT 10 ROWS ONLY;

3. برگرداندن ۱۰ نمره اول در درس ساختمان داده در نیم سال اول ۹۸-۹۹

SELECT STCOT. GRADE

FROM COT JOIN STCOT ON COT.COID = STCOT. COID

WHERE COT. COTITLE = 'DATA STRUCTURES' AND

TR = 2 AND

YRYR = '98-99'

ORDER BY STCOT GRADE DESC

[OFFSET 0 ROWS] // if you don't specify OFFSET, it is implicitly treated as OFFSET 0 ROWS

FETCH FIRST 10 ROWS ONLY;

# GROUPING SETS , CUBE , ROLLUP

#### بند GROUPING SETS برای تعیین مجموعه های گروه بندی متعدد در یک پرس و جو استفاده می شود. این به شما امکان می‌دهد تا sub totals و grand totals را در ابعاد چندگانه به روشی انعطاف‌پذیر و مختصر ایجاد کنید. GROUPING SETS مشابه عملگرهای CUBE و ROLLUP است، اما کنترل بیشتری بر مجموعه‌های گروه‌بندی که می‌خواهید در مجموعه نتایج بگنجانید، فراهم می‌کند.

### In the GROUP BY GROUPING SETS, you specify a list of grouping sets, where each grouping set represents a combination of columns for which you want to generate subtotals or grand totals.

### Example:

SELECT Region, Product, SUM(Sales) AS TotalSales

FROM SalesData

( (Region, Product), (Region), () );

In this example, the GROUP BY GROUPING SETS clause is used to calculate subtotals for each combination of Region and Product, subtotals for each Region, and a grand total for all combinations (represented by the empty set of parentheses).

You can include multiple grouping sets within the GROUPING SETS clause, each enclosed in parentheses. The empty parentheses () represent the grand total. The result set might look something like this:

Region | Product | TotalSales

-------|---------|-----------

East | A | 1000

East | B | 1500

West | A | 1200

West | B | 800

East | | 2500 -- Subtotal for East

West | | 2000 -- Subtotal for West

| | 4500 -- Grand Total

In this result set, the row with empty values for Region and Product represents the grand total for all combinations.

point : what is the meaning that when we write more than one column in GROUY BY ??? or in ORDER BY ???

#### هنگامی که بیش از یک ستون در عبارت GROUP BY می نویسید، به پایگاه داده دستور می دهید تا ردیف ها را بر اساس ترکیبات منحصر به فرد ستون های مشخص شده گروه بندی کند. این بدان معنی است که مجموعه نتایج شامل یک ردیف برای هر ترکیب منحصر به فرد مقادیر در ستون های فهرست شده در بند GROUP BY خواهد بود.

SELECT Region, Product, SUM(Sales) AS TotalSales

FROM SalesData

GROUP BY Region, Product;

The result set will contain rows for each unique combination of Region and Product, and the TotalSales column will represent the sum of sales for each combination.

| Region | Product | TotalSales |

|----------|----------|------------|

| North | Widget A | 1000 |

| North | Widget B | 1200 |

| South | Widget A | 800 |

| South | Widget B | 900 |

| East | Widget A | 1500 |

| East | Widget B | 1100 |

| West | Widget A | 1300 |

| West | Widget B | 1000 |

point : what is the meaning that when we write more than one column in ORDER BY ???

#### *وقتی بیش از یک ستون در عبارت ORDER BY می نویسید، سطرها ابتدا بر اساس ستون اول فهرست شده در عبارت ORDER BY مرتب می شوند و برای ردیف هایی با مقادیر یکسان در ستون اول، مرتب سازی بر اساس ستون دوم و غیره اعمال می شود.*

SELECT FirstName, LastName, BirthDate

FROM Employees

ORDER BY LastName ASC, FirstName ASC;

In this example, the result set is first sorted in ascending order based on the LastName column. For rows with the same last name, the sorting is then applied based on the FirstName column in ascending order.

#### *عملگر* CUBE *مجموعه ای از نتایج را تولید می کند که شامل تمام ترکیب های ممکن از ستون های مشخص شده در بند GROUP BY است*point :

**1.**

**SELECT Region, Product, SUM(Sales) AS TotalSales**

**FROM SalesData**

**GROUP BY CUBE (Region, Product);**

Region | Product | TotalSales

-------|---------|-----------

East | A | 1000

East | B | 1500

West | A | 1200

West | B | 800

East | | 2500 -- Subtotal for East

West | | 2000 -- Subtotal for West

| A | 2200 -- Subtotal for Product A

| B | 2300 -- Subtotal for Product B

| | 4500 -- Grand Total

2.

**SELECT Region, Product, SUM(Sales) AS TotalSales**

**FROM SalesData**

**GROUP BY Country, CUBE (Region, Product);**

Assuming you have the following data:

Country | Region | Product | Sales

--------|--------|---------|------

USA | East | A | 1000

USA | East | B | 1500

USA | West | A | 1200

USA | West | B | 800

Canada | East | A | 500

Canada | East | B | 700

Canada | West | A | 800

Canada | West | B | 600

The query result might look something like this:

Country | Region | Product | TotalSales

--------|--------|---------|-----------

USA | East | A | 1000

USA | East | B | 1500

USA | West | A | 1200

USA | West | B | 800

USA | | | 4500 -- Grand Total for USA

Canada | East | A | 500

Canada | East | B | 700

Canada | West | A | 800

Canada | West | B | 600

Canada | | | 2600 -- Grand Total for Canada

| East | A | 1500 -- Subtotal for East

| East | B | 2200 -- Subtotal for East

| West | A | 2000 -- Subtotal for West

| West | B | 1400 -- Subtotal for West

| | | 7100 -- Grand Total

### point :

#### *در GROUP BY ROLLUP، ستون هایی را مشخص می کنید که می خواهید برای آنها جمع های فرعی در یک سلسله مراتب ایجاد کنید. مجموعه نتایج شامل ترکیبی از مقادیر از ستون‌های مشخص شده است که جمع‌های فرعی را در امتداد سلسله مراتب مشخص شده تولید می‌کند.*

For example, consider a dataset with columns for Country, Region, and Product. The hierarchy might be structured as follows:

#### هنگام بحث در مورد سلسله مراتب ستون ها در زمینه ROLLUP یا سایر عملگرهای گروه بندی، معمولاً به معنای مرتب کردن ستون ها به گونه ای است که یک نظم یا رابطه طبیعی را نشان می دهد.

1. **Country Level:** Subtotals and grand totals are calculated for each country.
2. **Region Level:** Subtotals and grand totals are calculated for each region within each country.
3. **Product Level:** Subtotals and grand totals are calculated for each product within each region and country.

1.

**SELECT Region, Product, SUM(Sales) AS TotalSales**

**FROM SalesData**

**GROUP BY ROLLUP (Region, Product);**

#### *در این مثال، پرس و جو برای هر دو ستون "Region" و "Product" جمع های فرعی تولید می کند. مجموعه نتایج شامل ترکیب‌های مختلفی از این ابعاد است که جمع‌های فرعی را در امتداد یک سلسله مراتب ارائه می‌کند.*

Region | Product | TotalSales

-------|---------|-----------

East | A | 1000

East | B | 1500

East | | 2500 -- Subtotal for East

West | A | 1200

West | B | 800

West | | 2000 -- Subtotal for West

| | 4500 -- Grand Total

2. Let's assume the following data:

Country | Region | Product | Sales

--------|--------|---------|------

USA | East | A | 1000

USA | East | B | 1500

USA | West | A | 1200

USA | West | B | 800

Canada | East | A | 500

Canada | East | B | 700

Canada | West | A | 800

Canada | West | B | 600

**SELECT Country, Region, Product, SUM(Sales) AS TotalSales**

**FROM SalesData**

**GROUP BY ROLLUP (Country, Region, Product);**

The result set might look like this:

Country | Region | Product | TotalSales

--------|--------|---------|-----------

USA | East | A | 1000

USA | East | B | 1500

USA | East | | 2500 -- Subtotal for USA, East

USA | West | A | 1200

USA | West | B | 800

USA | West | | 2000 -- Subtotal for USA, West

USA | | | 4500 -- Grand Total for USA

Canada | East | A | 500

Canada | East | B | 700

Canada | East | | 1200 -- Subtotal for Canada, East

Canada | West | A | 800

Canada | West | B | 600

Canada | West | | 1400 -- Subtotal for Canada, West

Canada | | | 2600 -- Grand Total for Canada

| | A | 1800 -- Subtotal for Product A

| | B | 2300 -- Subtotal for Product B

| | | 4100 -- Grand Total

# MERGE

#### ***به ویژه زمانی مفید است که می خواهید داده ها را بین دو جدول همگام کنید یا داده ها را از یک جدول منبع در جدول هدف ادغام کنید.***

MERGE target\_table AS target USING source\_table AS source

ON <merge\_condition>

WHEN MATCHED THEN ...

WHEN NOT MATCHED [BY TARGET] THEN ...

WHEN NOT MATCHED BY SOURCE THEN ...

### Example:

**MERGE Products AS target**

**USING ProductsToUpdate AS source**

**ON target.ProductID = source.ProductID**

**WHEN MATCHED THEN**

**UPDATE SET target.ProductName = source.ProductName, target.Price = source.Price**

**WHEN NOT MATCHED THEN**

**INSERT (ProductID, ProductName, Price) VALUES (source.ProductID, source.ProductName, source.Price);**

In this example:

* Products is the target table.
* ProductsToUpdate is the source table.
* The condition for matching rows is based on the ProductID.
* When a match is found, it updates the ProductName and Price in the target.
* When no match is found, it inserts a new row into the target.

# CTE

#### ***یک عبارت جدول مشترک (CTE) یک مجموعه نتایج موقت است که می توانید در یک عبارت SELECT، INSERT، UPDATE یا DELETE به آن ارجاع دهید. با استفاده از کلمه کلیدی WITH تعریف می شود و می توان آن را به عنوان یک مجموعه نتایج موقت نامگذاری شده در نظر گرفت که فقط برای مدت زمان پرس و جو وجود دارد.***

## WITH expression\_name[(column\_name [,...])]

## AS (CTE\_definition)

## SQL\_statement;

### Example:

1.

WITH CTE\_DEPT\_COURSES (DEPTNAME, CONUM) AS (

SELECT DEPT.DENAME , COUNT(DISTINCT COID)

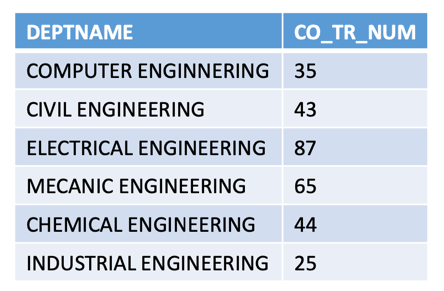
FROM DEPT JOIN COT ON DEPT.DEID = COT.CODEID

GROUP BY DEPT.DENAME

)

SELECT DEPTNAME , CONUM FROM CTE\_DEPT\_COURSES

WHERE DEPTNAME LIKE ‘%ENGINEERING’;

2.

WITH CTE\_DEPT\_TR\_COURSES (DEPTNAME, CO\_TR\_NUM) AS (

SELECT DEPT.DENAME , COUNT(DISTINCT STCOT.COID)

FROM DEPT JOIN COT ON DEPT.DEID = COT.CODEID

JOIN STCOT ON COT.COID = STCOT.COID

WHERE STCOT.TR = 1 AND STCOT.YRYR = ‘99-00’

GROUP BY DEPT.DENAME

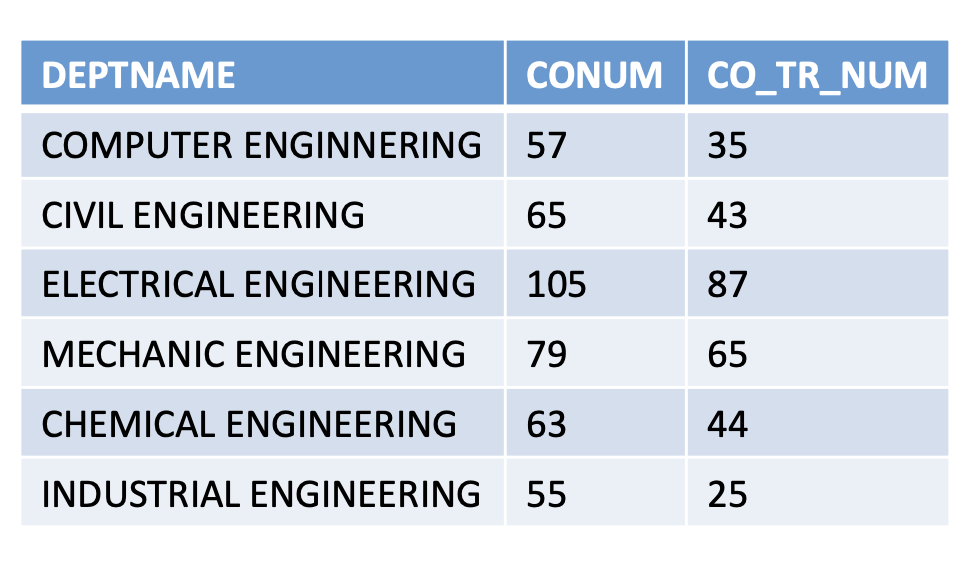
)

SELECT DEPTNAME , CO\_TR\_NUM FROM CTE\_DEPT\_TR\_COURSES

WHERE DEPTNAME LIKE ‘%ENGINEERING’;

3.

WITH CTE\_DEPT\_COURSES (DEPTNAME, CONUM) AS (

SELECT DEPT.DENAME , COUNT(DISTINCT COID)

FROM DEPT JOIN COT ON DEPT.DEID = COT.CODEID

GROUP BY DEPT.DENAME

),

CTE\_DEPT\_TR\_COURSES (DEPTNAME, CO\_TR\_NUM) AS (

SELECT DEPT.DENAME , COUNT(DISTINCT STCOT.COID)

FROM DEPT JOIN COT ON DEPT.DEID = COT.CODEID

JOIN STCOT ON COT.COID = STCOT.COID

WHERE STCOT.TR = 1 AND STCOT.YRYR = ‘99-00’

GROUP BY DEPT.DENAME

)

SELECT CTE\_DEPT\_COURSES.DEPTNAME ,

CTE\_DEPT\_COURSES.CO\_NUM ,

CTE\_DEPT\_COURSES.CO\_TR\_NUM

FROM CTE\_DEPT\_TR\_COURSES JOIN CTE\_DEPT\_COURSES

ON CTE\_DEPT\_TR\_COURSES.DEPTNAME = CTE\_DEPT\_COURSES.DEPTNAME

WHERE CTE\_DEPT\_COURSES.DEPTNAME LIKE ‘%ENGINEERING’;

point : ***can cte expression include something without select ?***

No, a Common Table Expression (CTE) in SQL cannot include anything other than a SELECT statement.

# PIVOT

## The PIVOT operator is used to rotate rows into columns, effectively transforming the output of a query by rotating the result set.

-- Simple PIVOT syntax

SELECT \*

FROM (

-- Subquery to select data

SELECT column1, column2, pivot\_column, value\_column

FROM your\_table

) AS SourceTable

PIVOT (

-- Pivot clause

AGGREGATE\_FUNCTION(value\_column)

FOR pivot\_column IN ([value1], [value2], ..., [valueN])

) AS PivotTable;

* **your\_table: The source table containing the data to be pivoted.**
* **pivot\_column: The column whose values will become the new column headers.**
* **value\_column: The column containing the values to be aggregated and placed in the new columns.**
* **AGGREGATE\_FUNCTION: The aggregate function to be applied to the values if there are duplicate combinations of pivot\_column and value\_column.**

### Example:

### 1.

### Let's assume the SalesData table looks like this:

Product | Month | Revenue

--------|------------|--------

A | January | 1000

A | February | 1200

A | March | 1500

A | April | 1300

A | May | 1600

A | June | 1800

B | January | 800

B | February | 900

B | March | 1000

B | April | 1100

B | May | 950

B | June | 1200

**SELECT \***

**FROM (**

**SELECT Product, Month, Revenue**

**FROM SalesData**

**) AS SourceTable**

**PIVOT (**

**SUM(Revenue)**

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

**) AS PivotTable;**

The expected output would now include the additional months:

Product | January | February | March | April | May | June

--------|---------|----------|-------|-------|------|-----

A | 1000 | 1200 | 1500 | 1300 | 1600 | 1800

B | 800 | 900 | 1000 | 1100 | 950 | 1200

### 2.

Assuming the ProductSales table looks like this:

Product | Category | Quarter | Revenue

--------|-------------|---------|--------

A | Electronics | Q1 | 1000

A | Electronics | Q2 | 1200

A | Clothing | Q1 | 800

A | Clothing | Q2 | 900

B | Electronics | Q1 | 1500

B | Electronics | Q2 | 1300

B | Clothing | Q1 | 700

B | Clothing | Q2 | 850

The PIVOT query for this scenario could be:

**-- PIVOT query for product sales by category and quarter**

**SELECT \***

**FROM (**

**SELECT Product, Category, Quarter, Revenue**

**FROM ProductSales**

**) AS SourceTable**

**PIVOT (**

**SUM(Revenue)**

**FOR Quarter IN ([Q1], [Q2])**

**) AS PivotTable;**

The expected output would be:

Product | Category | Q1 | Q2

--------|-------------|------|------

A | Electronics | 1000 | 1200

A | Clothing | 800 | 900

B | Electronics | 1500 | 1300

B | Clothing | 700 | 850

# COALESCE

**The COALESCE function is used to return the first non-null expression among its arguments. It takes multiple parameters and returns the first non-null value.**

**COALESCE(expression1, expression2, ..., expressionN)**

#### ***کاربرد آن به دو شکل خواهد بود :***

#### ***اولین مقدار NOT- NULL را بر می‌گرداند.***

#### ***مقادیر NULL یک جدول را با مقدار دلخواهی (از ورودی)جایگزین می‌کند****.*

### Example:

### 1.

**-- Sample COALESCE usage**

**DECLARE @value1 INT = NULL;**

**DECLARE @value2 INT = 42;**

**-- Using COALESCE to get the first non-null value**

**SELECT COALESCE(@value1, @value2) AS Result;**

In this example, COALESCE(@value1, @value2) returns 42 because @value1 is NULL, and the first non-null value encountered is @value2.

**2.**

**-- Using COALESCE in a SELECT query**

**SELECT**

**COALESCE(column1, 'Default') AS Result1,**

**COALESCE(column2, 'N/A') AS Result2,**

**COALESCE(column3, 0) AS Result3**

**FROM your\_table;**

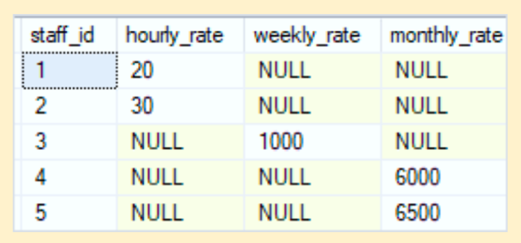
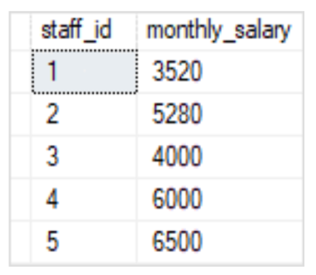
In this query, if column1 is NULL, it will be replaced with the string 'Default'. If column2 is NULL, it will be replaced with the string 'N/A'. If column3 is NULL, it will be replaced with the numeric value 0.

**3.**

**SELECT staff\_id,**

**COALESCE(hourly\_rate\*22\*8, weekly\_rate\*4, monthly\_rate) AS monthly\_salary**

**FROM salaries;**

****

**=>**

# NULLIF

#### ***تابعی است که دو پارامتر ورودی را مقایسه می کند.***

#### ***اگر دو پارامتر با هم برابر باشند، NULLبرمی گرداند.*** ) NULLIF(expression1, expression2

#### ***در غیر اینصورت مقدار پارامتر اول را برمی گرداند.***

### Example:

### 1.

**-- Sample NULLIF usage**

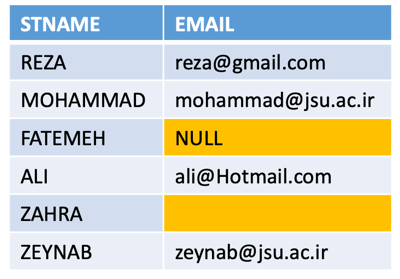
**DECLARE @value1 INT = 42;**

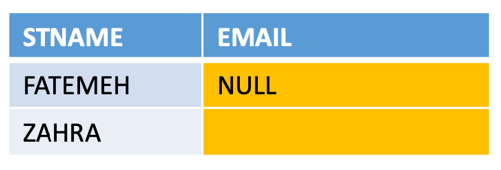
**DECLARE @value2 INT = 42;**

**-- Using NULLIF to return NULL if values are equal**

**SELECT NULLIF(@value1, @value2) AS Result;**

return NULL because @value1 is equal to @value2.

****

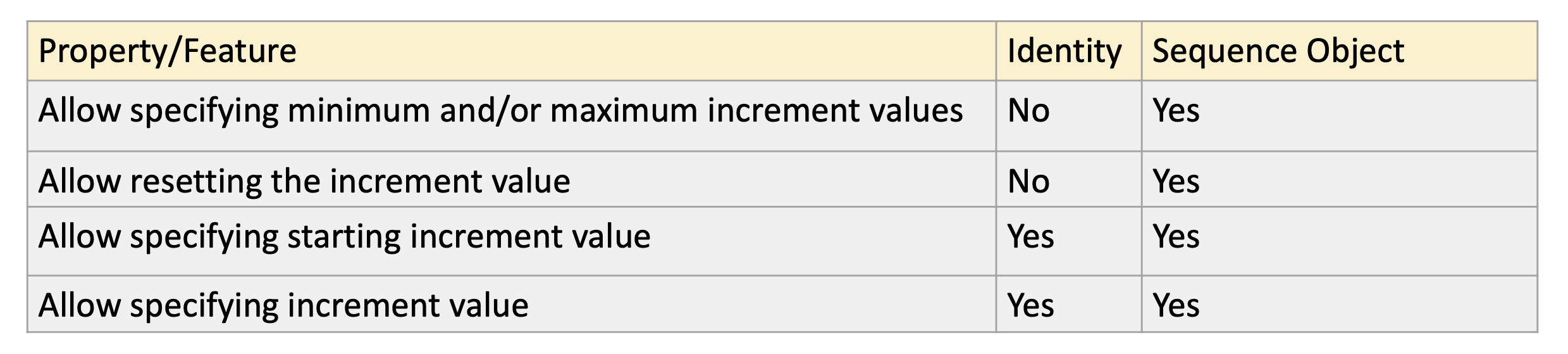
**2.**

**SELECT STNAME , EMAIL**

**FROM STT**

**WHERE NULLIF(EMAIL,’’) IS NULL =>**

# SEQUENCE



## CREATE SEQUENCE [schema\_name.] sequence\_name

## [ AS integer\_type ]

## [ START WITH start\_value ]

## [ INCREMENT BY increment\_value ]

## [ { MINVALUE [ min\_value ] } | { NO MINVALUE } ]

## [ { MAXVALUE [ max\_value ] } | { NO MAXVALUE } ]

## *وجه تمایز آن با IDENTITY ، این سه خط هستند* [ CYCLE | { NO CYCLE } ]

### point : use SEQUENCE like this

-- Create a sequence

CREATE SEQUENCE YourSequence

AS INT

START WITH 1

INCREMENT BY 1;

-- Use the sequence in a table

CREATE TABLE YourTable

(

ID INT PRIMARY KEY **DEFAULT (NEXT VALUE FOR YourSequence), // = IDENTITY(1,1)**

Column1 VARCHAR(50),

Column2 INT

);

-- Insert values into the table

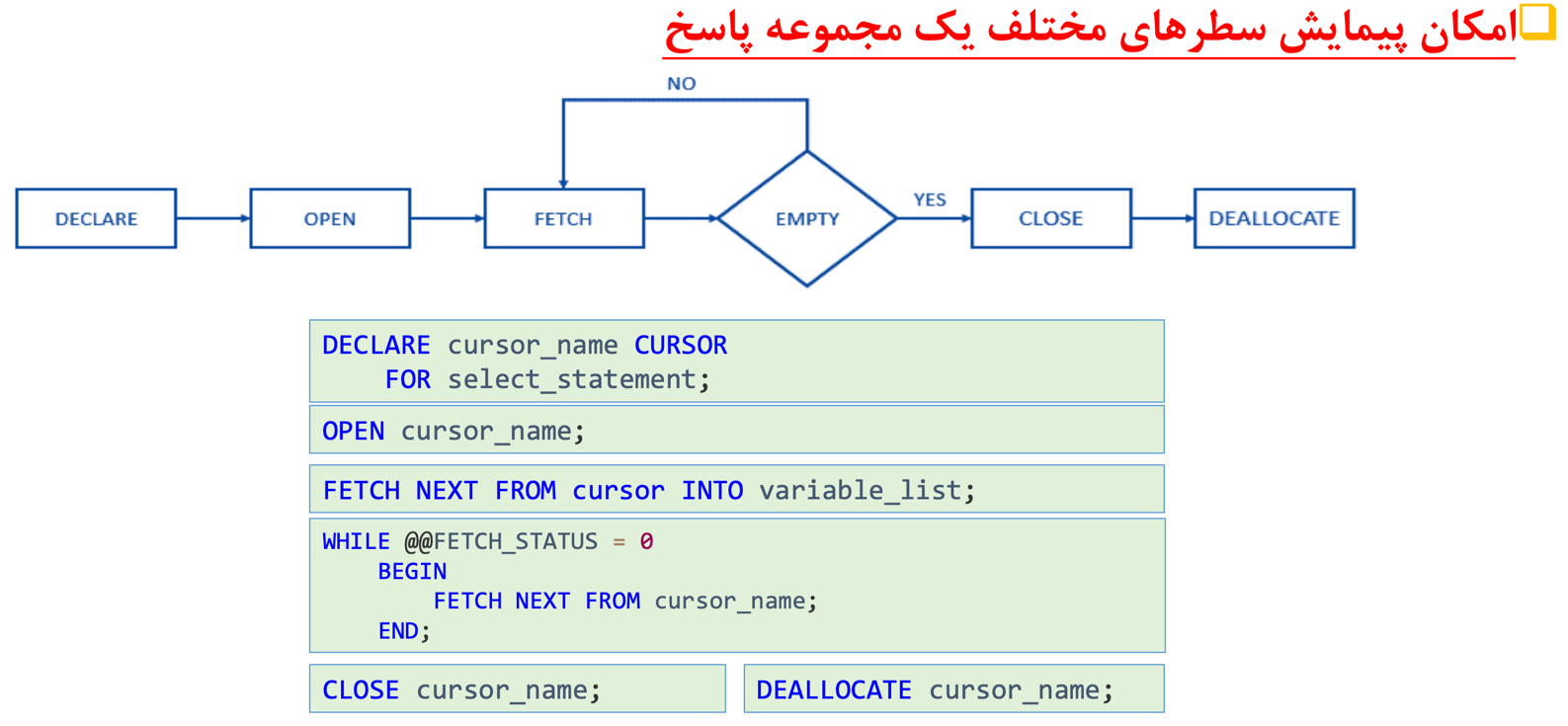
INSERT INTO YourTable (Column1, Column2) VALUES ('Value1', 42);

# CURSOR

#### **CURSOR *برای پردازش یک مجموعه نتیجه در یک ردیف در یک زمان استفاده می شود.* CURSOR *راهی برای تکرار در ردیف های یک مجموعه نتیجه ارائه می دهند و به شما این امکان را می دهند که عملیات را روی هر ردیف به صورت جداگانه انجام دهید.***

Explanation:

1. **Declare Variables:** Declare variables to store the values retrieved from the cursor.
2. **Declare Cursor:** Declare the cursor using the **DECLARE CURSOR** statement. The cursor is associated with a **SELECT** statement that defines the result set.
3. **Open Cursor:** Open the cursor to make it ready for fetching rows.
4. **Fetch Rows:** Use the **FETCH** statement to retrieve rows from the cursor into variables. This is done in a loop until there are no more rows (**@@FETCH\_STATUS = 0**).
5. **Process Rows:** Perform operations on each row as needed. In this example, a **PRINT** statement is used.
6. **Close and Deallocate:** Close the cursor to release resources and deallocate it.



### Example:

### 1.

-- Declare variables

DECLARE @EmployeeID INT;

DECLARE @FirstName VARCHAR(50);

DECLARE @LastName VARCHAR(50);

-- Declare the cursor

DECLARE EmployeeCursor CURSOR FOR

SELECT EmployeeID, FirstName, LastName

FROM Employees;

-- Open the cursor

OPEN EmployeeCursor;

-- Fetch the first row into variables

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @FirstName, @LastName;

-- Loop through the result set

WHILE @@FETCH\_STATUS = 0

BEGIN

-- Process the current row (replace this with your logic)

PRINT 'EmployeeID: ' + CAST(@EmployeeID AS VARCHAR(10)) +

', FirstName: ' + @FirstName +

', LastName: ' + @LastName;

-- Fetch the next row into variables

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @FirstName, @LastName;

END

-- Close and deallocate the cursor

CLOSE EmployeeCursor;

DEALLOCATE EmployeeCursor;

**2.**

-- Create a sample table

CREATE TABLE Products

(

ProductID INT PRIMARY KEY,

ProductName VARCHAR(50),

Price DECIMAL(10, 2)

);

-- Insert some sample data

INSERT INTO Products (ProductID, ProductName, Price)

VALUES (1, 'Laptop', 1200.00),

(2, 'Smartphone', 699.99),

(3, 'Tablet', 299.50);

-- Declare variables

DECLARE @ProductID INT;

DECLARE @ProductName VARCHAR(50);

DECLARE @NewPrice DECIMAL(10, 2);

-- Declare the cursor

DECLARE ProductCursor CURSOR FOR

SELECT ProductID, ProductName, Price

FROM Products;

-- Open the cursor

OPEN ProductCursor;

-- Fetch the first row into variables

FETCH NEXT FROM ProductCursor INTO @ProductID, @ProductName, @NewPrice;

-- Loop through the result set

WHILE @@FETCH\_STATUS = 0

BEGIN

-- Perform some operation (for example, increase the price by 10%)

SET @NewPrice = @NewPrice \* 1.10;

-- Update the product price in the table

UPDATE Products

SET Price = @NewPrice

WHERE ProductID = @ProductID;

-- Print information about the update

PRINT 'Updated price for ' + @ProductName + ' to ' + CAST(@NewPrice AS VARCHAR(20));

-- Fetch the next row into variables

FETCH NEXT FROM ProductCursor INTO @ProductID, @ProductName, @NewPrice;

END

-- Close and deallocate the cursor

CLOSE ProductCursor;

DEALLOCATE ProductCursor;

-- Check the updated prices

SELECT \* FROM Products;

In this example, we use a cursor to iterate through the Products table, retrieve the ProductID, ProductName, and Price, increase the price by 10%, and then update the Price in the table. The PRINT statement is used to display information about the update for each product.

# INDEX

#### ***روشی سریع و کارآمد برای جستجوی و بازیابی داده ها از یک جدول فراهم می کند. ایندکس ها بر روی یک یا چند ستون از یک جدول ایجاد می شوند و می توانند به طور قابل توجهی عملکرد پرس و جوهای* SELECT *را با اجازه دادن به موتور پایگاه داده برای مکان یابی سریعتر ردیف ها بهبود بخشند.***

**CREATE INDEX index\_name**

**ON table\_name (column1, column2, ...);**

**Types of Indexes:**

1. **Single-Column Index: A single-column index is created on a single column of a table. It is the most basic type of index.**

CREATE INDEX idx\_ColumnName

ON TableName (ColumnName);

1. **Composite Index (Multiple Columns):** **A composite index is created on multiple columns. It is useful when queries involve conditions on multiple columns.**

CREATE INDEX idx\_Column1\_Column2

ON TableName (Column1, Column2);

1. **Unique Index:** **A unique index ensures that the indexed columns do not contain duplicate values. It is similar to a primary key constraint but allows for NULL values.**

CREATE UNIQUE INDEX idx\_ColumnName

ON TableName (ColumnName);

1. **Clustered Index:** **A clustered index determines the physical order of data in a table. The leaf nodes of the index contain the actual data pages.**

CREATE CLUSTERED INDEX idx\_ColumnName

ON TableName (ColumnName);

1. **Non-Clustered Index:** *یک شاخص غیر خوشه ای بر ترتیب فیزیکی داده های جدول تأثیر نمی گذارد. این یک ساختار جداگانه حاوی لیست مرتب شده ای از ارجاعات به داده های جدول ایجاد می کند.*

CREATE NONCLUSTERED INDEX idx\_ColumnName

ON TableName (ColumnName);

**point : Non-Clustered Index** uses **B-TREE**.

**Additional Considerations:**

* **Filtered Index**: *یک شاخص فیلتر شده بر اساس زیر مجموعه ای از ردیف ها است که شرایط خاصی را برآورده می کنند*

-- Create a filtered index on the IsActive column of the Employees table

CREATE INDEX idx\_ActiveEmployees

ON Employees (EmployeeID)

WHERE IsActive = 1;

* **Spatial Index**: *شاخص فضایی برای انواع داده های مکانی مانند هندسه و جغرافیا استفاده می شود*

-- Create a spatial index on the Location column of the SpatialData table

CREATE SPATIAL INDEX idx\_SpatialData\_Location

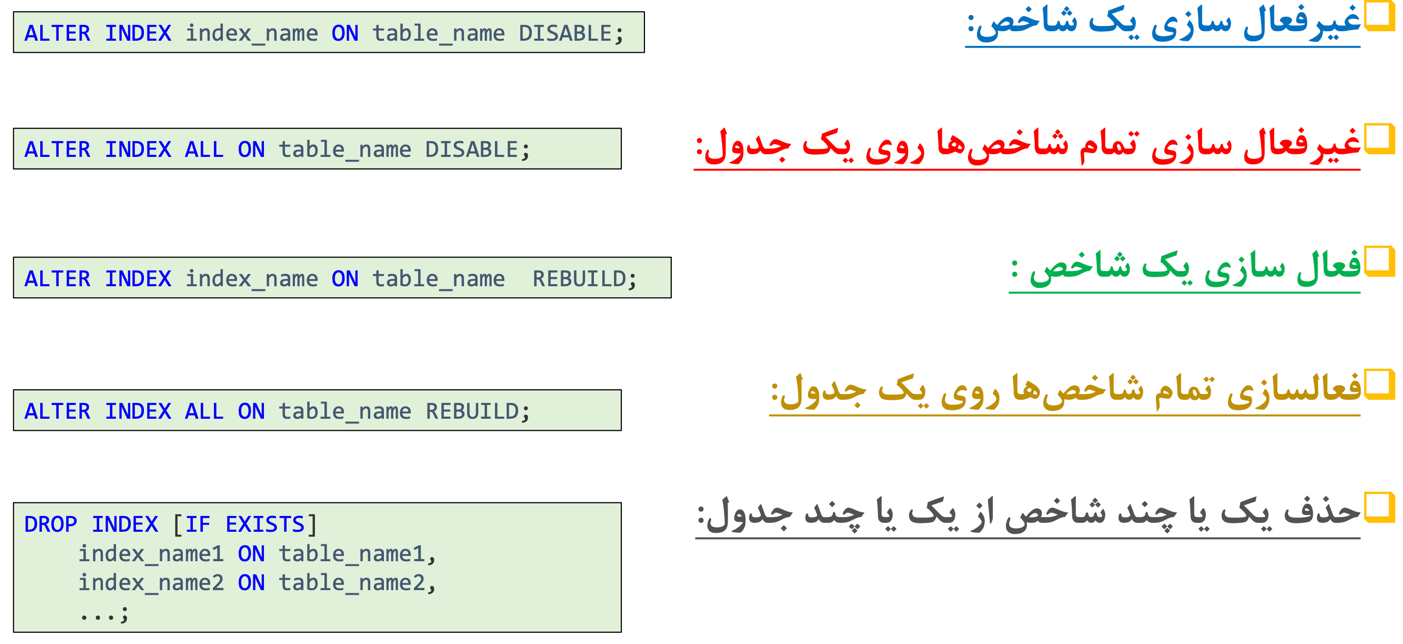
ON SpatialData (Location);

**point :**

The SQL Server Query Execution Plan can help analyze the impact of indexes on query performance.

#### *ایندکس‌ها عملکرد خواندن را بهبود می‌بخشند اما ممکن است بر عملکرد نوشتن تأثیر بگذارند، زیرا موتور پایگاه داده باید ساختار فهرست را در طول درج‌ها، به‌روزرسانی‌ها و حذف‌ها حفظ کند.*

**point : Primary keys automatically have a unique index created on them.**

****

#### *هر جدول میتواند یک یا چند Non-clustered indexداشته باشد*point **:**

**point :** In SQL Server, **a table can have only one clustered index**. The clustered index determines the physical order of data in the table, and therefore, there can be only one way to organize the data at the physical level. When you create a new clustered index on a table, any existing clustered index on that table (if one exists) will be dropped.

**point :**

#### *ترتیب فیزیکی داده ها در یک جدول به نحوه ذخیره ردیف ها بر روی دیسک یا یک رسانه ذخیره سازی اشاره دارد. در زمینه SQL Server و دیگر سیستم‌های مدیریت پایگاه داده رابطه‌ای (RDBMS)، ترتیب فیزیکی توسط شاخص خوشه‌ای جدول تعیین می‌شود. در اینجا چند نکته کلیدی در مورد ترتیب فیزیکی داده ها در یک جدول با شاخص خوشه ای وجود دارد:*

#### ***شاخص خوشه ای***

#### *هنگامی که جدولی دارای نمایه خوشه‌ای است، ردیف‌های جدول بر اساس ترتیب کلید فهرست خوشه‌بندی شده به‌صورت فیزیکی بر روی رسانه ذخیره‌سازی سازماندهی می‌شوند.*

#### *صفحات داده های جدول در یک ساختار شاخص درخت مانند B ساختار یافته اند و گره های برگ این ساختار شامل ردیف های داده واقعی به ترتیب مشخص شده توسط کلید فهرست خوشه ای است.*

#### *اولین ستون کلید شاخص خوشه ای، تعیین کننده اصلی نظم فیزیکی است.*

#### ***کلید اولیه و شاخص خوشه ای***

#### *در بسیاری از موارد، کلید اصلی جدول به عنوان شاخص خوشه ای استفاده می شود. با این حال، این یک قانون سختگیرانه نیست و یک جدول می‌تواند ستون یا مجموعه‌ای از ستون‌های متفاوت را به عنوان شاخص خوشه‌بندی شده داشته باشد.*

#### *اگر کلید اصلی به طور صریح تعریف نشده باشد یا اگر یک شاخص خوشه‌بندی شده مشخص نشده باشد، SQL Server ممکن است یک شناسه منحصربه‌فرد پنهان (به نام "شناسه ردیف" یا "RID") به عنوان یک کلید خوشه‌بندی ایجاد کند.*

#### ***کارایی پرس و جوهای محدوده***

#### *وقتی داده ها به ترتیب کلید فهرست خوشه ای سازماندهی می شوند، پرس و جوهای محدوده (پرس و جوهایی که طیفی از مقادیر را در بر می گیرند) می توانند کارآمدتر باشند زیرا ردیف های مرتبط از نظر فیزیکی به یکدیگر در دیسک نزدیک هستند.*

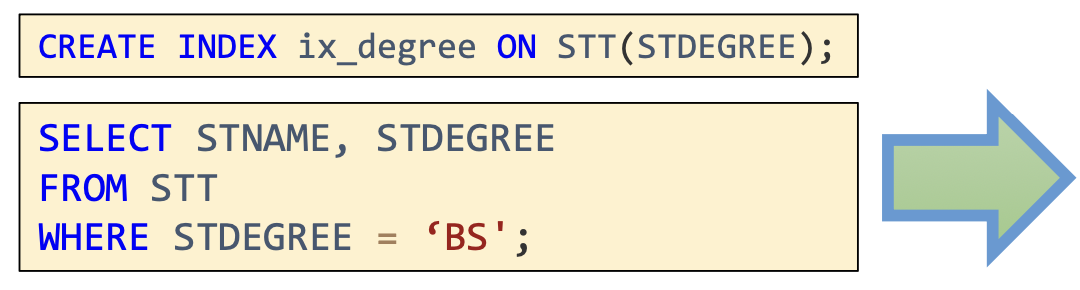
#### *با این حال، این به قیمت عملیات درج، به‌روزرسانی و حذف بالقوه کندتر است، به خصوص اگر کلید فهرست خوشه‌ای به طور یکنواخت افزایش نیابد.*

#### ***شاخص های غیر خوشه ای***

#### *جدولی با شاخص خوشه ای همچنان می تواند نمایه های غیر خوشه ای داشته باشد. شاخص های غیر خوشه ای مسیرهای دسترسی اضافی به داده ها را فراهم می کنند اما بر ترتیب فیزیکی ردیف ها تأثیر نمی گذارند.*

#### *نمایه های غیر خوشه ای یک نقشه بین مقادیر نمایه شده و مکان فیزیکی متناظر سطرها در نمایه خوشه ای ذخیره می کنند.*

#### *توجه به این نکته مهم است که ترتیب فیزیکی داده ها جزییات سطح پایینی است که توسط موتور پایگاه داده مدیریت می شود. اکثر پرس و جوها باید به گونه ای نوشته شوند که به ترتیب فیزیکی بستگی نداشته باشد و بهینه ساز پرس و جو موتور پایگاه داده بدون توجه به ترتیب فیزیکی، مسئول انتخاب کارآمدترین مسیر دسترسی است.*

**

**point :**

#### *در SQL Server، عبارت INCLUDE در زمینه نمایه های غیر خوشه ای برای گنجاندن ستون های غیر کلیدی در ساختار فهرست استفاده می شود. گنجاندن ستون‌های غیرکلیدی در فهرست به شما امکان می‌دهد پرس و جوهای بیشتری را پوشش دهید و عملکرد پرس و جو را بهبود ببخشید، به خصوص زمانی که این ستون‌ها در بندهای SELECT، WHERE و ORDER BY پرس و جوهای شما استفاده می‌شوند.*

CREATE NONCLUSTERED INDEX index\_name

ON table\_name (column1, column2, ...)

INCLUDE (included\_column1, included\_column2, ...);

Example:

1.

CREATE UNIQUE INDEX ix\_degree\_inc ON STT(STDEGREE)

INCLUDE(STNAME)

2.

CREATE NONCLUSTERED INDEX idx\_Customers\_Name

ON Customers (FirstName, LastName)

INCLUDE (Email, Phone);

* **FirstName** and **LastName** are the key columns, defining the sort order of the index.
* **Email** and **Phone** are included columns, meaning their values are included in the leaf level of the index to cover queries.

# VIEW

# @@ system\_variable

In Transact-SQL (T-SQL), @@ is a prefix used to indicate a system variable. System variables are special variables in SQL Server that provide information about the current state of the database or the results of the last SQL operation. They are prefixed with @@ to distinguish them from user-defined variables.

### Example:

**1.** **@@ROWCOUNT**

-- Example 1: UPDATE statement

UPDATE Employees

SET Salary = Salary \* 1.1

WHERE Department = 'IT';

-- Check the number of rows affected

SELECT @@ROWCOUNT AS 'RowsAffected';

**2. @@ROWCOUNT**

-- Example 2: DELETE statement

DELETE FROM Customers

WHERE LastPurchaseDate < '2022-01-01';

-- Check the number of rows affected

SELECT @@ROWCOUNT AS 'RowsAffected';

**3.** **@@FETCH\_STATUS**

DECLARE EmployeeCursor CURSOR FOR

SELECT EmployeeID, FirstName, LastName

FROM Employees;

OPEN EmployeeCursor;

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @FirstName, @LastName;

WHILE @@FETCH\_STATUS = 0

BEGIN

-- Process the current row

FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @FirstName, @LastName;

END

CLOSE EmployeeCursor;

DEALLOCATE EmployeeCursor;

**point :**

* 0 (FETCH\_SUCCESS): The fetch operation was successful. This means that a row was successfully fetched into the variables, and the loop should continue to process the row.
* -1 (FETCH\_ERROR): An error occurred during the fetch operation. This could be due to various reasons such as a conversion error or other runtime issues. It indicates that something went wrong, and you may need to handle the error appropriately.
* -2 (FETCH\_NO\_DATA): No more rows are available to fetch. This indicates that the result set has been fully processed, and there are no additional rows to retrieve. It is commonly used as a condition to exit the fetch loop.

# KEY

In SQL, there are several types of keys that serve different purposes.

1. **Primary Key**: A primary key uniquely identifies each row in a table. It must be unique and cannot contain null values. Each table can have only one primary key.

2. **Unique Key**: A unique key ensures that each value in the column is unique, but it allows null values. A table can contain multiple unique keys.

3. **Foreign Key**: A foreign key establishes a relationship between two tables. It is a field in one table that links to the primary key of another table. It helps maintain referential integrity, enforcing data consistency and integrity across related tables.

4. **Candidate Key**: A candidate key is a set of attributes that can uniquely identify a database record. It can be used as a primary key.

5. **Composite Key**: A composite key is a combination of two or more columns that together uniquely identify a record in a table. It is used when a single column cannot uniquely identify a record.

6. **Super Key**: A super key is a set of one or more keys that can uniquely identify a record. It may include additional attributes other than the minimal set required for identification.

Overall, the differences between these keys lie in the uniqueness, nullability, purpose, and relationships they establish in a database.

A **candidate key** is a unique identifier for a database table, which means that it can uniquely identify each row or record in the table. It is a set of one or more columns that has the following properties:

1. Uniqueness: Each value in the candidate key must be unique, meaning that no two rows can have the same combination of values for the candidate key columns.

2. Minimality: No proper subset of the candidate key should be able to uniquely identify a row in the table. This means that removing any column from the candidate key would make it unable to uniquely identify a row.

Candidate keys are important in database design as they help ensure data integrity and enable efficient retrieval of data. They serve as a basis for creating primary keys, which are selected from the set of candidate keys. The primary key is then used to uniquely identify each row in a table.

For example, consider a table named "Employees" with columns "Employee\_ID," "First\_Name," and "Last\_Name." In this case, the combination of the "Employee\_ID" column can be a candidate key as it is unique and removing any other column would make it unable to uniquely identify a row. The candidate key can be chosen as the primary key for this table, and it is used for referencing and joining with other tables.

### point : can two column be primary key in a table ?

No, a primary key in a table is a unique identifier for each row. It cannot consist of two columns. However, you can have a composite primary key, which is a primary key that is composed of two or more columns. This means that the combination of values in those columns must be unique for each row in the table.