MERLOS INFOR

valor tecnológico

# INTRODUCCIÓN

# A

# TRANSACT SQL

**Dto Programación**
**Merlos-Infor, SL**

## 1. Breve historia de SQL

La historia de SQL , que deberíamos pronunciar "ese-cu-ele" y no "siqüel" como se oye a menudo, empieza con la primera versión, que fue desarrollada en los laboratorios de IBM por Andrew Richardson, Donald C. Messerly y Raymond F. Boyce, a comienzos de los años 70, viendo la luz en 1974, y que se basaron en el *"modelo relacional de datos para grandes bancos de datos compartidos"* de Edgar Frank Cood. A esta versión le llamaron SEQUEL, Structured English QUEry Language, y fue diseñada para acceder y manejar los datos de la base de datos System R de IBM. Se implementó en un prototipo llamado SEQUEL-XRM entre 1974 y 1975. Las experimentaciones con ese prototipo condujeron, entre 1976 y 1977, a una revisión del lenguaje (SEQUEL/2), que a partir de ese momento cambió de nombre por motivos legales, convirtiéndose en SQL.
El prototipo (System R), basado en este lenguaje, se adoptó y utilizó internamente en IBM y lo adoptaron algunos de sus clientes elegidos. Gracias al éxito de este sistema, que no estaba todavía comercializado, también otras compañías empezaron a desarrollar sus productos relacionales basados en SQL.

En 1979, Relational Software Inc, primer nombre de Oracle Corporation, , lo introdujo por primera vez en un programa comercial, el Oracle V2 para servidores VAX (en la imagen un VAX 6000)

A partir de 1981, IBM comenzó a entregar sus productos relacionales y en 1983 empezó a vender DB2. En el curso de los años ochenta, numerosas compañías (por ejemplo Oracle y Sybase, sólo por citar algunos) comercializaron productos basados en SQL, que empezó a convertirse en el estándar industrial de hecho por lo que respecta a las bases de datos relacionales.

Fue en 1986, cuando el ANSI (American National Estándar Institute) adoptó SQL como estándar para los lenguajes relacionales como SQL-86 o SQL1, que en 1987 se transformó en estándar ISO.

La necesidad de ampliar los requerimientos del lenguaje a la vez que se experimentaba en bases de datos relacionales, provocó que se fueran realizando diferentes revisiones, que adoptaron nombre de SQL/98 y SQL/92, que fué llamado SQL2.

El hecho de tener un estándar definido por un lenguaje para bases de datos relacionales abrió potencialmente el camino a la intercomunicabilidad entre todos los productos que se basaban en él. Desde el punto de vista práctico, por desgracia las cosas fueron de otro modo. Efectivamente, en general cada productor adoptó e implementó en la propia base de datos sólo el corazón del lenguaje SQL, extendiéndolo de manera individual según la propia visión que cada cual tenga del mundo de las bases de datos.

En la actualidad el SQL es el estándar de la inmensa mayoría de los SGBD comerciales (MySQL, Postgres, DB2, Informix, Microsoft SQL Server, Access, Oracle, SyBase, etc) .
Y, aunque la diversidad de añadidos particulares que incluyen las diferentes versiones comerciales o gratuitas del lenguaje es amplia, el soporte al estándar SQL-92 es general y muy amplio.

Actualmente, está en marcha un proceso de revisión del lenguaje por parte de los comités ANSI e ISO, que debería terminar en la definición de lo que en este momento se conoce como SQL3. Las características principales de esta nueva encarnación de SQL deberían ser su transformación en un lenguaje stand-alone (mientras ahora se usa como lenguaje hospedado en otros lenguajes) y la introducción de nuevos tipos de datos más complejos que permitan, por ejemplo, el tratamiento de datos multimediales.

Las desventajas de SQL son pocas comparadas a las ventajas, y van más por el lado comercial y por lo complejo que puede ser el acceso a los comandos desde la consola.

- La implementación es inconsistente y, usualmente, incompatible entre vendedores. En especial las sintaxis para fechas, caractéres nulos, concatenación, y caracteres especiales.
- Los vendedores tienden a aumentar la incompatibilidad para mantener a sus clientes fieles a su producto
- La facilidad con la que se puede inducir al error, por ejemplo con una mala sintaxis de un comando
- La gramática SQL es demasiado compleja.

En la tabla siguiente vemos una pequeña reseña histórica de las versiones y lanzamientos de MS SQLServer.

| Versiones de SQL Server | | | |
|---|---|---|---|
| **Versión** | **Año** | **Nombre de la versión** | **Code Name** |
| 1.0 (OS/2) | 1989 | SQL Server 1.0 | - |
| 4.21 (NT) | 1993 | SQL Server 4.21 | - |
| 6.0 | 1995 | SQL Server 6.0 | SQL95 |
| 6.5 | 1996 | SQL Server 6.5 | Hydra |
| 7.0 | 1998 | SQL Server 7.0 | Sphinx |
| | 1999 | SQL Server Herramientas OLAP | Platón |
| 8.0 | 2000 | SQL Server 2000 | Shiloh |
| 8.0 | 2003 | SQL Server 2000 para 64 bits | Liberty |
| 9.0 | 2005 | SQL Server 2005 | Yukon |
| 10.0 | 2008 | SQL Server 2008 | Katmai |

## 2. Introducción al Transact SQL

SQL es un lenguaje de consulta para los sistemas de bases de datos relaciónales, pero que no es un lenguaje de programación. Es un conjunto de sentencias, mandatos y funciones estandarizadas pero sin la potencia de un lenguaje de programación.

Transact SQL es el lenguaje de programación que proporciona SQL Server para ampliar SQL con los elementos característicos de los lenguajes de programación: variables, sentencias de control de flujo, bucles .... Transact SQL es el lenguaje de programación que proporciona SQL Server para extender el SQL estándar con otro tipo de instrucciones, para darle las características de un lenguaje de programación.
Para poder seguir esta guía correctamente necesitaremos tener los siguientes elementos:

1. Un servidor SQL Server 2005. Podemos descargar gratuitamente la versión SQL Server Express desde el siguiente enlace.SQL Server 2005 Express.
2. Herramientas cliente de SQL Server por ejemplo Microsoft SQL Server Management Studio

## 3. Programación con Transact SQL

SQL es la herramienta ideal para trabajar con bases de datos, pero para poder realizar una aplicación completa para el manejo de una base de datos relacional, necesitamos usar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales.

Transact SQL nos aporta esta necesidad y con Transact SQL vamos a poder programar las unidades de programa de la base de datos SQL Server, como procedimientos almacenados, funciones, triggers o scripst.

## 4. Primeros pasos con Transact SQL

Para programar en Transact SQL es necesario conocer algunos elementos y conceptos básicos del lenguaje.

- Transact SQL no es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas.
- Soporta dos estilos de comentarios, el de línea simple y de multilínea, para lo cual son empleados ciertos caracteres especiales como son:
    1. `--` Para un comentario de línea simple
    2. `/*. . . */` Para un comentario de varias líneas
- Un literal es un valor fijo de tipo numérico, carácter, cadena o lógico no representado por un identificador (es un valor explícito).
- Una variable es un valor identificado por un nombre (identificador) sobre el que podemos realizar modificaciones. En Transact SQL los identificadores de variables deben comenzar por el carácter @, es decir, el nombre de una variable debe comenzar por @.Para declarar variables en Transact SQL debemos utilizar la palabra clave declare, seguido del identificador y tipo de datos de la variable.

Veamos algunos ejemplos:

```
-- Esto es un comentario de linea simple

/*
Este es un comentario con varias líneas.
Conjunto de Lineas.
```

```
*/

declare @nombrevar varchar(50)
set @nombre = 'valordelavariable'   -- Le doy valor a nombrevar
print @Nombre                       /* Imprime por pantalla el
                                    valor de @nombrevar. */
```

Un script de Transact SQL es un conjunto de sentencias de Transact SQL en formato de texto plano que se ejecutan en un servidor de SQL Server. Puede estar compuesto por varios lotes. Cada lote delimita el alcance de las variables y sentencias dentro del script. Dentro de un mismo script se diferencian los diferentes lotes a través de la instrucción GO.

En algunos casos concretos es necesario

```
-- Este es el primer lote del script
SELECT * FROM COMENTARIOS
GO -- GO es el separador de lotes
-- Este es el segundo lote del script
SELECT getdate()  -- getdate() es una función integrada que devuelve
                  -- la fecha
```

En ocasiones es necesario separar las sentencias en varios lotes, porque Transact SQL no permite la ejecución de ciertos comandos en el mismo lote, si bien normalmente también se utilizan los lotes para realizar separaciones lógicas dentro del script.

**Las Intercalaciones de SQL Server (Collation)**

Está palabra tan enrevesada es la empleada por Microsoft para definir como se ordenan y comparan las cadenas en los diferentes idiomas.

El saber que palabra es igual a otra, o cual va antes que otra (alfabéticamente) puede parecer una cosa sin importancia y bastante trivial, en nuestro caso puede ser, pero en el resto de los idiomas no es así. Por ejemplo el Japonés emplea tres alfabetos diferentes, y tiene unas reglas bastante complejas para determinar que carácter va antes y cual después. Es más, en castellano existen diferentes criterios para ordenar como puede ser que la combinación de letras “ch” se coloque después de la letra c.

Las intercalaciones permiten definir el idioma, así como una serie de criterios adicionales que se aplican a la hora de diferenciar y ordenar diferentes caracteres. En SQL Server 2005 los idiomas soportados son los mismo que se soportan en los sistemas operativos Windows. Aparte del idioma entre los otros criterios que pueden especificarse son:

- Permite diferenciar entre mayúsculas y minúsculas (CS Case Sensitive – CI Case Insensitive).
- Permite diferenciar entre una misma letra acentuada o sin acentuar. (AS Accent Sensitive – AI Accent Insensitive).
- Permite diferenciar entre las sílabas japonesas katakana e hiragana (KS Kana Sensitive – KI Kana Insensitive).

La intercalación puede definirse a diferentes niveles según nos interese en cada caso. Puede definirse a nivel de servidor de BD, a nivel de la BD, a nivel de tabla, a nivel del

campo de texto dentro de la tabla o nivel de la consulta. Al margen del primer caso que es para tener una política general para todas ellas, las otras se aplicarían una u otra según nuestras necesidades:

- Si se aplica a nivel de BD no solamente los datos tendrán en cuenta estos criterios sino los nombres de tablas, campos.. La modificación posterior de la intercalación de la base de datos no tendrá efecto sobre los campos cuya intercalación no sea la predeterminada de la base de datos.
- Si se aplica a en los campos de texto dentro de las tablas lo podríamos hacer o para todos los datos o sólo para algunos.
- Si se aplica en las consultas, ese criterio sólo se aplicaría en esa consulta concreta, puede ser interesante para permitir hacer búsquedas al usuario sin tener que preocuparse de mayúsculas, acentos, etc, inclusive que el mismo lo defina en el momento de hacer la consulta (opciones de la búsqueda).

Cuando se realiza la instalación de Eurowin, se definen las Intercalaciones de la siguiente forma:

- Intercalación a nivel de servidor de datos: El definido en la Instalación del SQL Server
- Intercalación a nivel de base de datos: CI-AI
- Intercalación a nivel de campo:
  - Por defecto los campos de tipo char son: CS-AI
  - Los campos de tipo TEXT son: CI-AI. La razón es porque en dado que en los campos de tipo TEXT no se pueden aplicar funciones de tipo string como por ejemplo: UPPER, LOWER, etc …

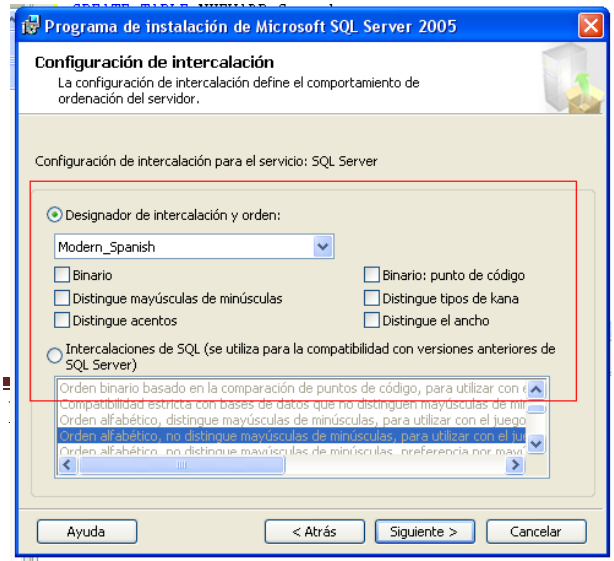
La Ñ y la Ç son considerados caracteres que se encuentran en el grupo de los acentos. Por tanto, si utilizamos la Intercalación CASE SENSITIVE – ACCENT INSENSITIVE y tenemos los códigos de artículos: N0001 y Ñ0001, para SQL Server es el mismo código.

Veamos algunos ejemplos:
1.- A nivel de base de datos:

```
CREATE DATABASE nuevadb
COLLATE Modern_Spanish_CI_AS
```

2.- A nivel de campo:

```
CREATE TABLE NUEVADB.Currela
    (
    Id int NOT NULL PRIMARY KEY ,
    Nombre varchar(50) COLLATE Modern_Spanish_CI_AI NOT NULL ,
    Telefono varchar(10) NULL
                                                    )
```




A nivel de Tabla, lo que realmente sucede es que toma la intercalación por defecto de la base de datos. Si hay un cambio, lo realizaremos a nivel de campo.

En la instalación de Servidor es muy importante fijarse en el apartado de la configuración de la intercalación,

de forma de no dejar activa ninguna opción de intercalación. Esto nos servirá para que en el caso de que creemos una base de datos tipo TEST, se creará con la misma intercalación que las bases de datos que monta Eurowin. Esto nos evitará problemas en el caso de UPDATES combinados entre tablas con diferentes Intercalaciones.

## 5. Tipos de datos en Transact SQL

Cuando definimos una tabla, variable o constante debemos asignar un tipo de dato que indica los posibles valores. El tipo de datos define el formato de almacenamiento, espacio que de disco-memoria que va a ocupar un campo o variable, restricciones y rango de valores validos.
Transact SQL proporciona una variedad predefinida de tipos de datos. Casi todos los tipos de datos manejados por Transact SQL son similares a los soportados por SQL.

**- Tipos de datos numéricos.**

SQL Server dispone de varios tipos de datos númericos. Cuanto mayor sea el número que puedan almacenar mayor será en consecuencia el espacio utilizado para almacenarlo. Como regla general se recomienda usar el tipo de dato mínimo posible. Todos los datos numéricos admiten el valor NULL.

1. **Bit**. Una columna o variable de tipo bit puede almacenar el rango de valores de 1 a 0.
2. **Tinyint**. Una columna o variable de tipo tinyint puede almacenar el rango de valores de 0 a 255.
3. **SmallInt**. Una columna o variable de tipo smallint puede almacenar el rango de valores -32768 a 32767.
4. **Int**. Una columna o variable de tipo int puede almacenar el rango de valores de $-2^{31}$ a $2^{31}-1$ .
5. **BigInt**. Una columna o variable de tipo bigint puede almacenar el rango de valores $-2^{63}$ a $2^{63-}1$ .
6. **Decimal(n,d)**. Una columna de tipo decimal puede almacenar datos números decimales sin redondear. Donde n es la precision (número total del dígitos) y d el número de valores decimales.
7. **Float.** Una columna de datos float puede almacenar un rango de valores de $-1{,}79X\text{-}10^{308}$ a $1{,}79X\text{-}10^{308}$, si la definimos con el valor máxmo de precisión. La precisión puede variar entre 1 y 53.
8. **Real**. Sinónimo de float(24). Puede almacenar el rango de valores $-3{,}4X\text{-}10^{38}$ a $3{,}4X\text{-}10^{38}$.
9. **Money**. Almacena valores númericos monetarios de $-2^{63}$ a $2^{63}-1$, con una precisión de hasta diez milesimas de la unidad monetaria.
10. **SmallMoney.** Almacena valores números monetarios de -214.748,3647 a 214.748,3647, con una precisión de hasta diez milesimas de la unidad monetaria.

Todos los tipos de datos enteros pueden marcarse con la propiedad identity para hacerlos autonuméricos.

```
DECLARE @bit bit,
      @tinyint tinyint,
      @smallint smallint,
      @int int,
      @bigint bigint,
```

```
      @decimal decimal(10,3), -- 10 digitos, 7 enteros y
                              -- 3 decimales
      @real real,
      @double float(53),
      @money money
set @bit = 1
print @bit
set @tinyint = 255
print @tinyint
set @smallint = 32767
print @smallint
set @int = 642325
print @int
set @decimal = 56565.234  -- Punto como separador decimal
print @decimal
set @money = 12.34
print @money
```

**- Tipos de datos de caracter.**

- **Char(n)**. Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo char, siempre se utilizan los n caracteres indicados, incluso si la entrada de datos es inferior. Por ejemplo, si en un char(5), guardamos el valor 'A', se almacena 'A    ', ocupando los cinco bytes.
- **Varchar(n)**.Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo varchar, unicamente se utilizan los caracteres necesarios,Por ejemplo, si en un varchar(255), guardamos el valor 'A', se almacena 'A', ocupando solo un byte bytes.
- **Varchar(max)**. Igual que varchar, pero al declararse como max puede almacenar 231-1 bytes.
- **Nchar(n)**.Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferentes idomas.
- **Nvarchar(n)**.Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferentes idomas.
- **Nvarchar(max)**.Igual que varchar, pero al declararse como max puede almacenar 231-1 bytes.

**- Tipos de datos de fecha.**

- **Datetime**. Almacena fechas con una precisión de milisegundo. Debe usarse para fechas muy específicas.
- **SmallDatetime.** Almacena fechas con una precisión de minuto, por lo que ocupa la mitad de espacio de que el tipo datetime, para tablas que puedan llegar a tener muchos datos es un factor a tener muy en cuenta.

Como avanzadilla sobre los que nos vendrá, en SQL SERVER 2008, aparece el tipo DATE, que a diferencia de DATETIME que ocupa 4 bits, el DATE ocupará 3, y sólo guardará la fecha.

En Eurowin, sobre la versión 2008, los campos de tipo fecha son traducidos al tipo DATE. Con esto se consiguen las siguientes ventajas:

- Rango totalmente ampliado (1/1/0001 – 31/12/9999). Evitamos el problema del 2005 por fuera de rango

- Los campos de tipo DATE ocupan menos bytes que los campos de tipo SMALLDATETIME, por tanto ocupa menos espacio en la base de datos y por tanto se traduce en una optimización a nivel de base de datos
- Eurowin no tiene que hacer un proceso adicional de conversión entre fechas de tipo Datetime a Date y por tanto aumentamos velocidad.

**- Tipos de datos binarios.**

- **Binary.** Se utiliza para almacenar datos binarios de longitud fija, con una longitud máxima de 8000 bytes.
- **Varbinary.** Se utiliza para almacenar datos binarios de longitud variable, con una longitud máxima de 8000 bytes..Es muy similar a binary, salvo que varbinary utiliza menos espacio en disco.
- **Varbinary(max)**.Igual que varbinary, pero puede almacenar 231-1 bytes

## 6. Variables en Transact SQL

**- Declarar variables es Transact SQL**

Una variable es un valor identificado por un nombre (identificador) sobre el que podemos realizar modificaciones.
En Transact SQL la identificación de variables deben comenzar por el caracter @, es decir, el nombre de una variable debe comenzar por @. Para declarar variables en Transact SQL debemos utilizar la palabra clave declare, seguido del identificador y tipo de datos de la variable.

```
declare @nombre varchar(50) -- la variable se llama nombre
                            -- y es de tipo varchar de 50 caracteres
```

**- Asignar variables en Transact SQL**

En Transact SQL podemos asignar valores a una variable de varias formas:
- A través de la instrucción SET.
- Utilizando una sentencia SELECT.
- Realizando un FETCH de un cursor.

El siguiente ejemplo muestra como asignar una variable utilizando la instrucción SET.

```
DECLARE @nombre VARCHAR(100)
-- La consulta debe devolver un único registro
SET @nombre =  (SELECT nombre
            FROM CLIENTES
            WHERE codigo ='43000124' )
PRINT @nombre
```

El siguiente ejemplo muestra como asignar variables utilizando una sentencia SELECT.

```
DECLARE @nombre VARCHAR(100),
      @direccion VARCHAR(100),
      @poblacion VARCHAR(100)
```

```
SELECT  @nombre=nombre ,
      @direccion=direccion,
      @poblacion=poblacion
FROM CLIENTES
WHERE codigo = '430000125'

PRINT @nombre
PRINT @direccion
PRINT @poblacion
```

Un punto a tener en cuenta cuando asignamos variables de este modo, es que si la consulta SELECT devuelve más de un registro, las variables quedarán asignadas con los valores de la última fila devuelta.

También podemos usar la sentencia SELECT inicializando las variables directamente.

```
DECLARE @nombre VARCHAR(100),
      @direccion VARCHAR(100),
      @poblacion VARCHAR(100)

SELECT  @nombre='Julián Besteiros' ,
      @direccion='Mora d''Ebre, 32',
      @poblacion='Barcelona'


PRINT @nombre
PRINT @direccion
PRINT @poblacion
```

Por último, veamos como asignar variables a través de un cursor.

```
DECLARE @nombre VARCHAR(100),
      @direccion VARCHAR(100),
      @poblacion VARCHAR(100)

DECLARE CurClientes CURSOR
FOR
SELECT nombre , direccion, poblacion
FROM CLIENTES

OPEN CurClientes
FETCH CurClientes INTO @nombre, @direccion, @poblacion

WHILE (@@FETCH_STATUS = 0)
BEGIN
      PRINT @nombre
      PRINT @direccion
      PRINT @poblacion
      FETCH CurClientes INTO @nombre, @direccion, @poblacion
END

CLOSE CurClientes
DEALLOCATE CurClientes
```

Veremos los cursores con más detalle más adelante en este tutorial.

## 7. Operadores en Transact SQL

La siguiente tabla ilustra los operadores de Transact SQL .

| Tipo de operador | Operadores |
|---|---|
| Operador de asignación | = |
| Operadores aritméticos | + (suma)<br>- (resta)<br>* (multiplicación)<br>/ (división)<br>** (exponente)<br>% (modulo) |
| Operadores relacionales o de comparación | = (igual a)<br><> (distinto de)<br>!= (distinto de)<br>< (menor que)<br>> (mayor que)<br>>= (mayor o igual a)<br><= (menor o igual a)<br>!> (no mayor a)<br>!< (no menor a) |
| Operadores lógicos | AND (y lógico)<br>NOT (negacion)<br>OR (o lógico)<br>& (AND a nivel de bit)<br>\| (OR a nivel de bit)<br>^ (OR exclusivo a nivel de bit) |
| Operador de concatenación | + |
| Otros | ALL (Devuelve TRUE si el conjunto completo de comparaciones es TRUE)<br>ANY(Devuelve TRUE si cualquier elemento del conjunto de comparaciones es TRUE)<br>BETWEEN (Devuelve TRUE si el operando está dentro del intervalo)<br>EXISTS (TRUE si una subconsulta contiene filas)<br>IN (TRUE si el operando está en la lista)<br>LIKE (TRUE si el operando coincide con un patron)<br>NOT (Invierte el valor de cualquier operador booleano)<br>SOME(Devuelve TRUE si alguna de las comparaciones de un conjunto es TRUE) |

## 8. Estructuras de control en Transact SQL

**Estructura condicional IF**

La estuctura condicional IF permite evaluar una expresion booleana (resultado SI - NO), y ejecutar las operaciones contenidas en el bloque formado por BEGIN END.

```
IF (<expresion>)
```

```
  BEGIN
      ...
  END
ELSE IF (<expresion>)
  BEGIN
      ...
  END
ELSE
  BEGIN
      ...
  END
```

Ejemplo de la estructura condicional IF.

```
  DECLARE @Web varchar(100),
            @diminutivo varchar(3)

    SET @diminutivo = 'EW'

    IF  @diminutivo = 'EW'
        BEGIN
            PRINT 'www.eurowin.com'
        END
    ELSE
      BEGIN
            PRINT 'Otro software '
      END
```

La estructura IF admite el uso de subconsultas:

```
DECLARE @codPais varchar(3),
      @nomPais varchar(70)
set @codPais = '034'
set @nomPais = 'Espanya'
IF EXISTS(SELECT * FROM PAISES
          WHERE CODIGO = @codPais)
  BEGIN
      UPDATE PAISES
      SET NOMBRE = @nomPais
      WHERE CODIGO = @codPais
  END

ELSE
  BEGIN
      INSERT INTO PAISES
      (CODIGO , NOMBRE) VALUES
      (@codPais, @nomPais)
  END
```

**Estructura condicional CASE**

La estructura condicional CASE permite evaluar una expresión y devolver un valor u otro.
La sintaxis general de case es:

```
CASE <expresion>
       WHEN <valor_expresion> THEN <valor_devuelto>
       WHEN <valor_expresion> THEN <valor_devuelto>
       ELSE <valor_devuelto>  -- Valor por defecto
```

```
END
```

Ejemplo de CASE.

```
DECLARE @Soft varchar(100),
        @diminutivo varchar(2)
SET @diminutivo = 'EW'
SET @Soft = (CASE @diminutivo
                WHEN 'EW' THEN 'www.eurowin.com'
                WHEN 'TW' THEN 'www.eurowintuweb.com'
                ELSE 'www.millorsoft.es'
             END)
PRINT @Soft
```

Otro aspecto muy interesante de CASE es que permite el uso de subconsultas.

```
DECLARE @Web varchar(100),
      @diminutivo varchar(2)
SET @diminutivo = 'EW'
SET @Web = (CASE
                WHEN @diminutivo = 'EW' THEN (SELECT  web
                                              FROM WEBS
                                              WHERE id=1)
                WHEN @diminutivo = 'TW' THEN (SELECT  web
                                              FROM WEBS
                                              WHERE id=2)
                ELSE 'www.eurowin.com'
             END)
PRINT @Web
```

**Bucle WHILE**

El bucle WHILE se repite mientras expresion se evalue como verdadero.
Es el único tipo de bucle del que dispone Transact SQL.

```
WHILE <expresion>
BEGIN
   ...
END
```

Un ejemplo del bucle WHILE.

```
DECLARE @contador int
SET @contador = 0
WHILE (@contador < 100)
BEGIN
  SET @contador = @contador + 1
  PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
END
```

Podemos pasar a la siguiente iteración del bucle utilizando CONTINUE.

```
DECLARE @contador int
  SET @contador = 0
  WHILE (@contador < 100)
  BEGIN
    SET @contador = @contador + 1
```

```
    IF (@contador % 2 = 0)
              CONTINUE
    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
  END
  El bucle se dejará de repetir con la instrucción BREAK.

  DECLARE @contador int
  SET @contador = 0
  WHILE (1 = 1)
  BEGIN
    SET @contador = @contador + 1
    IF (@contador % 50 = 0)
              BREAK
    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
  END
```

También podemos utilizar el bucle WHILE conjuntamente con subconsultas.

```
    DECLARE @coRecibo int
          WHILE  EXISTS (SELECT *
                 FROM RECIBOS
                 WHERE PENDIENTE = 'S')-- Ojo, la subconsulta se
                                       -- ejecuta una vez por
                                       -- cada iteración del bucle!
          BEGIN
                SET @coRecibo = (SELECT TOP 1 CO_RECIBO
                      FROM RECIBOS WHERE PENDIENTE = 'S')
                UPDATE RECIBOS
                SET PENDIENTE = 'N'
                WHERE CO_RECIBO = @coRecibo
          END
```

### Estructura GOTO

La sentencia GOTO nos permite desviar el flujo de ejecución hacia una etiqueta.

```
  DECLARE @divisor int,
          @dividendo int,
          @resultado int
    SET @dividendo = 100
    SET @divisor = 0
    SET @resultado = @dividendo/@divisor

    IF @@ERROR > 0
        GOTO error

    PRINT 'No hay error'
    RETURN
error:
    PRINT 'Se ha producido una division por cero'
```

## 9. Consultar datos en Transact SQL

### La sentencia SELECT

La sentencia SELECT nos permite consultar los datos almacenados en una tabla de la base de datos.
El formato de la sentencia select es:

```
SELECT [ALL | DISTINCT ][ TOP expression [ PERCENT ] [ WITH TIES ] ]
            <nombre_campos>
FROM <nombre_tabla>
[ INNER | LEFT [OUTER]| RIGHT [OUTER] | CROSS]
[JOIN ] <nombre_tabla> ON <condicion_join>[ AND|OR <condicion>]
[WHERE <condicion> [ AND|OR <condicion>]]
[GROUP BY <nombre_campos>]
[HAVING <condicion>[ AND|OR <condicion>]]
[ORDER BY <nombre_campo> [ASC | DESC]
```

El siguiente ejemplo muestra una consulta sencilla que obtiene el código y el nomnbre de la "familia" de una tabla llamada familias (representaría familias de productos por ejemplo).

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
```

El uso del asterisco indica que queremos que la consulta devuelva todos los campos que existen en la tabla.

```
SELECT * FROM [2008V8].[dbo].FAMILIAS
```

Ahora vamos a realizar una consulta obteniendo además de los datos de familias, las subfamilias asociadas a las familias y los productos.

```
SELECT *
FROM [2008V8].[dbo].FAMILIAS
INNER JOIN [2008V8].[dbo].SUBFAM
      ON SUBFAM.FAMILIA = FAMILIA.CODIGO
INNER JOIN [2008V8].[dbo].ARTICULO
      ON ARTICULO.FAMILIA = FAMILIA.CODIGO
```

La combinación se realiza a través de la clausula INNER JOIN, que es una clasula exclusiva, es decir las familias que no tengan subfamilia y los productos que no tengan familia, no se devolveran.

Si queremos realizar la consulta para que no sea exclusiva, tenemos que utilizar LEFT JOIN.

```
SELECT *
FROM [2008V8].[dbo].FAMILIAS
LEFT JOIN [2008V8].[dbo].SUBFAM
      ON SUBFAM.FAMILIA = FAMILIA.CODIGO
LEFT JOIN [2008V8].[dbo].ARTICULO
      ON ARTICULO.FAMILIA = FAMILIA.CODIGO
```

Los registros que no tengan datos relacionados en una consulta LEFT JOIN devolverán en valor NULL en los campos que correspondan a las tablas en las que no tienen dato.

También podemos forzar un producto cartesiano (todos con todos) a través de CROSS JOIN o FULL JOIN (a partir de SQL 2000)

```
SELECT *
FROM [2008V8].[dbo].FAMILIAS
CROSS JOIN [2008V8].[dbo].SUBFAM

SELECT *
FROM [2008V8].[dbo].FAMILIAS
FULL JOIN [2008V8].[dbo].SUBFAM
```

**La cláusula WHERE**

La cláusula WHERE es la instrucción que nos permite filtrar el resultado de una sentencia SELECT.

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
WHERE CODIGO='001'
```

Por supuesto, podemos especificar varias condiciones para el WHERE:

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
WHERE CODIGO='001' OR CODIGO='002'
```

Podemos agrupar varias valores para una condicion en la clausula IN:

```
SELECT *
FROM [2008V8].[dbo].FAMILIAS
WHERE CODIGO IN ('001','002')
```

La clausula WHERE se puede utilizar conjuntamente con INNER JOIN, LEFT JOIN ...

```
SELECT  FAMILIAS.CODIGO, FAMILIAS.NOMBRE
FROM [2008V8].[dbo].FAMILIAS
INNER JOIN [2008V8].[dbo].SUBFAM
ON SUBFAM.FAMILIA = FAMILIA.CODIGO
WHERE FAMILIAS.CODIGO IN ('001','002','099')
```

Siempre que incluyamos un valor alfanumérico para un campo en la condición WHERE este debe ir entre comillas simples:

```
WHERE CODIGO='001'
```

Para consultar campos alfanuméricos, es decir, campos de texto podemos utilizar el operador LIKE conjuntamente con comodines.

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
WHERE NOMBRE LIKE 'REFE%'
```

Los comodines que podemos utilizar en son los siguientes:

- '%' representa cualquier cadena de texto de cero a cualquier longitud alfanumérica

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
```

```
WHERE NOMBRE LIKE '%REFRE%'
```

- '_' representa un caracter.

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
WHERE NOMBRE LIKE 'REFRESCO_'
```

- [a-d] representa cualquier caracter del intervalo a-d.

```
SELECT CODIGO, NOMBRE
FROM FAMILIAS
WHERE NOMBRE LIKE '%[T-Z]%'
```

- [abcd] representa cualquier caracter del grupo abcd.

```
SELECT CODIGO, NOMBRE
FROM FAMILIAS
WHERE NOMBRE LIKE '[ENS]%'
```

- [^a-d] representa cualquier caracter diferente del intervalo a-d.

```
SELECT CODIGO, NOMBRE
FROM FAMILIAS
WHERE NOMBRE LIKE '[^A-Z]%'
```

- [^abcd] representa cualquier caracter distinto del grupo abcd.

```
SELECT CODIGO, NOMBRE
FROM FAMILIAS
WHERE NOMBRE LIKE '[^ANS]%'
```

También podemos obtener los valores distintos utilizando DISTINCT.

```
SELECT DISTINCT NOMBRE
FROM [2008V8].[dbo].FAMILIAS
```

Podemos limitar el número de registros que devuelve la consulta a través de la clausula TOP. La clausula TOP admite como parámetros un valor numérico entero o un porcentaje (sólo a partir de la version 2005)

```
SELECT TOP 10 * --Devuelve 10 registros
FROM [2008V8].[dbo].FAMILIAS

SELECT TOP 50 PERCENT  -- Devuelve el 50% de los registros
CODIGO, NOMBRE, TCP
FROM [2008V8].[dbo].FAMILIAS
```

La clausula TOP se puede combinar con WITH TIES en consultas agregadas. Este punto lo veremos propiamente en las consultas agregadas.

**La cláusula ORDER BY**

Podemos especificar el orden en el que serán devueltos los datos a través de la cláusula ORDER BY.

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
ORDER BY CODIGO
```

También podemos indicar el índice del campo en la lista de selección en lugar de su nombre :

```
SELECT CODIGO, NOMBRE
FROM [2008V8].[dbo].FAMILIAS
ORDER BY 2 -- Ordena por NOMBRE
```

**La cláusula GROUP BY**

La clausula GROUP BY combina los registros devueltos por una consulta SELECT obteniendo uno o varios valores agregados(suma, valor mínimo y máximo ...).
Para cada registro se puede crear un valor agregado si se incluye una función SQL agregada, como por ejemplo Sum o Count, en la instrucción SELECT. Su sintaxis es:

```
SELECT [ALL | DISTINCT ] [TOP <n> [WITH TIES]]
             <nombre_campo> [{,<nombre_campo>}]
             [{,<funcion_agregado>}]
FROM <nombre_tabla>|<nombre_vista>
        [{,<nombre_tabla>|<nombre_vista>}]
[WHERE <condicion> [{ AND|OR <condicion>}]]
[GROUP BY <nombre_campo> [{,<nombre_campo >}]]
[HAVING <condicion>[{ AND|OR <condicion>}]]
[ORDER BY <nombre_campo>|<indice_campo> [ASC | DESC]
       [{,<nombre_campo>|<indice_campo> [ASC | DESC ]}]]
```

Los valores Null en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores Null no se evalúan en ninguna de las funciones SQL agregadas.
Todos los campos de la lista de campos de SELECT deben incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada.

El siguiente ejemplo realiza una "cuenta" de los datos que hay en la tabla ARTICULOS

```
SELECT COUNT(*) FROM [2009RD].[dbo].ARTICULO
```

En este otro se muestra la suma del IMPORTE de cada uno de los productos que componen un pedido, para calcular el total del pedido agrupado por el numero de pedido.

```
SELECT EMPRESA, LETRA, NUMERO, SUM(IMPORTE)AS IMPORTE
FROM D_PEDIVE
GROUP BY EMPRESA, LETRA, NUMERO
```

Sql Server nos obliga a que en la agrupación aparezcan todos los campos no agrupados que hay en la consulta.

En el siguiente ejemplo veremos cómo añadir un campo más a la consulta sin la necesidad de que se agrupe.

```
SELECT EMPRESA, CUENTA, DATEPART(QQ,FECHA) AS TRIMESTRE, MAX(NUMERO)
AS CAMPO2,
SUM(DEBE)AS TOT_DEBE, SUM(HABER) AS TOT_HABER
```

```
INTO #TMP_TB2 FROM ASIENTOS
GROUP BY DATEPART(qq,FECHA),EMPRESA, CUENTA
-- Modifico la tabla temporal
ALTER TABLE #TMP_TB2 ALTER COLUMN CAMPO2 DECIMAL(15,4)
-- actualizo la tabla temporal para calcular el saldo
UPDATE #TMP_TB2 SET CAMPO2=(TOT_DEBE)-(TOT_HABER)

SELECT * FROM #TMP_TB2
```

Siempre que incluyamos una clausula WHERE en una consulta agregada esta se aplica antes de calcular el valor agregado. Es decir, si sumamos el valor de las ventas por producto, la suma se calcula después de haber aplicado el filtro impuesto por la clausula WHERE.

```
SELECT EMPRESA, LETRA, NUMERO, SUM(IMPORTE)AS IMPORTE
FROM D_PEDIVE
WHERE EMPRESA='01'
GROUP BY EMPRESA, LETRA, NUMERO
```

**La cláusula HAVING**

Es posible que necesitemos calcular un agregado, pero que no necesitemos obtener todos los datos, solo los que cumplan una condición del agregado. Por ejemplo, podemos calcular el valor de las ventas por producto, pero que solo queramos ver los datos de los pedidos que el importe sea mayor de 150. En estos casos se debe usar la clausula HAVING.

Una vez que GROUP BY ha combinado los registros, HAVING muestra cualquier registro agrupado por la cláusula GROUP BY que satisfaga las condiciones de la cláusula HAVING. Se utiliza la cláusula WHERE para excluir aquellas filas que no desea agrupar, y la cláusula HAVING para filtrar los registros una vez agrupados.

```
SELECT EMPRESA, LETRA, NUMERO, SUM(IMPORTE)AS IMPORTE2
FROM D_PEDIVE
WHERE EMPRESA='01'
GROUP BY EMPRESA, LETRA, NUMERO
HAVING SUM(IMPORTE) >150
```

**Funciones agregadas.**

Transact SQL pone a nuestra disposición múltiples funciones agregadas, las más comunes usaremos son:

**Count**

Calcula el número de registros devueltos por una consulta. Su sintaxis es la siguiente:

```
COUNT(<expr>)
```

En donde expr contiene el nombre del campo que desea contar. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL). Puede contar cualquier tipo de datos incluso texto.

Count simplemente cuenta el número de registros sin tener en cuenta qué valores se almacenan en los registros. La función Count no cuenta los registros que tienen campos null a menos que expr sea el carácter comodín asterisco (*).

```
SELECT COUNT(*) FROM D_PEDIVE
```

**Max, Min**

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo especifico de una consulta. Su sintaxis es:

```
MIN(<expr>)
MAX(<expr>)
```

En donde expr es el campo sobre el que se desea realizar el cálculo.

```
SELECT ALMACEN ,ARTICULO, MAX(FECHA), FINAL
FROM STOCKS
WHERE ALMACEN='01'
GROUP BY ALMACEN,ARTICULO, FINAL
```

**Sum**

Devuelve la suma del conjunto de valores contenido en un campo especifico de una consulta. Su sintaxis es:

```
SUM(<expr>)
```

En donde expr respresenta el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos.

```
SELECT EMPRESA, LETRA, NUMERO, SUM(IMPORTE)AS IMPORTE
FROM D_PEDIVE
GROUP BY EMPRESA, LETRA, NUMERO
```

Uso de Select TOP con consultas agregadas.

Podemos utilizar SELECT TOP con consultas agregadas como con cualquier otra instruccion Transact SQL.
En estos casos, la clausula TOP se aplica después de calcular el agregado, devolviendo las N filas indicadas.
Sin embargo, puede darse el caso, en consultas de rankings u otros cálculos, que la fila N+1 agregado idéntico al de la fila N. El uso de TOP N discriminaría el registro N+1. Para evitar este comportamiento, y que la consulta devuelva también el registro N+1utilizamos la clausula WITH TIES.

```
SELECT TOP 5 WITH TIES EMPRESA, LETRA, NUMERO, SUM(IMPORTE)AS IMPORTE
FROM D_PEDIVE
GROUP BY EMPRESA, LETRA, NUMERO
ORDER BY SUM(IMPORTE)DESC
```

En el caso de que el registro 5 y 6 sean iguales en el valor de `SUM(IMPORTE)`, la consulta devolvera 6 registros, siempre y cuando sea mayor o igual a 6 registros.

**Operaciones con conjuntos.**

SQL Server 2005 permite tres tipos de operaciones con conjuntos:
- UNION, disponible en todas las versiones de SQL Server.

- EXCEPT, nuevo en SQL Server 2005.
- INTERSECT, nuevo en SQL Server 2005.

Para utilizar operaciones de conjuntos debemos cumplir una serie de normas.

- Las consultas a unir deben tener el mismo número campos, y además los campos deben ser del mismo tipo.
- Sólo puede haber una única clausula ORDER BY al final de la sentencia SELECT.

**UNION**

UNION devuelve la suma de dos o más conjuntos de resultados. El conjunto obtenido como resultado de UNION tiene la misma estructura que los conjuntos originales
El siguiente ejemplo muestra el uso de UNION

```
SELECT CODIGO, NOMBRE, DIRECCION
FROM CLIENTES
UNION
SELECT CODIGO, NOMBRE, DIRECCION
FROM PROVEED
```

El siguente ejemplo, en el campo dirección del resultado nos rellenará el campo CIF en los resultados correspondientes a PROVEED

```
SELECT CODIGO, NOMBRE, DIRECCION
FROM CLIENTES
UNION
SELECT CODIGO, NOMBRE, CIF
FROM PROVEED
```

El siguente ejemplo, devolverá un error, dado que als consultas con UNION el numero de campos a consultar en cada tabla debe de ser el mismo.

```
SELECT CODIGO, NOMBRE, DIRECCION
FROM CLIENTEs
UNION
SELECT CODIGO, NOMBRE, DIRECCION, CIF
FROM PROVEED
```

Cuando realizamos una consulta con UNION internamente se realiza una operacion DISTINCT sobre el conjunto de resultados final. Si queremos obtener todos los valores debemos utiliza UNION ALL.

```
SELECT CODIGO, NOMBRE, DIRECCION
FROM CLIENTES
UNION ALL
SELECT CODIGO, NOMBRE, DIRECCION
FROM PROVEED
```

**EXCEPT**

EXCEPT devuelve la diferencia (resta) de dos o más conjuntos de resultados. El conjunto obtenido como resultado de EXCEPT tiene la misma estructura que los conjuntos originales.

El siguiente ejemplo muestra el uso de EXCEPT

```
SELECT CODIGO, NOMBRE
FROM CUENTAS
EXCEPT
SELECT CODIGO, NOMBRE
FROM CLIENTES
```

Este ejemplo devolverá la consulta de la tabla de CUENTAS sin las cuentas que se encuentren en la tabla CLIENTES

El uso de EXCEPT, como norma general, es mucho más rápido que utilizar condiciones NOT IN o EXISTS en la clausula WHERE.

**INTERSECT**

Devuelve la intersección entre dos o más conjuntos de resultados en uno. El conjunto obtenido como resultado de INTERSECT tiene la misma estructura que los conjuntos originales.

El siguiente ejemplo muestra el uso de INTERSECT

```
SELECT CODIGO, NOMBRE
FROM CUENTAS
INTERSECT
SELECT CODIGO, NOMBRE
FROM CLIENTES
```

Este ejemplo devolverá la consulta de la tabla de CUENTAS solamente con las cuentas que se encuentren en la tabla CLIENTES

Analicemos el siguiente ejemplo:

```
SELECT CODIGO, NOMBRE, cif
FROM CUENTAS
intersect
SELECT CODIGO, NOMBRE, direccion
FROM CLIENTES
```

## Uso de Select con la cláusula INTO

Si en una consulta con la instrucción SELECT le añadimos la cláusula INTO, estamos indicando que queremos externalizar la consulta. Esta externalización la podemos llevar a otra tabla de forma fija hasta que sea borrada (DROP) o a tablas temporales. Este método debería de convertirse en nuestro regla principal de SQLServer, antes de realizar cualquier operación en los datos.
Vamos a ver varios ejemplos:

Creación de tabla 'fija'

```
SELECT * INTO [TEST].dbo.cs_familias from [2009WB].dbo.familias
```

Tabla temporal únicamente para la ésta sesión:

```
SELECT * INTO #cs_familias2 from [2009WB].dbo.familias
```

```
select * from #cs_familias2
```

Tabla temporal únicamente para la todas las sesiones:

```
SELECT * INTO ##cs_familias3 from [2009WB].dbo.familias
select * from ##cs_familias3
```

## 9.- Inserción de filas.

**Inserción Individual**

Para realizar la insercción individual de filas SQL posee la instrucción INSERT INTO.La insercción individual de filas es la que más comúnmente utilizaremos. Su sintaxis es la siguiente:

```
INSERT INTO <nombre_tabla>
[(<campo1>[,<campo2>,...])]
values
(<valor1>,<valor2>,...);
```

El siguiente ejemplo muestra la inserción de un registro en la tabla CUENTAS.

```
INSERT INTO CUENTAS
(CODIGO, NOMBRE, DIVISA, CIF, VARIA, SECUNDARIA, DESCRIP, VISTA)
values
('62900002','Mantenimientos Sage Eurowin','000','','','','',1)
```

**Insertción múltiple de filas**.

También es posible insertar en una tabla el resultado de una consulta SELECT. De este modo se insertarán tantas filas como haya devuelto la consulta SELECT.

El siguiente ejemplo muestra la inserción múltiple de filas.

```
INSERT INTO CUENTAS
(CODIGO, NOMBRE, DIVISA, CIF, VARIA, SECUNDARIA, DESCRIP, VISTA)
SELECT
CODIGO, NOMBRE, IDIOMA, CIF, '' AS VARIA,'' AS SECUNDARIA,'' AS
DESCRIP, 1 AS VISTA
FROM CLIENTES
```

**Inserción de valores por defecto.**
También podemos forzar a que la insercción se realice con los datos por defecto establecidos para la tabla (o null si no tienen valores por defecto).

```
INSERT INTO NombreTabla DEFAULT VALUES
```

En SQL Sever podemos marcar un campo de una tabla como autonumérico (identity), cuando insertamos un registro en dicha tabla el valor del campo se genera automáticamente. Para recuperar el valor generado disponemos de varios métodos:

- Utilizar la funcion @@identity, que devuelve el último valor identidad insertado por la transacción. No especifica el ámbito.

El uso de @@Identity no siempre es válido, ya que al devolver el úlitmo valor identidad insertado por la transacción, no nos garantiza que el valor haya sido insertado en la tabla que nos interesa (por ejemplo la tabla podría tener un trigger que insertara datos en otra tabla con campos identidad).

- En este tipo de escenarios debemos utilizar la función, SCOPE_IDENTITY(). Devuelve el valor insertado en el ámbito actual.

Veamos un ejemplo para entender esto:

```
USE test
GO
CREATE TABLE Tabla1 ( id  int IDENTITY(1,1)PRIMARY KEY,
                            Nombre varchar(20) NOT NULL);

INSERT Tabla1  (Nombre) VALUES ('Anna')
INSERT Tabla1  (Nombre) VALUES ('Albert')
INSERT Tabla1  (nombre) VALUES ('Montse')

CREATE TABLE Tabla2 ( id  int IDENTITY(100,5)PRIMARY KEY,
                           Actividad varchar(20) NULL);

INSERT Tabla2 (Actividad) VALUES ('Lecher@')
INSERT Tabla2 (Actividad) VALUES ('Panader@')
INSERT Tabla2 (Actividad) VALUES ('Albañil')
------------- FIN BLOQUE 1 ------------

/*Creamos un trigger que inserte un registro
en la tabla2 al insertar uno en la tabla1 */

CREATE TRIGGER T1_Inserta
ON Tabla1
FOR INSERT AS
   BEGIN
      INSERT Tabla2 VALUES ('')
   END

--------- FIN BLOQUE 2 -------------
/* Ahora provocamos que se dispare el trigger
al inserta un nuevo nombre en Tabla1. */

INSERT Tabla1 (nombre) VALUES ('Ricardo')

Select * from tabla1
GO
Select * from tabla2
GO

SELECT SCOPE_IDENTITY() AS [SCOPE_IDENTITY]
GO
SELECT   @@IDENTITY AS [@@IDENTITY]
GO
```

Analicemos los valores devueltos por SCOPE_IDENTITY y por @@IDENTITY.

**Clausula OUTPUT**

A partir de la version de SQL Server 2005 disponemos de la clausula OUTPUT para recuperar los valores que hemos insertado. Al igual que en un trigger disponemos de las tablas lógicas INSERTED y DELETED.

Las columnas con prefijo DELETED reflejan el valor antes de que se complete la instrucción UPDATE o DELETE. Es decir, son una copia de los datos "antes" del cambio.

DELETED no se puede utilizar con la cláusula OUTPUT en la instrucción INSERT.

Las columnas con prefijo INSERTED reflejan el valor después de que se complete la instrucción UPDATE o INSERT, pero antes de que se ejecuten los desencadenadores. Es decir, son una copia de los datos "despues" del cambio.

INSERTED no se puede utilizar con la cláusula OUTPUT en la instrucción DELETE.

```
DECLARE @FILAS_INSERTADAS TABLE
( tb_codigo char(2),
  tb_nombre char(30),
  tb_cont int
)

INSERT INTO ACTIVI
(CODIGO, NOMBRE)
OUTPUT INSERTED.* INTO @FILAS_INSERTADAS
VALUES ('56' , 'INSTALADOR DE GAS' )

SELECT * FROM @FILAS_INSERTADAS

GO
```

# 10.- Actualizar datos en Transact SQL

La sentencia UPDATE permite la actualización de uno o varios registros de una única tabla. La sintaxis de la sentencia UPDATE en SQL Server es la siguiente

```
UPDATE <nombre_tabla>
SET  <campo1> = <valor1>
     {[,<campo2> = <valor2>,...,<campoN> = <valorN>]}
[ WHERE <condicion>];
```

El siguiente ejemplo muestra el uso de UPDATE.

```
UPDATE CLIENTES
SET
NOMBRE = 'JIMENES LOSANTOS, FEDERICO'
WHERE CLIENTE = '43000010'
```

Un aspecto a tener en cuenta, es que SQL graba los cambios inmediatamente sin necesidad de hacer COMMIT. Por supuesto podemos gestionar nosotros las transacciones pero es algo que hay que hacer de forma explícita con la instrucción BEGIN TRAN y que se verá en un curso más avanzado de T-SQL.
Update INNER JOIN

En ocasiones queremos actaualizar los datos de una tabla con los datos de otra. Habitualmente, usamos subconsultas para este proposito, pero Transact SQL permite la utilización de la sentencia UPDATE INNER JOIN.

```
UPDATE D_ALBVEN
SET
FAMILIA=ARTICULO.FAMILIA
FROM D_ALBVEN
INNER JOIN ARTICULO
      ON ARTICULO.CODIGO=D_ALBVEN.ARTICULO
```

Otra forma de realizar una actualización es el uso de comparativa de registros en tablas, tal como indica el ejemplo siguiente.

```
UPDATE D_ALBVEN
SET
FAMILIA= B.FAMILIA
FROM D_ALBVEN A, ARTICULO B
WHERE B.CODIGO = A.ARTICULO
```

**Clausula OUTPUT**

El uso de ésta clausula es idéntico al que hemos visto en el apartado anterior. Guardará los datos antes de actualizar o borrar.

```
DECLARE @FILAS_ACTUALIZADAS TABLE
(  codigo char(2), nombre char(30), vista bit)

UPDATE ACTIVI
SET
      NOMBRE='INSTALADOR DE GAS'
OUTPUT DELETED. * INTO @FILAS_ACTUALIZADAS
WHERE CODIGO='56'

SELECT * FROM @FILAS_ACTUALIZADAS

GO
```

## 11.- Funciones integradas de Transact SQL

SQL Server pone a nuestra disposición multitud de funciones predefinidas que proporcionan un amplio abanico de posibilidades. Veremos algunas de las más frecuentes. A través del siguiente enlace podemos acceder al listado completo.
http://technet.microsoft.com/es-es/library/ms174318.aspx

**Cast y Convert**

Convierten una expresión de un tipo de datos en otro de forma explícita. CAST y CONVERT proporcionan funciones similares.

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Donde:
- data_type, es el tipo de destino al que queremos convertir la expresion
- expresion, la expresion que queremos convertir

- style, parametro opcional que especifica el formato que tiene expresion. Por ejemplo, si queremos convertir un varchar a datetime, aqui debemos especificar el formato de la fecha (el tipo varchar).

Veamos unos ejemplos:

```
DECLARE @fecha varchar(20)
-- Convertimos un valor varchar a datetime
-- El 103 indica el formato en el que esta escrita la fecha
-- 103 => dd/mm/aaaa
SET @fecha = CONVERT(datetime, '19/03/2008',103)

SELECT @fecha
```

nos da como resultado: Mar 19 2008 12:00AM

```
DECLARE @fecha datetime,
      @fechaFormateada varchar(20)
-- Convertimos ahora una fecha a varchar y la formateamos
-- 3 => dd/mm/aa
SET @fecha = GETDATE()
SET @fechaFormateada = CONVERT(varchar(20), @fecha, 3)

SELECT @fechaFormateada
```

nos da como resultado: 01/10/09

```
DECLARE @fecha datetime,
      @fechaFormateada varchar(20)
SET @fecha = GETDATE()
SET @fechaFormateada = CONVERT(varchar(20), @fecha, 104)

SELECT @fechaFormateada
```

nos da como resultado: 01.10.2009

```
CAST ( expression AS data_type [ (length ) ])
```

Donde:

- expresion, la expresion que queremos convertir
- data_type, es el tipo de destino al que queremos convertir la expresión, por ejemplo un numérico a carácter o un carácter a numérico.

Un ejemplo utilizando CAST

```
DECLARE @dato int,
      @dato2 varchar(6),
SET @dato = 115
SET @dato2 = cast(@dato AS varchar)+cast(@dato AS varchar)

SELECT @dato2
```

Nos da como resultado: 115115

Para obtener más información sobre las tablas de conversión o los códigos de stilo podemor ir al siguiente enlace:

http://technet.microsoft.com/es-es/library/ms187928.aspx

**Isnull**

Evalua una expresion y si esta es NULL, reemplaza NULL con el valor de reemplazo especificado. El valor de reemplazo debe ser del mismo tipo de datos que la expresion a evaluar.

```
ISNULL ( expresión , valor_de_reemplazo )
```

Veamos un ejemplo.

```
update articulo set ult_fecha =
isnull(ult_fecha, convert(datetime,'31/05/09'))
```

Nos actualizará la fecha de última venta del articulo si está en NULL

**GetDate y GetUTCDate**

GetDate devuelve la fecha y hora actuales del sistema en el formato interno estándar de SQL Server 2005 para los valores datetime.
GetUTCDate devuelve el valor datetime que representa la hora UTC (hora universal coordinada u hora del meridiano de Greenwich) actual.

Veamos un elemplo

```
DECLARE @fechaLocal datetime,
      @fechaUTC datetime

SET @fechaLocal = getdate()
SET @fechaUTC = GETUTCDATE()

SELECT @fechaLocal, @fechaUTC
```

Nos devuelve:
2009-10-05 12:53:14.263 2009-10-05 10:53:14.263

**Datepart**

Datepart devuelve un entero referido a la parte de una fecha o de una hora.

```
DATEPART ( argumento , fecha )
```

Sinedo :

- Argumento: la parte a obtener
- Fecha: la fecha a analizar

Veamos un ejemplo:

```
DECLARE @Trimestre int,
      @Diadelaño int, @semana int

set @Trimestre = DATEPART( qq, getdate() )
```

```
set @Diadelaño = DATEPART( dy, GETDATE() )
set @semana = DATEPART( wk, GETDATE() )
select @trimestre as trimestre,
            @Diadelaño as dia_del_año,
            @semana as Semana
```

Nos devolverá:

| Trimestre | dia_del_año | semana |
|---|---|---|
| 4 | 278 | 41 |

**EXECUTE o EXEC**:

Ejecuta una cadena de comandos o una cadena de caracteres que contienen un proceso por lotes de Transact-SQL

La sintaxis es la siguiente

```
-- Ejecuta un string o cadena de caracteres:
{ EXEC | EXECUTE }
        ( { @string_variable | [ N ]'tsql_string' } [ + ...n ] )
    [ AS { LOGIN | USER } = ' name ' ]
[;]
```

Nosotros solo vamos a estudiar la ejecución de un string, dado que lo veremos continuamente en el capítulo de alta disponibilidad. Pero recordamos que la función EXEC nos servirá también para ejecutar procesos, procedimientos almacenados o funciones que veremos en una ampliación de T-SQL.

Argumentos :

*@ string_variable*
Es el nombre de una variable local. *@string_variable* puede ser cualquier tipo de datos **char**, **varchar**, **nchar** o **nvarchar**.

Ejemplo:

```
DECLARE @SentenciaSql varchar(max), @BBDD varchar(10)
set @BBDD='2009Wb'
set @sentenciaSql='SELECT * FROM ['+@BBDD+'].[dbo].articulo'
EXEC(@SentenciaSql)
```

[N] '*tsql_string*'

Es una cadena de constante. *tsql_string* puede ser del tipo de datos **nvarchar** o **varchar**. Si se incluye N, la cadena se interpreta como del tipo de datos **nvarchar**.
Ejemplo:

```
EXEC(  'SELECT * FROM [2009WB].[dbo].articulo')
```

AS { LOGIN | USER } = ' name '
Usaremos estos argumento en el caso de que el contexto del en el que se ejecuta la instrucción se deba de cambiar o suplantar. Login suplata el usuario a nivel de servidor, User lo suplanta a nivel de base de datos.

## Funciones de análisis PIVOR, ROLLUP y CUBE

### Pivot

Pivot gira, por decirlo de alguna manera, los resultados verticales de una tabla en resultados horizontales, realizando las agregaciones (sum, avg, etc..) que sean necesárias para obtener los resultados.

Para mayor comprensión, veamos un ejemplo:

Esta seria la sintaxis para obtener la suma de ventas por trimestre:

```
SELECT datepart(q,fecha), sum (importe)
    FROM d_albven group by datepart(q,fecha)
```

Ahora vamos a Pivotar el resultado:

```
SELECT 'Ventas' AS Trimestre,
 [1] , [2], [3], [4]
FROM
(SELECT datepart(q,fecha)as trimestre, importe
    FROM d_albven ) AS SourceTable
PIVOT
(
sum(importe)
FOR trimestre IN ( [1], [2], [3], [4])
) AS PivotTable;
```

**ROLLUP:**
El operador ROLLUP es útil para generar informes con subtotales y totales de forma automática. Nos presenta los totales agrupados por los campos del group by.

```
select cliente,articulo ,sum(unidades) as unidades, sum(importe) as
importe
from d_albven
group by cliente, articulo
with rollup
```

**CUBE:**
El operador CUBE es útil para generar un conjunto de resultados que es un cubo multidimensional.

```
select cliente,articulo ,sum(unidades) as unidades, sum(importe) as
importe
from d_albven
group by cliente, articulo
with cube
```

Ahora podemos ejecutar las dos consultas juntas, de forma que analizaremos los resultados:

```
select cliente,articulo ,sum(unidades) as unidades, sum(importe) as
importe
from d_albven
group by cliente, articulo
with rollup
go

select cliente,articulo ,sum(unidades) as unidades, sum(importe) as
importe
from d_albven
group by cliente, articulo
with cube
go
```

En T-SQL también podemos crear funciones propias. Esto lo veremos en posteriores cursos sobre uso avanzado de SQL Server

## 12.- Cursores en Transact SQL

Un cursor es una variable que nos permite recorrer con un conjunto de resultados obtenido a través de una sentencia SELECT fila a fila.
Cuando trabajemos con cursores debemos seguir los siguientes pasos.

- Declarar el cursor, utilizando DECLARE
- Abrir el cursor, utilizando OPEN
- Leer los datos del cursor, utilizando FETCH ... INTO
- Cerrar el cursor, utilizando CLOSE
- Liberar el cursor, utilizando DEALLOCATE

La sintaxis general para trabajar con un cursor es la siguiente.

**Declaración del cursor**

```
DECLARE <nombre_cursor> CURSOR
FOR <sentencia_sql>
```

**Apertura del cursor**

```
OPEN <nombre_cursor>
```

**Lectura de la primera fila del cursor**

```
FETCH <nombre_cursor> INTO <lista_variables>

WHILE (@@FETCH_STATUS = 0)
BEGIN
```

**Lectura de la siguiente fila de un cursor**

```
FETCH <nombre_cursor> INTO <lista_variables>
...
END -- Fin del bucle WHILE
```

**Cierra el cursor**

```
CLOSE <nombre_cursor>
```

**Libera los recursos del cursor**

```
DEALLOCATE <nombre_cursor>
```

Vamos a ver un ejemplo que nos ayude a la comprensión y uso de un cursor:

```
use [master]
go
declare @nombreBBDD nvarchar(128), @sentenciaSQL nvarchar(max)

DECLARE cursorBBDD CURSOR read_only fast_forward forward_only
      FOR (SELECT name  FROM sys.databases where database_id>4)

OPEN cursorBBDD
FETCH NEXT FROM cursorBBDD
      INTO @nombreBBDD
WHILE @@FETCH_STATUS = 0
BEGIN

      set @sentenciaSQL = 'DBCC SHRINKDATABASE(N'''+ @nombreBBDD +''')
;'
      exec(@sentenciaSQL)
FETCH NEXT FROM cursorBBDD
      into @nombreBBDD
END
CLOSE cursorBBDD
DEALLOCATE cursorBBDD
```

Cuando trabajamos con cursores, la funcion @@FETCH_STATUS nos indica el estado de la última instrucción FETCH emitida, los valores posibles son:

| Valor devuelto | Descripción |
|---|---|
| 0 | La instrucción FETCH se ejecutó correctamente. |
| -1 | La instrucción FETCH no se ejecutó correctamente o la fila estaba más allá del conjunto de resultados. |
| -2 | Falta la fila recuperada. |

En la apertura del cursor, podemos especificar los siguientes parámetros:

```
DECLARE <nombre_cursor> CURSOR
[ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR <sentencia_sql>
```

El primer conjunto de parámetros que podemos especificar es [ LOCAL | GLOBAL ]. A continuación mostramos el significado de cada una de estas opciones.

• LOCAL: Especifica que el ámbito del cursor es local para el proceso por lotes, procedimiento almacenado o desencadenador en que se creó el cursor.

• GLOBAL: Especifica que el ámbito del cursor es global para la conexión. Puede hacerse referencia al nombre del cursor en cualquier procedimiento almacenado o proceso por lotes que se ejecute en la conexión.

Si no se especifica GLOBAL ni LOCAL, el valor predeterminado se controla mediante la configuración de la opción de base de datos default to local cursor.

• FORWARD_ONLY: Especifica que el cursor sólo se puede desplazar de la primera a la última fila. FETCH NEXT es la única opción de recuperación admitida.

• SCROLL: Especifica que están disponibles todas las opciones de recuperación (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Si no se especifica SCROLL en una instrucción DECLARE CURSOR la única opción de recuperación que se admite es NEXT. No es posible especificar SCROLL si se incluye también FAST_FORWARD.
Si se incluye la opción SCROLL, la forma en la realizamos la lectura del cursor varia, debiendo utilizar la siguiente sintaxis: FETCH [ NEXT | PRIOR | FIRST | LAST | RELATIVE | ABSOLUTE ] FROM < INTO

• STATIC: Define un cursor que hace una copia temporal de los datos que va a utilizar. Todas las solicitudes que se realizan al cursor se responden desde esta tabla temporal de tempdb; por tanto, las modificaciones realizadas en las tablas base no se reflejan en los datos devueltos por las operaciones de recuperación realizadas en el cursor y además este cursor no admite modificaciones.

• KEYSET: Especifica que la pertenencia y el orden de las filas del cursor se fijan cuando se abre el cursor. El conjunto de claves que identifica las filas de forma única está integrado en la tabla denominada keyset de tempdb.

• DYNAMIC: Define un cursor que, al desplazarse por él, refleja en su conjunto de resultados todos los cambios realizados en los datos de las filas. Los valores de los datos, el orden y la pertenencia de las filas pueden cambiar en cada operación de recuperación. La opción de recuperación ABSOLUTE no se puede utilizar en los cursores dinámicos.

• FAST_FORWARD: Especifica un cursor FORWARD_ONLY, READ_ONLY con las optimizaciones de rendimiento habilitadas. No se puede especificar FAST_FORWARD si se especifica también SCROLL o FOR_UPDATE.

• READ_ONLY: Evita que se efectúen actualizaciones a través de este cursor. No es posible hacer referencia al cursor en una cláusula WHERE CURRENT OF de una instrucción UPDATE o DELETE. Esta opción reemplaza la capacidad de actualizar el cursor.

• SCROLL_LOCKS: Especifica que se garantiza que las actualizaciones o eliminaciones posicionadas realizadas a través del cursor serán correctas. Microsoft SQL Server bloquea las filas cuando se leen en el cursor para garantizar que estarán disponibles para futuras modificaciones. No es posible especificar SCROLL_LOCKS si se especifica también FAST_FORWARD o STATIC.

• OPTIMISTIC: Especifica que las actualizaciones o eliminaciones posicionadas realizadas a través del cursor no se realizarán correctamente si la fila se ha actualizado después de ser leída en el cursor. SQL Server no bloquea las filas al leerlas en el cursor. En su lugar, utiliza comparaciones de valores de columna timestamp o un valor de suma de comprobación si la tabla no tiene columnas timestamp, para determinar si la fila se ha modificado después de leerla en el cursor. Si la fila se ha modificado, el intento de actualización o eliminación posicionada genera un error. No es posible especificar OPTIMISTIC si se especifica también FAST_FORWARD.

• TYPE_WARNING: Especifica que se envía un mensaje de advertencia al cliente si el cursor se convierte implícitamente del tipo solicitado a otro.

Como hemos visto en el ejemplo, podemos especificar multiples parámetros en la apertura de cursor, pero unicamente un parámetro de cada grupo. Por ejemplo:

```
DECLARE cursorBBDD CURSOR read_only fast_forward forward_only
```

# Anexos:

## TRUCOS Y CURIOSIDADES:

**Dar formato a un numero:**
De una tabla con códigos de cliente numérico, convertirlos a códigos de cliente de Eurowin.

```
DECLARE @num int, @nivel3 varchar(3),
@formato varchar(5),@expresion varchar(8)

SET @nivel3 = '430'
SET @formato = '00000'

SET @num = 2002

SET @expresion = @nivel3 + RIGHT( @formato + cast(@num AS varchar), 5)

SELECT @expresion
```

**Importar datos de un Excel:**

- Aprovechando el Integrador de servicios
- Con el Copy-Paste de toda la vida

**Ejecutar comados de sistema desde el MSSMS**

```
-- Para cambiar la configuracion avanzada.
EXEC sp_configure 'show advanced options', 1
GO
-- Para guardar cambios en la configuración
RECONFIGURE
GO
-- Para activar la herramienta
EXEC sp_configure 'xp_cmdshell', 1
GO
-- Para guardar cambios en la configuración
RECONFIGURE
GO
DECLARE @result int
EXEC @result = xp_cmdshell 'dir c:\*.* /s'
IF (@result = 0)
   PRINT 'Realizado'
ELSE
   PRINT 'Fallo'
```

Esto puede ser muy útil si hemos de borra un archivo del disco duro desde la consola del MSSMS, o ejecutar un comando de consola de DOS, como por ejemplo el BCP.

**Uso de trazas y análisis de trazas**

Podemos ejecutar trazas (profiles) en el servidor de SQL Server para visualizar todas las llamadas que se ejecutan sobre la Base de Datos**.**

Gracias a las trazas podemos observar muchos aspectos:

- Podemos visualizar las llamadas de storeds**.** Es un gran indicador para verificar que no se están realizando llamadas absurdas a la base de datos.

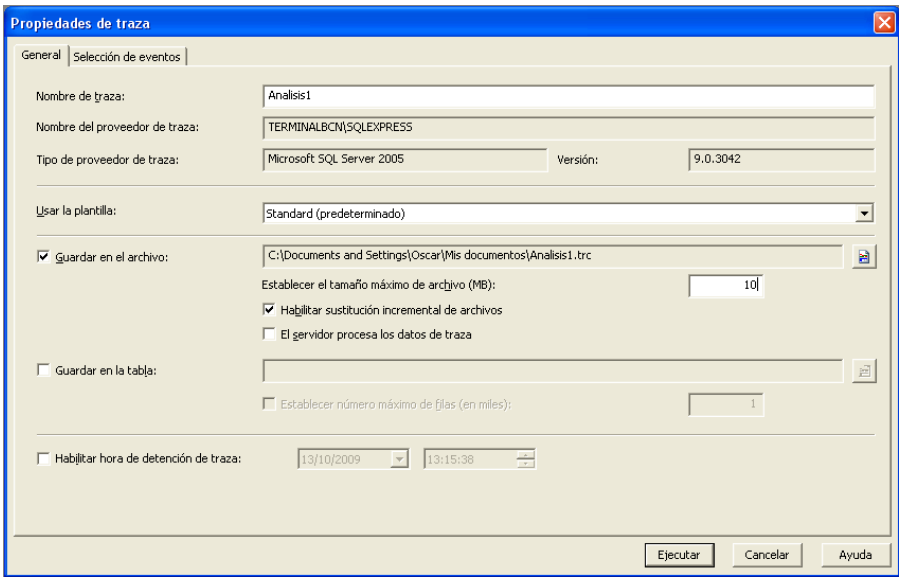
- Podemos ver los tiempos de ejecución de los storeds **(“**duration**”).** Si un stored tarda mucho tiempo en ejecutarse es probable que podamos optimizar las consultas que realiza o incluso añadir índices en las tablas donde participa.

- Otro dellate para estudiar son las lecturas y escrituras**.** Las trazas permiten visualizar los accesos que tenemos en disco físico (lecturas u escrituras). Si tenemos altas lecturas u escrituras podemos optimizar las consultas o añadir índices en las tablas donde participa.

- Y por finalizar, diría que gracias a las trazas nos permiten realizar un pequeño **“**debug”. Si estamos programando aplicaciones que trabajan con storeds puede sernos útil lanzar una traza, recoger la llamada del stored con todos sus parámetros y ejecutar la consulta en el “Query analyzer”.

**Cómo lanzar una traza:**

Suponiendo que tenemos las herramientas de Cliente de SQL Server y un servidor donde poder lanzar una traza apretamos en Inicio, Programas, Microsoft SQL Server 2005, Herramientas de Rendimiento, SQL Profiler**.**




**Cómo localizar la tabla que tiene un campo buscado:**

Supongamos que tenemos el campo ‘NOMBRE2’, pero no sabemos en que tablas está.
Con la siguente consulta podemos encontrar en que tablas está el campo

```
USE  [2009WB];
SELECT TABLE_NAME,*
```

```
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%NOMBRE2%'
GO
```

## COALESCE