

## **INTRODUCCIÓN**

En este manual aprenderemos sobre el lenguaje de manipulación de datos (DML) para ello se requieren, conocimientos básicos del lenguaje de definición de datos (DDL) para poder comprender algunos temas. Se darán a conocer los diferentes tipos de tablas, así como las funciones que se realizar con ellas, podremos aprender cómo aplicar las estructuras de control dentro de las bases de datos, y reconocer los diferentes tipos de manejadores de errores. Por consiguiente, otro de los temas aplicados son los servicios en la nube de la base de datos en la plataforma de Azure Microsoft.

## LENGUAJE DE MANIPULACION DE DATOS

El lenguaje de manipulación de datos (DML) permite realizar consultas, modificación y eliminación de datos dentro de una base de datos.

Las sentencias son las siguientes:

- SELECT. Permite consultar.
- INSERT. Inserta nuevos registros.
- UPDATE. Permite modificar.
- DELETE. Permite eliminar.
- MERGE. Permite realizar varias acciones dentro de una misma sentencia.

También existen condicionantes que permiten filtrar y manipular datos:

- WHERE
- Operadores lógicos
- JOIN
- UNION
- ORDER BY
- GROUP BY

## SENTENCIA SELECT

```
SELECT * | COL1,COL2,COLn...  
FROM esquema.tabla  
WHERE valor1[Condicion] valor2
```

Figura 14 Sintaxis del comando select

Dónde:

- El “\*” devuelve todas las columnas de la “table” que se está consultando, también se puede especificar las columnas que devolverá la consulta.
- Valor1 y valor2 pueden ser columnas de la “table” o algún otro dato.
- Condición puede ser un operador lógico o alguna palabra reservada que indique una condición como BETWEEN.

## JOINS

La sentencia JOIN permite combinar dos conjuntos de datos con una determinada condición.

Existen estos tipos de JOIN:

- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- CROSS JOIN

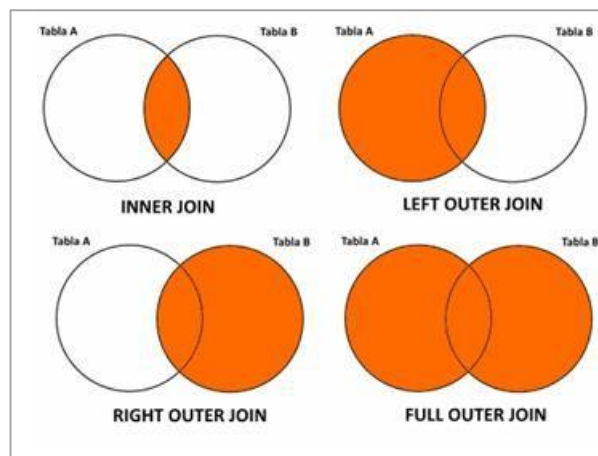


Figura 15 Representación gráfica de los datos que considera el inner

Ejemplo:

```
SELECT *
FROM esquema.tabla_1 alias1
INNER JOIN esquema.tabla_2 alias2 ON alias1.col = alias2.col

SELECT *
FROM esquema.tabla_1 alias1
LEFT OUTER JOIN esquema.tabla_2 alias2 ON alias1.col = alias2.col

SELECT *
FROM esquema.tabla_1 alias1
RIGHT OUTER JOIN esquema.tabla_2 alias2 ON alias1.col = alias2.col

SELECT *
FROM esquema.tabla_1 alias1
FULL OUTER JOIN esquema.tabla_2 alias2 ON alias1.col = alias2.col

SELECT *
FROM esquema.tabla_1 alias1
CROSS JOIN esquema.tabla_2 alias2
```

Figura 16 Sintaxis de aplicabilidad del inner

## UNION

La sentencia UNION permite unir dos consultas en una sola.

Restricciones:

- El resultado de las dos consultas debe de contener el mismo número de columnas.
- Las columnas en ambas consultas deben de coincidir en el tipo de dato.
- Los nombres de las columnas del resultado de UNION serán los de la primera consulta individual.
- Puede utilizar un GROUP BY en cada consulta individual pero no al resultado final.
- La cláusula ORDER BY puede ser utilizada para afectar el resultado final, pero no se puede usar en cada consulta individual.

Ejemplo:

```
SELECT Nombre, ApellidoPaterno, ApellidoMaterno
FROM ventas.clientes ct
UNION
SELECT NombreCompleto, Apellido_Pat, Apellido_Mat
FROM seguridad.Usuarios

SELECT Nombre, 'Ventas' AS Transaccion, COUNT(*) Cantidad
FROM venta.Ventas
GROUP BY Nombre
UNION
SELECT Nombre, 'Compras' AS Transaccion, COUNT(*) Cantidad
FROM compra.Compras
GROUP BY Nombre
ORDER BY Nombre
```

Figura 17 Sintaxis de ejemplos del comando union

## GROUP BY Y ORDER BY

La cláusula “GROUP BY” permite agrupar registros iguales bajo ciertos criterios en uno solo.

```
SELECT col1,col2,coln,<función de agregado>  
FROM esquema.tabla  
GROUP BY col1,col2,coln
```

Figura 18 Sintaxis de uso de funciones agregadas

La cláusula “ORDER BY” permite ordenar de manera descendente o ascendente los registros bajo ciertos criterios.

```
SELECT col1,col2,coln  
FROM esquema.tabla  
ORDER BY col1,col2
```

Figura 19 Sintaxis de uso para ordenar los datos

# INSERT

La sentencia INSERT permite añadir registros a una tabla dentro de una base de datos.

```
INSERT INTO NombreTabla [(Campo1, ..., CampoN)] VALUES (Valor1, ..., ValorN)
```

Figura 20 Sintaxis para agregar registros en una tabla

Dónde:

- **Nombre tabla:** la tabla en la que se van a insertar las filas.
- **(Campo1, ..., CampoN):** representa el campo o campos en los que vamos a introducir valores.
- **(Valor1, ..., ValorN):** representan los valores que se van a almacenar en cada campo.

Se puede realizar una inserción masiva de registros utilizando la sentencia INSERT de la siguiente forma:

Ejemplo:

```
INSERT INTO NombreTabla [(Campo1, ..., CampoN)]
SELECT ...

Ejemplos:
INSERT INTO Region VALUES(35, 'Madrid')
INSERT INTO Region (RegionID, RegionDescription) VALUES(35, 'Madrid')
INSERT INTO Region (RegionID, RegionDescription) VALUES(35, 'Madrid'),
(36, 'Barcelona')

INSERT INTO Customers
SELECT * FROM NewCustomers WHERE Country = 'Spain'
INSERT INTO Customers (Country, CustomerName)
SELECT Country, CustomerName FROM NewCustomers WHERE Country = 'Spain'
```

Figura 21 Ejemplo de diferentes usos de agregado de datos

## UPDATE

La sentencia UPDATE permite modificar los registros de una tabla en la base de datos.

```
UPDATE esquema.tabla SET col_1 = valor1, col_2 = valor2
```

Figura 22 Sintaxis para modificar datos en un registro

UPDATE se puede combinar con:

- WHERE
- FROM
- JOIN

Ejemplo:

```
UPDATE esquema.tabla SET col1 = valor, col2 = valor
WHERE col_id = valor

UPDATE alias1
    SET alias1.col1 = alias2.col1,
        alias1.col2 = alias2.col2
FROM esquema.tabla1 alias1
JOIN esquema.tabla2 AS alias2 ON alias2.id = alias1.id
WHERE alias1.col3 <> valor
```

Figura 23 Ejemplo de uso con sintaxis en update



## DELETE

La sentencia DELETE permite eliminar registros de una tabla en la base de datos.

Ejemplo:

```
DELETE FROM esquema.Tabla

Ejemplos:
DELETE Employees WHERE EmployeeID = 9

DELETE Customers WHERE LastName LIKE '%Desconocido%'

DELETE Products WHERE CategoryID =
(SELECT CategoryID FROM Categories
WHERE CategoryName = 'Beverages')

DELETE Products WHERE CategoryID =
(SELECT CategoryID FROM Categories WHERE CategoryName =
'Beverages')
AND UnitPrice > 50
```

Figura 24 Ejemplo de aplicación en comando delete

## MERGE

La sentencia MERGE nos permite insertar, actualizar o borrar filas de acuerdo a los resultados que se realizan de la combinación de una tabla, con otro origen de datos, esta última puede ser un CTE, una variable tipo tabla, entre algunas otras:

```
MERGE INTO <tablaobjetivo> AS obj
USING <tablafuente> AS fuente
ON <clausula coincidencia>
WHEN MATCHED [ AND <clausula >]
THEN <codigo >
WHEN NOT MATCHED [BY TARGET] [AND <clausula >]
THEN INSERT...
WHEN NOT MATCHED BY SOURCE [AND <clausula >]
THEN <codigo>
[OUTPUT ...]
```

Figura 25 Sintaxis de aplicabilidad en el comando merge

Dónde:

- MERGE INTO <tablaObjetivo>: Define la tabla a la cual le realizaremos las operaciones INSERT, UPDATE, o DELETE.
- USING <tablaFuente>: Define la tabla de la cual provienen los datos, aunque también se puede utilizar un CTE o tabla derivada entre algunas otras opciones. Lo más común es utilizar una tabla.
- ON <cláusula de coincidencia>: Define la cláusula utilizada para encontrar las coincidencias entre ambas tablas, fuente y destino, muy parecido al ON de un JOIN.
- WHEN MATCHED [ AND <clausula>] THEN <código>: Se utiliza cuando existen coincidencias a través de la cláusula ON, por lo tanto, la acción INSERT no está permitida; es posible utilizar dos cláusulas WHEN MATCHED, una para utilizar la acción UPDATE y

otra para la acción DELETE, la única condicionante es que deben tener filtros si se utilizan ambas.

- WHEN NOT MATCHED [BY TARGET] [AND <clausula>] THEN INSERT... Se utiliza cuando una fila existe en la fuente, pero no en el destino, por lo tanto, la única operación permitida es un INSERT.
- WHEN NOT MATCHED BY SOURCE [AND <clausula >] THEN <código> Es el caso contrario a la cláusula anterior, cuando la fila existe en la tabla destino, pero no en la fuente, no se puede aplicar una operación INSERT, pero si UPDATE y DELETE, también se puede declarar dos cláusulas de este tipo al igual que la cláusula WHEN MATCHED, con la misma condicionante que deben tener filtros.
- OUTPUT Se pueden obtener los datos insertados, eliminados y actualizados por medio de las palabras reservadas insert y delete

Ejemplo:

Tabla origen (@origen)				Tabla destino (@destino)			
clave	tamano	cantidad	nombre	clave	tamano	cantidad	nombre
1	CHICO	15	MOUSE	2	MEDIANO	16	DISCO DURO
2	CHICO	8	MEMORIA USB	3	MEDIANO	15	MONITOR
3	GRANDE	5	MONITOR	4	GRANDE	12	TECLADO Y MOUSE
4	MEDIANO	35	TECLADO	5	CHICO	50	CARGADOR
				6	CHICO	3	MEMORIA RAM

```

MERGE INTO @destino AS obj
USING @origen AS fuente
ON obj.clave = fuente.clave
WHEN MATCHED AND obj.cantidad > fuente.cantidad
THEN UPDATE SET obj.tamano = fuente.tamano,
                obj.cantidad = fuente.cantidad,
                obj.nombre = fuente.nombre
WHEN MATCHED AND obj.cantidad < fuente.cantidad
THEN DELETE
WHEN NOT MATCHED BY TARGET
THEN INSERT VALUES (fuente.clave,fuente.tamano,fuente.cantidad,fuente.nombre)
WHEN NOT MATCHED BY SOURCE AND obj.cantidad <= 25
THEN DELETE
WHEN NOT MATCHED BY SOURCE AND obj.cantidad > 25
THEN UPDATE SET obj.nombre = 'MODIFICADO'
OUTPUT $action AS theAction, inserted.*, deleted.*;

```

Figura 26 Ejemplo de uso del comando merge

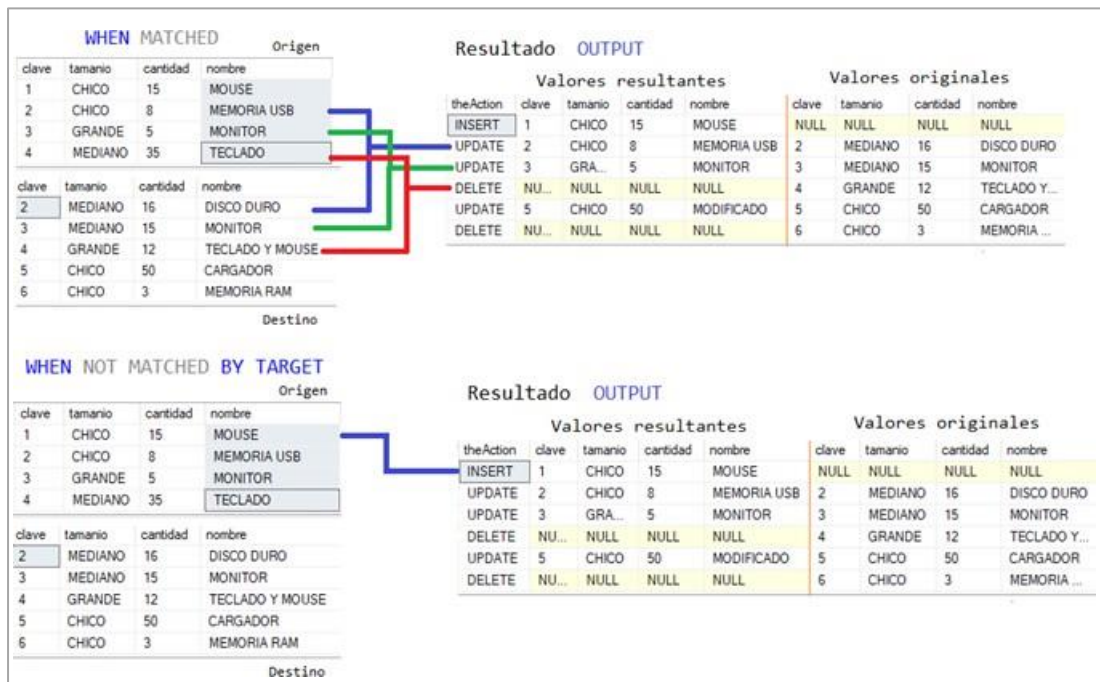


Figura 27 Combinación de tablas con otra para modificar datos

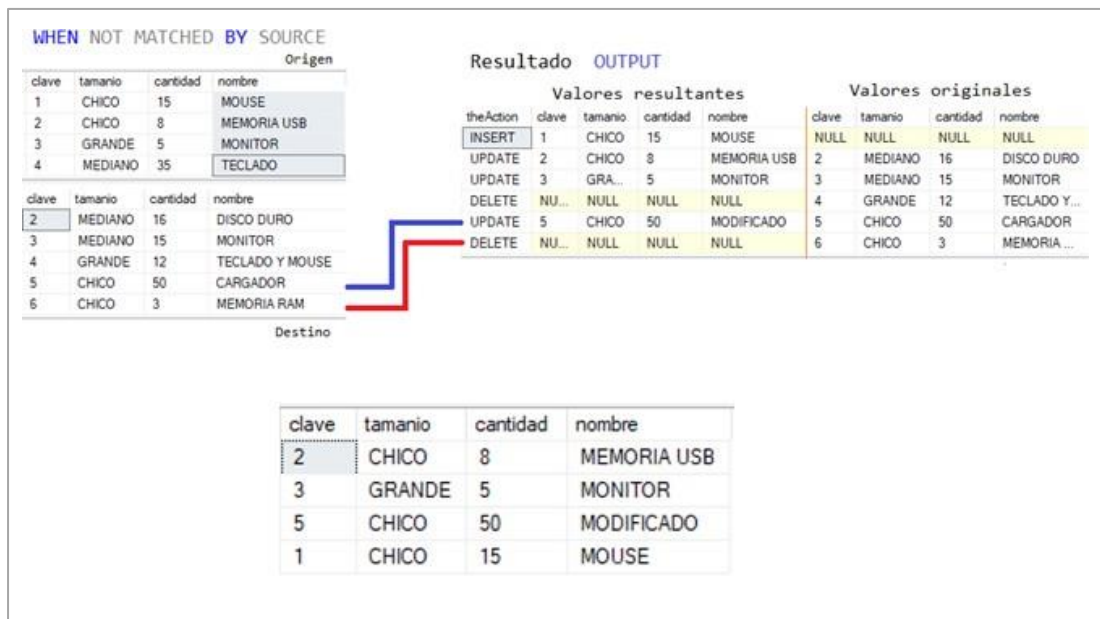


Figura 28 Combinación de tablas para eliminar y modificar tablas

## TABLAS TEMPORALES

Las tablas temporales son objetos que se almacenan en la base de datos tempdb, se usan para manipular grandes cantidades de datos.

Tipos de tablas temporales:

- **Tabla Temporal Local:** Se utiliza con un # (hash) para indicar que la tabla temporal será utilizada localmente (Solo será accedida a través de la conexión con cual fue creada). Esta tabla es destruida automáticamente en cuanto se cierre la conexión de SQL Server al menos que no se haya eliminado la tabla manualmente mediante el Query.
- **Tabla Temporal Global:** Se utiliza con doble ## (hash) para indicar que la tabla temporal será utilizada globalmente. Esta tabla puede utilizarla cualquier conexión de SQL Server y estas conexiones pueden modificar, borrar o eliminar la tabla temporal y sus registros. Al igual que las tablas temporales locales, esta se destruye cuando la última sesión utilizada por esta, cierre sesión.

Ejemplo:

```
CREATE TABLE #TablaTemporal
(
    Id int identity primary key,
    Orden int,
    Campo varchar(50)
)

INSERT INTO #TablaTemporal VALUES (1,'Primer campo')
INSERT INTO #TablaTemporal VALUES (2,'Segundo campo')

SELECT * FROM #TablaTemporal

DROP TABLE #TablaTemporal
```

Figura 29 Sintaxis para crear tablas temporales y agregar datos en tabla

Recomendaciones para usar Tablas Temporales:

- Evitar usar las tablas temporales.
- Si la tabla temporal es muy grande, utilizar índices.
- Eliminar la tabla temporal cuando ya no se necesite.
- Evitar crear tablas temporales con SELECT INTO dentro de procedimientos almacenados y funciones.
- No usar tablas temporales en Trigger ni en Transacciones.

## VARIABLES TIPO TABLAS

Las variables tipo tabla son tablas que se crean en memoria y solo existen durante la ejecución del código sql.

Se utiliza con arroba (@) para indicar que se utiliza como variable, al crear la tabla temporal como variable, se debe construir la estructura de la tabla en el Query.

Ejemplo:

```
DECLARE @Usuarios TABLE
(
    Clave INT IDENTITY PRIMARY KEY,
    Nombre VARCHAR(50),
    Apellidos VARCHAR(50),
    INDEX IX_Nombre NONCLUSTERED(Nombre)
)

INSERT INTO @Usuarios VALUES ('Samanta',
'Rodriguez')

SELECT * FROM @Usuarios
```

Figura 30 Ejemplo de uso para realizar variable tipo tabla y agregar datos

## TIPO DE DATOS TABLA

Un tipo de datos tabla es la definición de la estructura de una tabla creada por el usuario, lo cual permite crear variable tipo tablas y pasar estas variables como parámetros a procedimientos almacenados y funciones.

- El tipo de datos tabla no se puede asignar a una columna.
- El tipo de datos tabla no se puede modificar.
- Se crea con la instrucción CREATE TYPE.
- Se elimina con la instrucción DROP TYPE.
- Cuando se utiliza como parámetro en un procedimiento o función se declara con la propiedad READONLY.

Ejemplo:

```
CREATE TYPE compra.TLineasPedido AS TABLE
(
    CodArticulo INT NOT NULL
    ,Cantidad INT NOT NULL
    ,DescriArticulo VARCHAR(50) NOT NULL
    ,PrecioArticulo NUMERIC(9,2) NOT NULL
    PRIMARY KEY (CodArticulo)
)

DECLARE @lineasPedido compra.TLineasPedido

CREATE PROCEDURE compra.agregarLineasPedido
    @CodPedido INT, @LineasPedido compra.TLineasPedido READONLY
AS
BEGIN
    INSERT INTO compra.LineasPedido
    SELECT @CodPedido, CodArticulo, Cantidad, DescriArticulo, PrecioArticulo
    FROM @LineasPedido
END
```

Figura 31 Sintaxis aplicada para realizar un tipo de dato tabla y un procedimiento almacenado

## CONSULTAS CTES

Common Table Expressions o (CTEs). Se trata de una manera de definir un conjunto de datos temporal (como una tabla temporal) que sólo pervive mientras se ejecute nuestra consulta y no se almacena en ningún sitio. Sin embargo, nos permite consultarla y trabajar con ella como si fuese una tabla real de la base de datos. Estas CTE pueden hacer referencia a sí mismas y se puede usar varias veces en la misma consulta.

```
WITH  
NombreCTE [(campos devueltos)]  
AS  
(Subconsulta)  
SELECT * FROM NombreCTE...
```

Ejemplo:

```
WITH EmpTitle (JobTitle,numtibles)  
AS  
(  
    SELECT a.JobTitle,  
           COUNT(*) numtibles  
    FROM HumanResources.Employee a  
    GROUP BY a.JobTitle  
)  
SELECT B.BusinessEntityID,  
       B.JobTitle, C.numtibles  
FROM EmpTitle as C  
INNER JOIN HumanResources.Employee B  
    ON C.JobTitle=B.JobTitle;  
  
SELECT B.BusinessEntityID,  
       B.JobTitle, EmpTitle.numtibles  
FROM (  
    SELECT a.JobTitle,  
           COUNT(*) numtibles  
    FROM HumanResources.Employee as a  
    GROUP BY a.JobTitle  
    ) as EmpTitle  
INNER JOIN HumanResources.Employee B  
    ON EmpTitle.JobTitle=B.JobTitle;
```

Figura 32 Ejemplo de consulta de CTE's

Otra posibilidad que nos da las consultas CTEs es anidar dos o más consultas CTEs y permite que la siguiente consulta haga referencia a las anteriores.

```
WITH NombreCTE1 [(campos devueltos)]  
AS  
    (Subconsulta),  
    NombreCTE2 [(campos devueltos)]  
AS  
    (Subconsulta),  
    NombreCTE3 [(campos devueltos)]  
AS  
    (Subconsulta)
```

Figura 33 Sintaxis para anidar dos o más consultas CTE's



## CTES RECURSIVAS

Una expresión de tabla común (CTE) ofrece la gran ventaja de poder hacer referencia a sí misma, creando así una CTE recursiva. Una CTE recursiva es aquella en la que una CTE inicial se ejecuta varias veces para devolver subconjuntos de datos hasta que se obtenga el conjunto de resultados completo.

Todas las definiciones de consulta de miembro no recursivo deben colocarse antes de la primera definición de miembro recursivo y debe utilizarse un operador UNION ALL para combinar el último miembro no recursivo con el primer miembro recursivo.

```
WITH cte_name ( column name [,...n] )  
AS  
(  
    CTE_query_definition -- Query no recursivo.  
    UNION ALL  
    CTE_query_definition -- Query recursivo  
)  
SELECT * FROM cte_name
```

Figura 34 Sintaxis de CTE's recursivas

## **FUNCIONES DEL SQL SERVER**

SQL Server proporciona numerosas funciones integradas que ayudan a la manipulación y transformación de los datos.

- Funciones de agregado.
- Funciones de configuración.
- Funciones de conversión.
- Funciones del cursor.
- Tipos de datos y funciones de fecha y hora.
- Funciones JSON.
- Funciones lógicas.
- Funciones matemáticas.
- Funciones de metadatos.
- Funciones de seguridad.
- Funciones de cadena.
- Funciones del sistema.
- Funciones estadísticas.
- Funciones de texto e imagen.

Función	Descripción
AVG	Devuelve el promedio de los valores de un grupo
COUNT	Devuelve el numero de elementos de un grupo en INT
COUNT_BIG	Devuelve el numero de elementos de un grupo en BIGINT
MAX	Devuelve el valor máximo
MIN	Devuelve el valor mínimo
SUM	Devuelve la suma de todos los valores o solo de los valores DISTINCT de la expresión

Figura 35 Uso de las funciones agregadas

Estas funciones se pueden combinar con la cláusula OVER, la cláusula OVER permite dividir el resultado de la consulta en grupos con la palabra PARTITION BY y también permite ordenar el resultado con ORDER BY.

```

AVG ( [ ALL | DISTINCT ] expression ) [ OVER ( [ partition by clause ]
order by clause ) ]

COUNT ( { [ ALL | DISTINCT ] expression } | * )
COUNT ( [ ALL ] { expression | * } ) OVER ( [ <partition by clause> ] )

COUNT_BIG ( { [ ALL | DISTINCT ] expression } | * )
COUNT_BIG ( [ ALL ] { expression | * } ) OVER ( [ <partition by clause> ] )

MAX( [ ALL | DISTINCT ] expression )
MAX ( [ ALL ] expression ) OVER ( [ <partition by clause> ] [ <order by clause> ] )

MIN ( [ ALL | DISTINCT ] expression )
MIN ( [ ALL ] expression ) OVER ( [ <partition by clause> ] [ <order by clause> ] )

SUM ( [ ALL | DISTINCT ] expression )
SUM ( [ ALL ] expression ) OVER ( [ partition by clause ] order by clause )

```

Figura 36 Sintaxis de funciones agregadas

Función	Descripción
CAST	Convierte un tipo de dato a otro
CONVERT	Convierte un tipo de dato a otro y permite darle un estilo
TRY_CAST	Convierte un tipo de dato a otro, si esto falla devuelve NULL
TRY_CONVERT	Convierte un tipo de dato a otro y permite darle un estilo, si esto falla devuelve NULL

Figura 37 Descripción de Funciones de validación de errores

CAST (expression AS data\_type [ (length)])

CONVERT (data\_type [ (length)], expression [, style])

Función	Descripción
DATEADD	Agrega un valor a una fecha, ya sea día, mes o año
DATEDIFF	Devuelve el recuento (como un valor entero con firma) de los límites <u>datepart</u> que se han cruzado entre los valores <u>startdate</u> y <u>enddate</u> especificados
DATENAME	Esta función devuelve una cadena de caracteres que representa el parámetro <u>datepart</u> especificado del argumento <i>date</i> especificado.
DATEPART	Esta función devuelve un entero que representa el parámetro <u>datepart</u> especificado del parámetro <i>date</i> especificado
GETDATE	Devuelve la fecha del sistema
DAY	Devuelve un entero que representa el día del mes
MONTH	Devuelve un entero que representa el mes
YEAR	Devuelve un entero que representa el año

Figura 38 Descripción de Funciones tipo fecha

DATEADD (datepart, number, date)

DATEDIFF (datepart, startdate, enddate)

DATENAME (datepart, date)

DATEPART (datepart, date)

GETDATE ( )

DAY (date)

MONTH (date)

YEAR (date)

Función	Descripción
CHARINDEX	Busca una expresión de caracteres dentro de una segunda expresión de caracteres, y devuelve la posición inicial de la primera expresión si se encuentra.
CONCAT	Devuelve una cadena resultante de la concatenación, o la combinación, de dos o más valores de cadena de una manera integral
LEFT	Devuelve la parte izquierda de una cadena de caracteres con el número de caracteres especificado
LEN	Devuelve el número de caracteres de la expresión de cadena especificada, excluidos los espacios en blanco finales
LOWER	Devuelve una expresión de caracteres después de convertir en minúsculas los datos de caracteres en mayúsculas
LTRIM	Devuelve una expresión de caracteres tras quitar todos los espacios iniciales en blanco
REPLACE	Reemplaza todas las instancias de un valor de cadena especificado por otro valor de cadena

Figura 39 Descripción de uso de las funciones de cadena parte 1

CHARINDEX (expressionToFind , expressionToSearch [ , start\_location ])

CONCAT (string\_value1, string\_value2 [, string\_valueN ])

LEFT (character\_expression , integer\_expression)

LEN (string\_expression)

LOWER (character\_expression)

LTRIM (character\_expression)

REPLACE (string\_expression , string\_pattern , string\_replacement)

REPLICATE	Repite un valor de cadena un número especificado de veces
REVERSE	Devuelve el orden inverso de un valor de cadena
RIGHT	Devuelve la parte derecha de una cadena de caracteres con el número de caracteres especificado
RTRIM	Devuelve una cadena de caracteres después de trincar todos los espacios finales
STRING_SPLIT	Divide una cadena en filas de <u>subcadenas</u> , según un carácter separador especificado
SUBSTRING	Devuelve parte de una expresión de caracteres
UPPER	Devuelve una expresión de caracteres con datos de caracteres en minúsculas convertidos a mayúsculas
ROWNUMBER	

Figura 40 Descripción de uso de las funciones de cadena parte 2

REPLICATE (string\_expression ,integer\_expression)

REVERSE (string\_expression)

RIGHT (character\_expression , integer\_expression)

RTRIM (character\_expression)

STRING\_SPLIT (string , separator)

SUBSTRING (expression ,start , length)

UPPER (character\_expression)

Función	Descripción
ROWNUMBER	Enumera los resultados de un conjunto de resultados. Concretamente, devuelve el número secuencial de una fila dentro de una partición de un conjunto de resultados, empezando por 1 para la primera fila de cada partición
ISNULL	Sustituye el valor NULL por el valor especificado
ISNUMERIC	Determina si una expresión es un tipo numérico válido
NEWID	Crea un valor único del tipo <u>uniqueidentifier</u>
@@IDENTITY	Devuelve el último valor de identidad insertado
@@ROWCOUNT	Devuelve el número de filas afectadas por la última instrucción

Figura 41 Descripción de uso de validación de errores de cadena

ROW\_NUMBER ( ) OVER ([ PARTITION BY value\_expression , ... [ n ] ]  
order\_by\_clause)

ISNULL (check\_expression , replacement\_value)

ISNUMERIC (expression)

NEWID ( )

@@IDENTITY

@@ROWCOUNT

## LENGUAJE DE CONTROL DE FLUJO

El lenguaje de control de flujo permite controlar bloques de sentencias dándole un sentido programado.

Elementos del Control de flujo:

- IF...ELSE.
- BEGIN...END.
- WHILE.
- BREAK.
- CONTINUE.
- RETURN.
- TRY...CATCH.
- CASE.
- RAISERROR.

### BEGIN... END

Begin- End: agrupa un conjunto de instrucciones que forman parte de una instrucción de control de flujo.

BEGIN

{ sql\_statement | statement\_block }

END



## IF-ELSE

La sentencia IF permite evaluar una o más condiciones, si esta condición resulta verdadera se ejecutara un bloque de sentencias en un BEGIN... END, si resulta falso, opcionalmente con la sentencia ELSE puede ejecutar otro bloque de sentencias contenidas en un BEGIN... END.

IF Boolean\_expression

{ sql\_statement | statement\_block }

[ ELSE

{ sql\_statement | statement\_block } ]

## WHILE

Establece una condición para la ejecución repetida de una instrucción o bloque de instrucciones SQL, las instrucciones se ejecutan repetidamente siempre que la condición especificada sea verdadera. Se puede controlar la ejecución de instrucciones en el bucle WHILE con las palabras clave BREAK y CONTINUE

WHILE Boolean\_expression { sql\_statement | statement\_block | BREAK | CONTINUE }

- **BREAK:** Produce la salida del bucle WHILE más interno, se ejecutan las instrucciones que aparecen después de la palabra clave END, que marca el final del bucle.

- **CONTINUE:** Hace que se reinicie el bucle WHILE y omite las instrucciones que haya después de la palabra clave CONTINUE.
- **EXISTS:** Se usa como condición, el WHILE será verdadero mientras exista valor en la condición.

Ejemplo:

```

DECLARE @contador INT = 0
WHILE (@contador < 5)
BEGIN
    PRINT 'Iniciando vuelta '
        + convert(varchar,@contador);
    SET @contador = @contador + 1
    IF @contador = 2
    CONTINUE
    ELSE
    BEGIN
        PRINT 'Sin saltar'
        IF @contador = 4
        BREAK
    END
END

DECLARE @persona TABLE
(
    Id int,
    Nombre Varchar(20),
    Apellido varchar(20))

DECLARE @id int = 0,
        @cadena varchar(100)

INSERT INTO @persona
SELECT TOP 10 BusinessEntityID,
    FirstName, LastName
FROM Person.Person

WHILE EXISTS (SELECT * FROM @persona)
BEGIN
    SELECT TOP 1 @id = Id,
        @cadena = Nombre + ' ' + Apellido
    FROM @persona
    PRINT @cadena
    DELETE FROM @persona WHERE Id = @id
END

```

Figura 42 Sintaxis de control de flujo por sentencias

## TRY... CATCH

Es un mecanismo de control de errores, se puede incluir un grupo de instrucciones Transact-SQL en un bloque TRY, si se produce un error en el bloque TRY, el control se transfiere a otro grupo de instrucciones que está incluido en un bloque CATCH.

BEGIN TRY

{ sql\_statement | statement\_block }

END TRY

BEGIN CATCH

[ { sql\_statement | statement\_block } ]

END CATCH

Ejemplo:

```
BEGIN TRY
    -- División entre cero.
    SELECT 1/0;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

Figura 43 Sintaxis de validación de errores por try-catch

## RAISERROR

Genera un mensaje de error e inicia el procesamiento de errores de la sesión. RAISERROR puede hacer referencia a un mensaje definido por el usuario almacenado en la vista de catálogo sys.messages o puede generar un mensaje dinámicamente, el mensaje se devuelve como un mensaje de error de servidor a la aplicación que realiza la llamada o a un bloque CATCH asociado de una construcción TRY...CATCH.

RAISERROR ( { msg\_id | msg\_str | @local\_variable } { ,severity ,state }

## THROW

Genera una excepción y transfiere la ejecución a un bloque CATCH de una construcción TRY...CATCH en SQL Server 2017.

THROW [ { error\_number | @local\_variable }, { message | @local\_variable }, { state | @local\_variable } ]

Ejemplo:

```
THROW 51000, 'The record does not exist.', 1;

-----

USE tempdb;
GO
CREATE TABLE dbo.TestRethrow
(
    ID INT PRIMARY KEY
);
BEGIN TRY
    INSERT dbo.TestRethrow(ID) VALUES(1);
    -- Force error 2627, Violation of PRIMARY KEY constraint to be
    raised.
    INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH

    PRINT 'In catch block.';
    THROW;
END CATCH;
```

Figura 44 Sintaxis validación por throw

## CASE

Evalúa una lista de condiciones y devuelve una de las varias expresiones de resultado posibles.

La expresión CASE tiene dos formatos:

- La expresión CASE sencilla compara una expresión con un conjunto de expresiones sencillas para determinar el resultado.
- La expresión CASE buscada evalúa un conjunto de expresiones booleanas para determinar el resultado.

Ambos formatos admiten un argumento ELSE opcional.

Ejemplo:

```
--- CASE Simple:
CASE input_expression
  WHEN when_expression THEN result_expression [...n]
  [ ELSE else_result_expression ]
END

--- CASE Buscada:
CASE
  WHEN Boolean_expression THEN result_expression
  [...n ]
  [ ELSE else_result_expression ]
END
```

Figura 45 Sintaxis de la sentencia case-when

## FUNCIONES DEFINIDAS POR EL USUARIO

Las funciones definidas por el usuario de SQL Server son rutinas que aceptan parámetros, realizan una acción, como un cálculo complejo, y devuelven el resultado de esa acción como un valor. El valor devuelto puede ser un valor escalar único o un conjunto de resultados (tabla).

### FUNCIÓN ESCALAR

- Las funciones escalares definidas por el usuario devuelven un único valor de datos del tipo definido en la cláusula RETURNS.
- El tipo devuelto puede ser de cualquier tipo de datos excepto text, ntext, image, cursory timestamp.

Ejemplo:

```
CREATE FUNCTION dbo.ufnGetInventoryStock(@ProductID int)
RETURNS int
AS
-- Returns the stock level for the product.
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID
        AND p.LocationID = '6';
    IF (@ret IS NULL)
        SET @ret = 0;
    RETURN @ret;
END;
```

Figura 46 Sintaxis para crear funciones con consultas

## FUNCIÓN CON VALOR DE TABLA

- Las funciones con valores de tabla definidas por el usuario devuelven un tipo de datos table.
- La tabla es el conjunto de resultados de una sola instrucción SELECT.

Ejemplo:

```
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
```

Figura 47 Ejemplo para crear funciones con valor de tabla en funciones agregadas

## USO DE FUNCIONES

- Las funciones se usan en una sentencia, en el contexto en donde se espera el tipo de resultado.

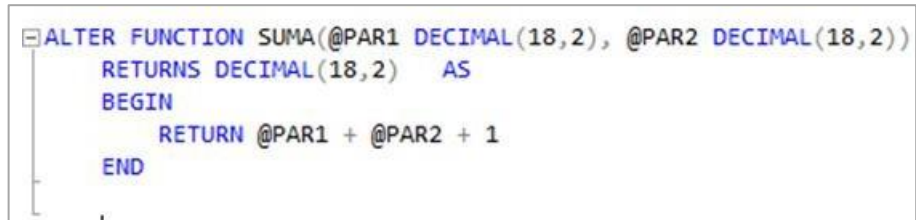
Ejemplo con el comando EXEC:

- EXEC @RESP = dbo.SUMA 4, 5

## MODIFICAR FUNCIONES

Para realizar un cambio en una función, se usa la cláusula ALTER.

Ejemplo:

A screenshot of a SQL code editor showing the ALTER FUNCTION statement for a function named SUMA. The code is as follows:

```
ALTER FUNCTION SUMA(@PAR1 DECIMAL(18,2), @PAR2 DECIMAL(18,2))  
RETURNS DECIMAL(18,2) AS  
BEGIN  
    RETURN @PAR1 + @PAR2 + 1  
END
```

Figura 48 Ejemplo para realizar cambios en funciones

## ELIMINAR UNA FUNCIÓN

- Para eliminar una función se usa la cláusula DROP
- DROP FUNCTION SUMA

## PROCEDIMIENTOS ALMACENADOS

- Un procedimiento almacenado de SQL Server es un grupo de una o varias instrucciones.
- Los procedimientos se asemejan a las construcciones de otros lenguajes de programación, porque pueden:
  - Aceptar parámetros de entrada
  - Contener llamadas a otros procedimientos.
  - Realizar operaciones en la base de datos.



- Devolver un valor de estado.

### **Ventajas de SP**

- Tráfico de red reducido entre el cliente y el servidor.
- Mayor seguridad.
- Reutilización del código.
- Mantenimiento más sencillo.
- Rendimiento mejorado.

### **Crear un SP**

Se muestra la estructura básica para crear un sp:

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.uspGetEmployeesTest2
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS
    SET NOCOUNT ON;
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName
    AND EndDate IS NULL;
GO
```

Figura 49 Ejemplo de creación de procedimiento almacenado

### **Ejecutar un SP**

- A continuación, se muestra una sentencia que invoca la ejecución de un sp existente en la base de datos.
- EXECUTE HumanResources.uspGetEmployeesTest2 N'Ackerman', N'Pilar';

### **Modificar un SP**

Para realizar un cambio en un procedimiento almacenado, se utiliza la cláusula ALTER.

```
ALTER PROCEDURE Purchasing.uspVendorAllInfo
    @Product varchar(25)
AS
    SET NOCOUNT ON;
    SELECT LEFT(v.Name, 25) AS Vendor, LEFT(p.Name, 25) AS 'Pro
    'Rating' = CASE v.CreditRating
        WHEN 1 THEN 'Superior'
        WHEN 2 THEN 'Excellent'
        WHEN 3 THEN 'Above average'
        WHEN 4 THEN 'Average'
        WHEN 5 THEN 'Below average'
        ELSE 'No rating'
    END
    , Availability = CASE v.ActiveFlag
        WHEN 1 THEN 'Yes'
        ELSE 'No'
```

Figura 50 Ejemplo para modificar un procedimiento almacenado con función case

## Eliminar un SP

- Para eliminar un procedimiento almacenado, existente en la base de datos, se usa la cláusula DROP
- La estructura es:
  - DROP PROCEDURE ESQUEMA.NOMBRE\_SP
- Ejemplo:
  - DROP PROCEDURE dbo.uspGetEmployeesTest