Tema 4

Lenguajes de interrogación de bases de datos. Estándar ANSI-SQL.

Guion-resumen

1. **Lenguajes de interrogación de bases de datos**
2. **Subconjuntos de ANSI-SQL**
   - 2.1. Formas de utilizar ANSI-SQL
   - 2.2. Sentencias SQL
3. **Consultas de selección**
   - 3.1. Consultas básicas
   - 3.2. Agrupamiento de registros y cálculo de totales con funciones agregadas
   - 3.3. Subconsultas
   - 3.4. Unión de consultas
   - 3.5. Consultas de combinación entre tablas
4. **Funciones**
   - 4.1. Funciones colectivas
   - 4.2. Funciones escalares
5. **Mantenimiento de los datos. DML**
6. **Definición de los datos. DDL**
   - 6.1. Crear objetos
   - 6.2. Modificar objetos
   - 6.3. Eliminar objetos
7. **Conceptos de interés**
   - 7.1. Variables
   - 7.2. Control de ejecución
   - 7.3. Transacciones
   - 7.4. Cursores
   - 7.5. Procedimientos almacenados *(Store procedure)*
   - 7.6. Desencadenadores *(triggers)*
   - 7.7. Bloqueos

1. Lenguajes de interrogación de bases de datos

SQL (*Structure Query Language* o lenguaje estructurado de consulta) es un lenguaje que permite realizar operaciones diversas sobre datos almacenados en bases de datos relacionales, en los que la información se almacena en tablas bidimensionales, con los datos dispuestos en filas y columnas.

Las sentencias de SQL permiten manejar conjuntos de registros, en lugar de un registro cada vez. La mayoría de los gestores, tanto los basados en una arquitectura cliente/servidor como otros entornos de programación, usan SQL como medio para acceder a los datos.

SQL tiene una estructura relativamente simple, que le otorga una gran flexibilidad y potencia. El número de sentencias existentes en este lenguaje es muy reducido, por lo que facilita el aprendizaje del mismo.

La versión original fue desarrollada por IBM y se denominaba SEQUEL; tenía unas pocas palabras reservadas utilizables con una sintaxis muy sencilla. Cada nuevo producto ha ido incorporando nuevas palabras reservadas, dando paso a nuevos "dialectos" de un mismo lenguaje (SQL de ORACLE, SQL/400 de IBM, Transact SQL Server de Microsoft, etc.).

SQL ha sido estandarizado para lograr así un lenguaje más o menos común para todos los gestores de bases de datos, pero cada uno de estos lenguajes tiene algunos mandatos "propios" que no están incluidos en la lista de palabras reservadas por el American National Standard Institute SQL, o lo que es lo mismo, ANSI-SQL.

Los mandatos se escriben en inglés, y no importa que estén en mayúsculas, minúsculas o intercaladas. Hay que respetar en todo momento el orden sintáctico de las sentencias, no sólo escribir los mandatos correctamente.

2. Subconjuntos de ANSI-SQL

El uso principal de ANSI-SQL es consultar y modificar los datos almacenados en bases de datos relacionales, aunque el lenguaje permite realizar otras tareas. La clasificación de estas tareas permite hacer lo propio con el lenguaje, que se clasifica en:

— Consulta de datos DQL *(Data Query Language)*.

  Consta de sentencias que se encargan de visualizar, organizar y seleccionar los datos de las tablas. La sentencia principal es SELECT.

— Manipulación de datos DML *(Data Manipulation Language).*

  Son sentencias que permiten añadir, modificar y borrar filas sobre las tablas. Estas sentencias son INSERT (para añadir), UPDATE (para modificar) y DELETE (para borrar).

— Definición de datos DDL *(Data Definition Language).*

Son sentencias para crear, modificar, renombrar o borrar objetos (CREATE, ALTER, RENAME y DROP), otorgar restricciones a los campos de las tablas (CHECK, CONSTRAINT y NOT NULL), establecer relaciones entre tablas (PRIMARY KEY, FOREIGN KEY y REFERENCES).

— Control de datos DCL *(Data Control Language).*

Controla la seguridad de los datos; por ejemplo, otorga permisos a usuarios para acceder a los datos. Las sentencias que realizan esto son GRANT y REVOKE.

— Procesado de transacciones TPL *(Transaction-Processing Language).*

Son sentencias encargadas de vigilar mandatos del DML para que funcionen de forma coherente. COMMIT, ROLLBACK y BEGIN TRANSACTION.

— Control de cursores CCL *(Cursor-Control Language).*

Opera sobre filas individuales de una tabla, resultado que afecta a varios registros; FETCH INTO, UPDATE WHERE CURRENT, DECLARE CURSOR.

2.1. Formas de utilizar ANSI-SQL

Los mandatos de SQL pueden ejecutarse en diferentes entornos y lenguajes de programación. Los métodos de uso son:

- **Estáticos:**

  — **SQL interactivo**

  Las sentencias se escriben directamente por parte del usuario y el gestor las responde de manera directa. Un gestor, por ejemplo Access, nos brinda la posibilidad de ejecutar los mandatos directamente (dentro de la sección de consultas, vista SQL), en lugar de utilizar las opciones de los menús.

  — **SQL inmerso en programas *(Embedded SQL)***

  Lenguajes de programación como C, Visual Basic, ASP o JavaScript permiten insertar mandatos de SQL entre sus líneas de código. Cuando se ejecuta el programa, un precompilador interpreta estas órdenes SQL y las envía al gestor de la base de datos. Por ejemplo SQLJ permite embeber sentencia SQL en programas escritos en Java.

  Un programa con SQL embebido es mucho más potente y rápido que si se utiliza el código del propio lenguaje, pues SQL es el lenguaje que utilizan la mayoría de los gestores y la orden la ejecuta directamente.

  — **SQL modular**

  Permite compilar sentencias SQL por separado del lenguaje de programación, para posteriormente enlazarlas (link) con el resto de módulos del programa.

- **Dinámico:**
  - — **SQL dinámico**

    Se dice que los anteriores son SQL estáticos, pues los mandatos ya están escritos. Un ejemplo sería que, una vez que está corriendo un programa, este permitiera al usuario escribir un sentencia SQL y enviarla al gestor.

2.2. Sentencias SQL

Una sentencia SQL está compuesta por:

— **Palabras predefinidas**: tienen un significado propio. Todas las sentencias empiezan por una palabra predefinida. Ejemplos: SELECT, ORDER BY, etc.

— **Nombres de campos y de tablas:** son meros identificadores inventados al crear la tabla y sus campos.

— **Contantes (literales):** representan un valor predeterminado. Los datos alfanuméricos (texto) van entre apóstrofes.

— **Delimitadores:** sirven para delimitar o separar a los anteriores. Son los paréntesis, las comas, espacio en blanco, etc.

— **Tipos de datos:**

- **Numéricos:**

VALOR	LONGITUD	DESCRIPCIÓN
BIT	1 byte	Valores enteros con un valor de 0 o 1
SMALLINT	2 bytes	Un entero corto entre -32.768 y 32.767
INT	4 bytes	Un entero largo entre -2.147.483.648 y 2.147.483.647
DECIMAL	4 bytes	Números de precisión y escala fija con valores de $-10^{38}+1$ a $10^{38}-1$
REAL	4 bytes	Valores de precisión flotante desde $-3{,}40^{38}$ a $3{,}40^{38}$
FLOAT	8 bytes	Valores de precisión flotante desde $-1{,}79^{308}$ a $1{,}79^{308}$

- **Alfanuméricos:**

Valor	Longitud	Descripción
CHAR	1 byte por carácter	Campos de caracteres de longitud fija por carácter Unicode con un tamaño máximo de 8.000 caracteres.
NCHAR	1 byte por carácter	Datos Unicode de longitud fija con un tamaño máximo de 4.000 caracteres.
VARCHAR	1 byte por carácter	Campos de caracteres de longitud variable no Unicode con un tamaño máximo de 8.000 caracteres.
NVARCHAR	1 byte por carácter	Datos Unicode de longitud variable con un tamaño máximo de 4.000 caracteres.
TEXT	1 byte por carácter	Campos de caracteres de longitud variable no Unicode con un tamaño máximo de 2.147.483.647 caracteres.

- **Fecha y hora:**

Valor	Longitud	Descripción
DATETIME	8 bytes	Valores de fecha y hora desde el 1 de enero de 1753 al 31 de Diciembre de 9999.
TIMESTAMP	8 bytes	Captura del sistema, un Instante.

- **Binarios:**

Valor	Longitud	Descripción
BINARY	1 byte	Datos binarios de longitud fija con un tamaño máximo de 8.000 bytes.

— **Operadores:**

- **Lógicos:**

OPERADOR	USO
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve un valor de verdad solo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

- **Relacionales o de comparación:**

OPERADOR	USO
<	Menor que.
>	Mayor que.
<>	Distinto de.
<=	Menor o igual que.
>=	Mayor o igual que.
=	Igual que.
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo.
IN	Utilizado para especificar registros de una base de datos.

- **Aritméticos:**

OPERADOR	USO
()	Paréntesis
*	Multiplicación
/	División
+	Suma
-	Resta
%	Módulo

— **Cláusulas:**

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

CLÁUSULA	DESCRIPCIÓN
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros.
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar.
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos.
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo.
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico.

— **Funciones de agregado:** las funciones de agregado se usan dentro de una clausura SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

FUNCIÓN	DESCRIPCIÓN
AVG	Utilizada para calcular el promedio de los valores de un campo determinado.
COUNT	Utilizada para devolver el número de registros de la selección.
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado.
MAX	Utilizada para devolver el valor más alto de un campo especificado.
MIN	Utilizada para devolver el valor más bajo de un campo especificado.

3. Consultas de selección

SELECT [predicado]

*{ * | tabla.* | [tabla.] campo1 [AS alias1] [, tabla.]campo2 [AS alias2] [, ...] }*

FROM tabla [, ...]

[WHERE criterio] [NOT] [IN] [(valor1, [valor2, [...]])]

[GROUP BY expresion_group]

[HAVING criterio]

[ORDER BY expresion_order [ASC | DESC]]

[cláusula_subconsulta [cláusula_subconsulta [...]]]

Predicado	*Palabra clave (ALL, DISTINCT, TOP) que puede seguir a la cláusula SELECT para restringir el número de registros que se obtienen.*
Tabla	*Nombre de la tabla de la que vamos a obtener los campos.*
campo_n	*Nombre de los campos que van a ser mostrados.*
AS	*Indica que se utiliza un nombre alternativo en lugar del nombre del campo.*
Alias	*Nombre alternativo utilizado al mostrar los campos.*
Criterio	*Condición que determina que registros van a aparecer en la consulta.*
NOT	*Palabra clave que, utilizada como parte de un criterio o junto con el IN, permite indicar qué valores NO han de obtenerse en una consulta.*
IN	*Palabra clave que permite indicar una lista de valores dentro de la cual vamos a buscar.*
Valor1	*Parámetro usado en la cláusula IN para indicar la lista de valores en la que queremos buscar.*
Expresion_group	*Parámetro que especifica por qué campo(s) vamos a crear grupos.*
Expresion_order	*Parámetro que indica qué campo(s) vamos a utilizar para ordenar los resultados y con qué criterio.*
ASC | des	*Especifica que los resultados de la consultas van a ser ordenados en orden ASC Ascendente o DESC Descendente.*
Clausula_subconsulta	*Una consulta anidada.*

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos. Si la tabla estuviera vacía, no mostraría datos o iría acompañado de un mensaje indicando los 0 registros encontrados.

Todos los ejemplos de consultas van a utilizar la tabla árboles:

CLAVE	TIPO	ALTURA	PRECIO	VENDIDO	FECHA
1	pino	2	10,00 €	No	01/01/2008
2	abeto	3	12,00 €	Sí	01/01/2008
3	pino	2	15,00 €	No	01/01/2008
4	cerezo	3	14,00 €	No	01/01/2008
5	pino	4	13,00 €	Sí	01/01/2008
6	abeto	3	18,00 €	No	02/01/2008
7	pino	2	15,00 €	No	02/01/2008
8	pino	5	14,00 €	No	02/01/2008
9	abeto	6	16,00 €	No	02/01/2008
10	cerezo	4	17,00 €	Sí	02/01/2008

3.1. Consultas básicas

SELECT campos **FROM** tabla

Donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

SELECT Tipo, Altura FROM Arboles

TIPO	ALTURA
pino	2
abeto	3
pino	2
cerezo	3
pino	4
abeto	3
pino	2
pino	5
abeto	6
cerezo	4

Podemos preceder el nombre del campo del nombre de la tabla en el caso de que varios campos coincidan en el nombre.

SELECT Clientes.Nombre, Clientes.Teléfono FROM Clientes

En determinadas ocasiones nos puede interesar incluir una columna con un texto fijo en una consulta de selección; por ejemplo, supongamos que tenemos una tabla de empleados y deseamos recuperar sus tarifas semanales, podríamos realizar la siguiente consulta:

SELECT Tipo,'Total:', [Precio]+10 FROM arboles;

TIPO	EXPR1001	EXPR1002
pino	Total:	20,00 €
abeto	Total:	22,00 €
pino	Total:	25,00 €
...	...	...

TIPO	Total:
pino	20,00 €
abeto	22,00 €
pino	25,00 €
cerezo	24,00 €
...	...

También podemos darle un Alias al nuevo campo calculado del siguiente modo:

SELECT Tipo, [Precio]+10 AS Total FROM arboles;

Adicionalmente se puede **especificar el orden** en que se desean recuperar los registros de las tablas mediante la cláusula **ORDER BY** lista de campos. En donde lista de campos representa los campos a ordenar. Ejemplo:

SELECT Tipo, Precio FROM arboles ORDER BY Precio;

Se pueden ordenar los registros por más de un campo, como por ejemplo:

SELECT CodigoPostal, Nombre, Telefono FROM Clientes

ORDER BY CodigoPostal, Nombre

TIPO	ALTURA	PRECIO
pino	2	10,00 €
abeto	3	12,00 €
pino	4	13,00 €
pino	5	14,00 €
cerezo	3	14,00 €
pino	2	15,00 €
pino	2	15,00 €
pino	6	16,00 €
cerezo	4	17,00 €
abeto	3	18,00 €

TIPO	ALTURA	PRECIO
abeto	2	10,00 €
pino	3	12,00 €
pino	4	13,00 €
cerezo	5	14,00 €
abeto	3	14,00 €
cerezo	2	15,00 €
abeto	2	15,00 €
pino	6	16,00 €
pino	4	17,00 €
pino	3	18,00 €

Incluso se puede especificar el orden de los registros: ascendente mediante la cláusula (**ASC** - se toma este valor por defecto) o descendente (**DESC**).

SELECT Tipo, Altura,Precio FROM arboles ORDER BYAltura DESC ,Precio;

El **predicado** se incluye entre la cláusula y el primer nombre del campo a recuperar, los posibles predicados son:

TIPO
pino
abeto
pino
cerezo
pino
abeto
pino
pino
abeto
cerezo

— **ALL:** si no se incluye ninguno de los predicados se asume ALL. El motor de base de datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL y devuelve todos y cada uno de sus campos. No es conveniente abusar de este predicado ya que obligamos al motor de la base de datos a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados.

SELECT ALL Tipo FROM arboles

SELECT * Tipo FROM arboles

ALTURA
6
5
4
4

— **TOP**: devuelve un cierto número de registros que entran entre al principio o al final de un rango especificado por una cláusula ORDER BY (sino devolvería 4 registros elegidos arbitrariamente). Supongamos que queremos recuperar los 4 primeros registros de árboles ordenados por altura en descendente:

SELECT TOP 4 Altura FROM arboles ORDER BY Altura DESC

El predicado TOP no distingue entre valores iguales. No confundamos la anterior consulta con la consulta "obtener las cuatro alturas mayores de la tabla". Vemos que el 4 sale dos veces. Tendríamos que utilizar DISTINCT para eliminar duplicados.

ALTURA
6
5
4
3

SELECT DISTINCT TOP 4 Altura FROM arboles ORDER BY Altura DESC

— **DISTINCT:** omite los registros que contienen datos duplicados en los campos seleccionados. DISTINCT devuelve aquellos registros cuyos campos indicados en la cláusula SELECT posean un contenido diferente.

TIPO
abeto
cerezo
pino

SELECT DISTINCT Tipo FROM arboles

La cláusula WHERE puede usarse para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. Después de escribir esta cláusula se deben especificar las condiciones expuestas en los dos primeros apartados de este capítulo. Si no se emplea esta cláusula, la consulta devolverá todas las filas de la tabla. WHERE es opcional, pero cuando aparece debe ir a continuación de FROM.

SELECT Apellidos, Salario FROM Empleados WHERE Salario > 21000

SELECT Id, Existencias FROM Productos WHERE Existencias <= Pedido

SELECT Apellidos, Nombre FROM Empleados WHERE Apellidos = 'King'

SELECT * FROM Empleados WHERE Edad > 25 AND Edad < 50

SELECT * FROM Empleado WHERE (Edad > 25 AND Edad < 50) OR SUELDO=1

SELECT * FROM Empleados WHERE NOT Estado = 'Soltero'

SELECT * FROM Empleados WHERE (Sueldo > 100 AND Sueldo < 500) OR

(Provincia = 'Madrid' AND Estado = 'Casado')

Intervalos de Valores: para indicar que deseamos recuperar los registros, según el intervalo de valores de un campo, emplearemos el operador **BETWEEN** cuya sintaxis es:

campo **[NOT] BETWEEN** valor1 **AND** valor2 (la condición Not es opcional)

En este caso la consulta devolvería los registros que contengan en "campo" un valor incluido en el intervalo valor1, valor2 (ambos inclusive). Si anteponemos la condición Not devolverá aquellos valores no incluidos en el intervalo.

SELECT * FROM Pedidos WHERE CodPostal BETWEEN 28000 AND 28999

(Devuelve los pedidos realizados en el código postal de la provincia de Madrid)

SELECT Apellidos, Salario FROM Empleados WHERE Salario

BETWEEN 200 AND 300

SELECT Apellidos, Salario FROM Empl WHERE Apellidos

BETWEEN 'Lon' AND 'Tol';

El Operador LIKE: se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL. Su sintaxis es:

expresión **LIKE** modelo

En donde expresión es una variable y modelo un patrón con el que se compara dicha expresión. Se puede utilizar el operador **LIKE** para encontrar valores en los campos que coincidan con el modelo especificado. Por modelo puede especificar un valor completo (Ana María), o se pueden utilizar caracteres comodín como los reconocidos por el sistema operativo para encontrar un rango de valores (LIKE 'An%').

CARÁCTER COMODÍN	DESCRIPCIÓN
%	Cualquier cadena compuesta por uno o más caracteres.
_ (underscore)	Cualquier carácter (sólo uno).
[rango]	Cualquier carácter especificado dentro del rango.
[^ rango]	Cualquier carácter que no aparezca dentro del rango.
#	Cualquier dígito.
a-z	Rango de valores de la 'a' a la 'z' ambos inclusive.

El operador **LIKE** se puede utilizar en una expresión para comparar un valor de un campo con una expresión de cadena. Por ejemplo, si introduce LIKE 'C%' en una consulta SQL, la consulta devuelve todos los valores de campo que comiencen por la letra C.

El ejemplo siguiente devuelve los datos que comienzan con la letra P seguido de cualquier letra entre A y F y de tres dígitos:

```
LIKE 'P[A-F]###'
```

Este ejemplo devuelve los campos cuyo contenido empiece con una letra de la A a la D seguidas de cualquier cadena:

```
LIKE '[A-D]%'
```

En la tabla siguiente se muestra cómo utilizar el operador LIKE para comprobar expresiones con diferentes modelos.

TIPO DE COINCIDENCIA	MODELO	COINCIDE (TRUE)	NO COINCIDE (FALSE)
Varios caracteres	a*a	aa, aBa, aBBBa	aBC
	ab	Abc, AABB, Xab	aZb, bac
	ab*	abcdefg, abc	cab, aab
Caracteres especiales	a[*]a	a*a,	aaa
Un solo carácter	a?a	aaa, a3a, aBa	aBBBA
Un solo dígito	a#a	a0a, a1a, a2a	aaa, a10a
Rango de caracteres	[a-z]	f, p, j	2, &
Fuera de un rango	[!a-z]	9, &, %	b, a
Distinto de un dígito	[!0-9]	A, a, &	0, 1, 9
Combinada	a[!b-m]	An9, az0, a99	abc, aj0

El Operador IN. Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los de una lista. Su sintaxis es:

expresión **[NOT] IN** (valor1, valor2, . . .)

SELECT * FROM Pedidos WHERE Provincia IN ('Madrid', 'Barcelona', 'Sevilla')

SELECT Nombre FROM Empleados WHERE Ciudad IN ('Sevilla', 'Los Angeles')

3.2. Agrupamiento de registros y cálculo de totales con funciones agregadas

La cláusula **GROUP BY** combina los registros con valores idénticos, en la lista de campos especificados, en un único registro. Para cada grupo se calcula un total si se incluye una función SQL agregada, como por ejemplo SUM o COUNT dentro de la instrucción SELECT. GROUP BY es opcional.

A menos que contenga un dato Memo u Objeto OLE, un campo de la lista de campos GROUP BY puede referirse a cualquier campo de las tablas que aparecen en la cláusula FROM, incluso si el campo no está incluido en la instrucción SELECT, siempre y cuando la instrucción SELECT incluya al menos una función SQL agregada.

Todos los campos de la lista de campos de SELECT deben, o incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada.

SELECT Id_Familia, SUM(Stock) FROM Productos GROUP BY Id_Familia

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 sobre los valores calculados como totales.

```
SELECT Id_Familia SUM(Stock) FROM Productos

WHERE NombreProducto LIKE BOS%

GROUP BY Id_Familia

HAVING SUM(Stock) > 100
```

— **AVG**: función agregada que calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta. Su sintaxis es **AVG (expr)** donde **expr** representa el campo que contiene los datos numéricos para los que se desea calcular la media o una expresión que realiza un cálculo utilizando los datos de dicho campo. La media calculada por AVG es la media aritmética (la suma de los valores dividido por el número de valores). La función AVG no incluye ningún campo Null en el cálculo.

```
SELECT AVG (Gastos) AS Promedio FROM Pedidos WHERE Gastos > 100
```

— **COUNT**: calcula el número de registros que hemos obtenido al ejecutar la consulta. Su sintaxis es **COUNT(expr)** 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.

Aunque **expr** puede realizar un cálculo sobre un campo, 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 (*). Si utiliza un asterisco, COUNT calcula el número total de registros, incluyendo aquellos que contienen campos null. COUNT (*) es considerablemente más rápida que COUNT (Campo). No se debe poner el asterisco entre dobles comillas ('*').

```
SELECT COUNT(*) AS Total FROM Pedidos
```

Si **expr** identifica a múltiples campos, la función COUNT cuenta un registro sólo si al menos uno de los campos no es Null. Si todos los campos especificados son Null, no se cuenta el registro. Hay que separar los nombres de los campos con ampersand (&).

```
SELECT COUNT(FechaEnvío & Transporte) AS Total FROM Pedidos
```

Podemos hacer que el gestor cuente los datos diferentes de un determinado campo:

```
SELECT COUNT(DISTINCT Localidad) AS Total FROM Pedidos
```

— **MAX y MIN**: devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo específico de una consulta. Su sin-

taxis es **MIN(expr)** y **MAX(expr),** en donde expr es el campo sobre el que se desea realizar el cálculo. **Expr** puede 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).

SELECT MIN(Gastos) AS ElMin FROM Pedidos WHERE Pais = 'España'

SELECT MAX(Gastos) AS ElMax FROM Pedidos WHERE Pais = 'España'

— **SUM**: devuelve la suma del conjunto de valores contenido en un campo específico de una consulta. Su sintaxis es: **SUM(expr**) en donde **expr** representa el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. 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).

SELECT SUM(PrecioUnidad * Cantidad) AS Total FROM DetallePedido

Ejemplos de totales sobre la tabla árboles.

SELECT Count(Clave) AS CuentaDeClave FROM arboles

Contamos el campo clave y nos devuelve 10 registros. No hemos agrupado por ningún campo, razón por la cual los diez registros forman un único grupo y obtenemos un único total.

CONSULTA1
CuentaDeClave
10

SELECT Count(Clave) AS CuentaDeClave,Tipo FROM arboles GROUP BY Tipo;

Ídem al anterior, pero agrupando por Tipo. Como tenemos tres tipos distintos de árboles, por cada grupo obtenemos un total que nos dice cuántos registros hay en cada grupo.

CONSULTA1	
CuentaDeClave	Tipo
3	Abeto
2	Cerezo
5	Pino

SELECT Tipo, Count(Clave) AS CuentaDeClave, Max(Altura) AS MáxDeAltura, Min(Altura) AS MínDeAltura, Avg(Altura) AS PromedioDeAltura FROM arboles GROUP BYTipo;

CONSULTA1				
Tipo	CuentaDeClave	MáxDeAltura	MínDeAltura	PromedioDeAltura
Abeto	3	6	3	4
Cerezo	2	4	3	3,5
Pino	5	5	2	3

SELECT Tipo,Altura, Sum(Precio) AS SumaDePrecio FROM arboles GROUP BY Tipo,Altura;

CONSULTA1		
Tipo	Altura	Suma De Precio
Abeto	3	30,00 €
Abeto	6	16,00 €
Cerezo	3	14,00 €
Cerezo	4	17,00 €
Pino	2	40,00 €
Pino	4	13,00 €
Pino	5	14,00 €

Primer ejemplo de un total en el que se agrupa por dos campos a la vez, El tipo de árbol y la Altura del mismo. Se han de hacer grupos en los que coincidan el tipo y la altura. Por ello obtenemos abetos de 3 y de 6.

SELECT Tipo, Sum(Precio) AS SumaDePrecio FROM arboles WHERE Tipo="pino"

GROUP BY arboles.Tipo;

En este caso añadimos un criterio con Where para limitar el número de registros que intervienen en la consulta. Sólo quiero totalizar los pinos.

CONSULTA1	
Tipo	Suma De Precio
Pino	67,00 €

SELECT Tipo, Sum(Precio) AS Suma arboles GROUP BYTipo HAVING Sum(.Precio)>40

Y ahora ponemos un criterio al totalizado utilizando HAVING. Sólo quiero aquellas sumas mayores de 40.

CONSULTA1	
Tipo	Suma
Abeto	46,00 €
Pino	67,00 €

3.3. Subconsultas

Una subconsulta es una instrucción SELECT anidada dentro de una instrucción SELECT, SELECT...INTO, INSERT...INTO, DELETE, o UPDATE o dentro de otra subconsulta. Puede utilizar tres formas de sintaxis para crear una subconsulta:

comparación **[ANY | ALL | SOME]** (instrucción SQL)

expresión **[NOT] IN** (instrucción SQL)

[NOT] EXISTS (instrucción SQL)

Tablas que van a utilizarse en los ejemplos de consultas anidadas:

PADRES			
Dni	Nombre	Altura	Fecha Nacimiento
1	Luis	123	12/01/1987
2	Ana	145	13/04/2000
3	Jose	167	17/08/2001
4	Alberto	187	23/04/1998
5	Ana María	123	26/02/1986
6	Alba	156	17/08/2001
7	Santiago	159	23/04/1998
8	Adolfo	100	01/01/2000

HIJOS			
Factura	Producto	Cantidad	Dni
1	p1	12	1
2	p2	10	3
3	p3	6	2
4	p1	7	4
5	p2	5	3
6	p3	8	5
7	p1	9	4
8	p2	4	6
9	p3	12	7
10	p1	19	1
11	p2	9	2
12	p3	8	3
13	p1	5	2
14	p2	4	4
15	p3	6	5

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING. En una subconsulta, se utiliza una instrucción SELECT para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula WHERE o HAVING.

Se puede utilizar el predicado **ANY** o **SOME**, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta.

El ejemplo siguiente devuelve todos los productos cuyo precio unitario es mayor que el de cualquier producto vendido con un descuento igual o mayor al 25 por ciento:

SELECT * FROM PADRES WHERE DNI = ANY (SELECT DNI FROM HIJOS WHERE PRODUCTO LIKE 'P3');

Este ejemplo nos permite mostrar toda la información disponible de aquellos padres que han comprado el producto p3.

Dni	Nombre	Altura	Fecha Nacimiento
2	Ana	145	13/04/2000
2	Jose	167	17/08/2001
5	Ana María	123	26/02/1986
7	Santiago	159	23/04/1998

El predicado **ALL** se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Esto es mucho más restrictivo.

SELECT * FROM Padres WHERE Dni<>All (SELECT dni from Hijos where Producto like 'P3');

Dni	Nombre	Altura	Fecha Nacimiento
1	Luis	123	12/01/1987
4	Alberto	187	23/04/1998
6	Alba	156	17/08/2001

El predicado **IN** se emplea para recuperar únicamente aquellos registros de la consulta principal para los que algunos registros de la subconsulta contienen un valor igual.

SELECT * FROM Padres WHERE DNI IN (SELECT dni from Hijos where Producto like 'P3');

Dni	Nombre	Altura	Fecha Nacimiento
2	Ana	145	13/04/2000
3	Jose	167	17/08/2001
5	Ana María	123	26/02/1986
7	Santiago	159	23/04/1998

Este ejemplo nos permite mostrar toda la información disponible de aquellos padres que han comprado el producto p3. Como podemos observar es idéntica a la anterior.

=ANY es igual que IN

Igualmente se puede utilizar NOT IN para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual.

SELECT * FROM PADRES WHERE DNI IN (SELECT DNI FROM HIJOS WHERE PRODUCTO LIKE 'P3');

Dni	Nombre	Altura	Fecha Nacimiento
1	Luis	123	12/01/1987
4	Alberto	187	23/04/1998
6	Alba	156	17/08/2001

El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro.

SELECT * FROM PADRES WHERE EXISTS (SELECT * FROM HIJOS WHERE HIJOS.DNI=PADRES.DNI)

Esta consulta es equivalente a esta otra:

SELECT * FROM padres WHERE DNI IN (SELECT DNI FROM HIJOS)

Dni	Nombre	Altura	Fecha Nacimiento
1	Luis	123	12/01/1987
2	Ana	145	13/04/2000
3	Jose	167	17/08/2001
4	Alberto	187	23/04/1998
5	Ana María	123	26/02/1986
6	Alba	156	17/08/2001
7	Santiago	159	23/04/1998

Utilizando **NOT** podemos buscar los padres que no tienen hijos.

SELECT * FROM PADRES WHERE NOT EXISTS (SELECT * FROM HIJOS WHERE HIJOS.DNI=PADRES.DNI);

Que es equivalente a:

SELECT * FROM PADRES WHERE PADRES.DNI NOT IN (SELECT DNI FROM HIJOS);

Dni	Nombre	Altura	Fecha Nacimiento
8	Adolfo	100	01/01/2000

Se puede utilizar también alias del nombre de la tabla en una subconsulta para referirse a tablas listadas en la cláusula FROM fuera de la subconsulta. El ejemplo siguiente devuelve los nombres de los empleados cuyo salario es igual o mayor que el salario medio de todos los empleados con su mismo título. A la tabla Empleados se le ha dado el alias T1:

```
SELECT Apellido, Nombre, Titulo, Salario FROM Empleados AS T1
    WHERE Salario >= (SELECT AVG(Salario) FROM Empleados
        WHERE T1.Titulo =Empleados.Titulo) ORDER BY Titulo
```

En el ejemplo anterior, la palabra reservada AS es opcional.

```
SELECT Apellidos, Nombre, Cargo, Salario FROM Empleados
    WHERE Cargo LIKE 'Agente Ven*' AND Salario >ALL (SELECT Salario FROM
    Empleados
        WHERE Cargo LIKE '*Jefe*' OR Cargo LIKE *Director*' )
```

(Obtiene una lista con el nombre, cargo y salario de todos los agentes de ventas cuyo salario es mayor que el de todos los jefes y directores.)

```
SELECT DISTINCT NombreProducto, Precio_Unidad FROM Productos
    WHERE PrecioUnidad = (SELECT PrecioUnidad FROM Productos
        WHERE NombreProducto = 'Almíbar anisado')
```

(Obtiene una lista con el nombre y el precio unitario de todos los productos con el mismo precio que el almíbar anisado).

```
SELECT Nombre, Apellidos FROM Empleados AS E
    WHERE EXISTS (SELECT * FROM Pedidos AS O
        WHERE O.IdEmpleado = E.IdEmpleado)
```

(Selecciona el nombre de todos los empleados que han reservado al menos un pedido).

```
SELECT DISTINCT Pedidos.Id_Producto, Pedidos.Cantidad,
    (SELECT Productos.Nombre FROM Productos
        WHERE Productos.IdProducto = Pedidos.IdProducto) AS ElProducto
            FROM Pedidos WHERE Pedidos.Cantidad = 150
                ORDER BY Pedidos.Id_Producto
```

(Recupera el Código del Producto y la Cantidad pedida de la tabla pedidos, extrayendo el nombre del producto de la tabla de productos).

```
SELECT NumVuelo, Plazas FROM Vuelos
    WHERE Origen = 'Madrid' AND Exists (SELECT T1.NumVuelo FROM Vuelos
    AS T1
        WHERE T1.PlazasLibres > 0 AND T1.NumVuelo=Vuelos.NumVuelo)
```

(Recupera números de vuelo y capacidades de aquellos vuelos con destino Madrid y plazas libres).

Supongamos ahora que tenemos una tabla con los identificadores de todos nuestros productos y el stock de cada uno de ellos. En otra tabla se encuentran todos los pedidos que tenemos pendientes de servir. Se trata de averiguar qué productos no se pueden servir por falta de stock.

```
SELECT PedidosPendientes.Nombre FROM PedidosPendientes
    GROUP BY PedidosPendientes.Nombre
        HAVING SUM(PedidosPendientes.Cantidad > (SELECT Pro-
        ductos.Stock FROM Productos WHERE Productos.IdProducto
        = PedidosPendientes.IdProducto))
```

Supongamos que en nuestra tabla de empleados deseamos buscar todas las mujeres cuya edad sea mayor a la de cualquier hombre:

SELECT Empleados.Nombre FROM Empleados WHERE Sexo = 'M' AND Edad > ANY (SELECT Empleados.Edad FROM Empleados WHERE Sexo ='H')

o lo que sería lo mismo:

SELECT Empleados.Nombre FROM Empleados WHERE Sexo = 'M' AND Edad > (SELECT Max(Empleados.Edad)FROM Empleados WHERE Sexo ='H')

3.4. Unión de consultas

Se utiliza la operación **UNION para crear una consulta de unión, combinando los resultados de dos o más consultas o tablas independientes**. Su sintaxis es:

[TABLE] consulta1 **UNION [ALL] [TABLE]**

consulta2 **[UNION [ALL] [TABLE]** consultaN [...]]

En donde:

consulta1, consulta2, consultaN son instrucciones SELECT, el nombre de una consulta almacenada o el nombre de una tabla almacenada precedido por la palabra clave **TABLE**.

Puede combinar los resultados de dos o más consultas, tablas e instrucciones SELECT, en cualquier orden, en una única operación UNION. El ejemplo siguiente combina una tabla existente llamada Nuevas Cuentas y una instrucción SELECT:

TABLE [Nuevas Cuentas] UNION ALL SELECT * FROM Clientes

WHERE [Cantidad pedidos] > 1000

Si no se indica lo contrario, no se devuelven registros duplicados cuando se utiliza la operación **UNION**, no obstante puede incluir el predicado **ALL** para asegurar que se devuelven todos los registros. Esto hace que la consulta se ejecute más rápidamente. Todas las consultas en una operación UNION deben pedir el mismo número de campos, no obstante los campos no tienen por qué tener el mismo tamaño o el mismo tipo de datos.

Se puede utilizar una cláusula GROUP BY y/o HAVING en cada argumento consulta para agrupar los datos devueltos. Puede utilizar una cláusula ORDER BY al final del último argumento consulta para visualizar los datos devueltos en un orden específico.

SELECT [Nombre de compañía], Ciudad FROM Proveedores

WHERE País = 'Brasil' UNION SELECT [Nombre de compañía], Ciudad FROM Clientes

WHERE País = "Brasil"

(Recupera los nombres y las ciudades de todos proveedores y clientes de Brasil)

SELECT [Nombre de compañía], Ciudad FROM Proveedores

WHERE País = 'Brasil' UNION SELECT [Nombre de compañía], Ciudad FROM Clientes

WHERE País = 'Brasil' ORDER BY Ciudad

(Recupera los nombres y las ciudades de todos proveedores y clientes radicados en Brasil, ordenados por el nombre de la ciudad)

SELECT [Nombre de compañía], Ciudad FROM Proveedores

WHERE País = 'Brasil' UNION SELECT [Nombre de compañía], Ciudad FROM Clientes

WHERE País = 'Brasil' UNION SELECT [Apellidos], Ciudad FROM Empleados

WHERE Región = 'América del Sur'

(Recupera los nombres y las ciudades de todos los proveedores y clientes de Brasil y los apellidos y las ciudades de todos los empleados de América del Sur)

TABLE [Lista de clientes] UNION TABLE [Lista de proveedores]

(Recupera los nombres y códigos de todos los proveedores y clientes)

3.5. Consultas de combinación entre tablas

Las vinculaciones entre tablas se realizan mediante la cláusula **INNER** que combina registros de dos tablas siempre que haya concordancia de valores en un campo común.

Se puede utilizar una operación **INNER JOIN** en cualquier cláusula FROM. Esto crea una combinación por equivalencia, conocida también como unión interna. Las combinaciones equivalentes son las más comunes; estas combinan los registros de dos tablas siempre que haya concordancia de valores en un campo común a ambas tablas. Se puede utilizar **INNER JOIN** con las tablas Departamentos y Empleados para seleccionar todos los empleados de cada departamento. Por el contrario, para seleccionar todos los departamentos (incluso si alguno de ellos no tiene ningún empleado asignado) se emplea **LEFT OUTER JOIN** (la tabla resultado incluye todas las filas de la tabla especificada a la izquierda o tabla dominante. Si los campos de la tabla no tienen correspondencia en la subordinada, se completan con valores NULL) o todos los empleados (incluso si alguno no está asignado a ningún departamento), en este caso **RIGHT OUTER JOIN** (la tabla resultado incluye todas las filas de la tabla situada a la derecha o tabla dominante. Si los campos de la tabla no tienen correspondencia en la subordinada, se completan con valores NULL).

Si se intenta combinar campos que contengan datos Memo u Objeto OLE, se produce un error. Se pueden combinar dos campos numéricos cualesquiera, incluso si son de diferente tipo de datos.

El ejemplo siguiente muestra cómo podría combinar las tablas Categorías y Productos basándose en el campo IDCategoria:

```
SELECT Nombre_Categoría, NombreProducto FROM Categorias
    INNER JOIN Productos ON Categorias.IDCategoria = Productos.IDCategoria
```

En el ejemplo anterior, IDCategoria es el campo combinado, pero no está incluido en la salida de la consulta ya que no está incluido en la instrucción SELECT. Para incluir el campo combinado, se incluye el nombre del campo en la instrucción SELECT, en este caso, Categorias.IDCategoria.

También se pueden enlazar varias cláusulas ON en una instrucción **JOIN**, utilizando la sintaxis siguiente:

```
SELECT campos FROM tabla1 INNER JOIN tabla2
    ON tb1.campo1 comp tb2.campo1 AND
    ON tb1.campo2 comp tb2.campo2) OR
    ON tb1.campo3 comp tb2.campo3)
```

También se pueden anidar instrucciones JOIN utilizando la siguiente sintaxis:

```
SELECT campos FROM tb1 INNER JOIN (tb2 INNER JOIN [( ]tb3
    [INNER JOIN [( ]tablax [INNER JOIN ...)]
    ON tb3.campo3 comp tbx.campox)]
    ON tb2.campo2 comp tb3.campo3)
    ON tb1.campo1 comp tb2.campo2
```

Un **LEFT OUTER JOIN** o un **RIGHT OUTER JOIN** pueden anidarse dentro de un **INNER JOIN**, pero un **INNER JOIN** no puede anidarse dentro de un **LEFT OUTER JOIN** o un **RIGHT OUTER JOIN**.

Por ejemplo:

```
SELECT DISTINCT SUM([Precio unidad] * [Cantidad]) AS [Ventas],
[Nombre] & " " & [Apellidos] AS [Nombre completo] FROM [Detalles de pedidos],
Pedidos, Empleados, Pedidos INNER JOIN [Detalles de pedidos] ON Pedidos.
[ID de pedido] = [Detalles de pedidos].[ID de pedido], Empleados INNER JOIN
Pedidos ON Empleados.[ID de empleado] = Pedidos.[ID de empleado] GROUP BY
[Nombre] & " " & [Apellidos]
```

Crea dos combinaciones equivalentes: una entre las tablas Detalles de pedidos y Pedidos, y la otra entre las tablas Pedidos y Empleados. Esto es necesario ya que la tabla Empleados no contiene datos de ventas y la tabla Detalles de pedidos no contiene datos de los empleados. La consulta produce una lista de empleados y sus ventas totales.

Si empleamos la cláusula **INNER** en la consulta se seleccionarán solo aquellos registros de la tabla de la que hayamos escrito a la izquierda de **INNER JOIN** que contengan al menos un registro de la tabla que hayamos escrito a la derecha. Para solucionar esto tenemos dos cláusulas que sustituyen a la palabra clave **INNER**, estas cláusulas son **LEFT OUTER y RIGHT OUTER. LEFT OUTER** toma todos los registros de la tabla de la izquierda aunque no tengan ningún registro en la tabla de la derecha. **RIGHT OUTER** realiza la misma operación pero al contrario, toma todos los registros de la tabla de la derecha aunque no tenga ningún registro en la tabla de la izquierda.

- **Consultas de auto combinación**

  La auto combinación se utiliza para unir una tabla consigo misma, comparando valores de dos columnas con el mismo tipo de datos. La sintaxis en la siguiente:

  SELECT alias1.columna, alias2.columna, ... **FROM** tabla1 as alias1, tabla2 as alias2

  WHERE alias1.columna = alias2.columna AND otras condiciones

  Por ejemplo, para visualizar el número, nombre y puesto de cada empleado, junto con el número, nombre y puesto del supervisor de cada uno de ellos se utilizaría la siguiente sentencia:

  SELECT t.num_emp, t.nombre, t.puesto, t.num_sup,s.nombre, s.puesto

  FROM empleados AS t, empleados AS s WHERE t.num_sup = s.num_emp

- **Consultas de combinaciones no comunes**

  La mayoría de las combinaciones están basadas en la igualdad de valores de las columnas que son el criterio de la combinación. Las no comunes se basan en otros operadores de combinación, tales como NOT, BETWEEN, <>, etc.

  Por ejemplo, para listar el grado salarial, nombre, salario y puesto de cada empleado, ordenando el resultado por grado y salario, habría que ejecutar la siguiente sentencia:

  SELECT grados.grado,empleados.nombre, empleados.salario, empleados.puesto

  FROM empleados, grados WHERE empleados.salario

  BETWEEN grados.salarioinferior AND grados.salariosuperior

  ORDER BY grados.grado, empleados.salario

Para listar el salario medio dentro de cada grado salarial habría que lanzar esta otra sentencia:

```
SELECT grados.grado, AVG(empleados.salario) FROM empleados, grados
        WHERE empleados.salario
                BETWEEN grados.salarioinferior AND grados.salariosuperior
                        GROUP BY grados.grado
```

- **SELF JOIN**

  SELF JOIN es una técnica empleada para conseguir el producto cartesiano de una tabla consigo misma. Su utilización no es muy frecuente, pero pongamos algún ejemplo de su utilización:

  Supongamos la siguiente tabla (el campo autor es numérico, aunque para ilustrar el ejemplo utilice el nombre):

CÓDIGO (CÓDIGO DEL LIBRO)	AUTOR (NOMBRE DEL AUTOR)
B0012	1. Francisco López
B0012	2. Javier Alonso
B0012	3. Marta Rebolledo
C0014	1. Francisco López
C0014	2. Javier Alonso
D0120	2. Javier Alonso
D0120	3. Marta Rebolledo

Queremos obtener, para cada libro, parejas de autores:

```
SELECT A.Codigo, A.Autor, B.Autor FROM Autores A, Autores B
    WHERE A.Codigo = B.Codigo
```

El resultado es el siguiente:

AUTOR	AUTOR
1. Francisco López	1. Francisco López
1. Francisco López	2. Javier Alonso
1. Francisco López	3. Marta Rebolledo
2. Javier Alonso	2. Javier Alonso
2. Javier Alonso	1. Francisco López
2. Javier Alonso	3. Marta Rebolledo
3. Marta Rebolledo	3. Marta Rebolledo
3. Marta Rebolledo	2. Javier Alonso
3. Marta Rebolledo	1. Francisco López
1. Francisco López	1. Francisco López
1. Francisco López	2. Javier Alonso
2. Javier Alonso	2. Javier Alonso
2. Javier Alonso	1. Francisco López
2. Javier Alonso	2. Javier Alonso
2. Javier Alonso	3. Marta Rebolledo
3. Marta Rebolledo	3. Marta Rebolledo
3. Marta Rebolledo	2. Javier Alonso

Como podemos observar, las parejas de autores se repiten en cada uno de los libros, podemos omitir estas repeticiones de la siguiente forma:

```
SELECT A.Codigo, A.Autor, B.Autor FROM Autores A, Autores B
    WHERE A.Codigo = B.Codigo AND A.Autor < B.Autor
```

El resultado ahora es el siguiente:

AUTOR	AUTOR
1. Francisco López	2. Javier Alonso
1. Francisco López	3. Marta Rebolledo
1. Francisco López	2. Javier Alonso
2. Javier Alonso	3. Marta Rebolledo

Ahora tenemos un conjunto de resultados en formato Autor-CoAutor.

Si en la tabla de empleados quisiéramos extraer todas las posibles parejas que podemos realizar, utilizaríamos la siguiente sentencia:

```
SELECT Hombres.Nombre, Mujeres.Nombre
    FROM Empleados Hombre, Empleados Mujeres
        WHERE Hombre.Sexo = 'Hombre' AND Mujeres.Sexo = 'Mujer'
            AND Hombres.Id <>Mujeres.Id
```

Para concluir supongamos la tabla siguiente:

ID	NOMBRE	SUBJEFE
1	Marcos	6
2	Lucas	1
3	Ana	2
4	Eva	1
5	Juan	6
6	Antonio	

Queremos obtener un conjunto de resultados con el nombre del empleado y el nombre de su jefe:

```
SELECT Emple.Nombre, Jefes.Nombre FROM Empleados Emple, Empleados Jefe
    WHERE Emple.SuJefe = Jefes.Id
```

4. Funciones

Una función representa un valor único que se obtiene aplicando unas determinadas operaciones a otros valores (argumentos). Hay dos tipos de funciones: colectivas y escalares.

4.1. Funciones colectivas

Dan como resultado un único valor después de aplicar la función a un grupo de valores.

SELECT AVG(sueldo), SUM (sueldo) MAX (sueldo) MIN (sueldo) COUNT (sueldo) FROM (empleados).

Da como resultado un único registro con cinco columnas. La primera columna correspondería con la media aritmética de los sueldos de los empleados, la segunda con el total del sueldo de todos los empleados, la tercera con el sueldo máximo, la cuarta con el sueldo mínimo y la quinta con el total de registros que tiene la tabla.

4.2. Funciones escalares

Operan sobre un único dato y devuelven un único valor como resultado.

FUNCIÓN	DESCRIPCIÓN
DATE	Obtiene la fecha y hora actual del sistema.
DAY (fecha)	Obtiene el día de una fecha como un valor entero.
MONTH(fecha)	Obtiene el mes de una fecha como un valor entero.
YEAR(fecha)	Obtiene el año de una fecha como un valor entero.

- **Funciones matemáticas**

FUNCIÓN	DESCRIPCIÓN
ABS(x)	Valor absoluto. Convierte números negativos en positivos, o deja solo números positivos.
ACOS(x)	Obtiene el arcocoseno.
ATAN(x)	Obtiene la arcotangente.
CEIL(x)	Obtiene el menor entero, mayor o igual que x. Redondeo hacia arriba.
COS(x)	Obtiene el coseno trigonométrico.
COT(x)	Obtiene la cotangente trigonométrica.
EXP(x)	Obtiene el valor del exponente.
FLOOR(x)	Obtiene el mayor entero, menor o igual que x. Redondeo hacia abajo.
INT(x)	Devuelve la parte entera.

FUNCIÓN	DESCRIPCIÓN
LOG(x)	Obtiene el logaritmo natural.
LOG10(x)	Obtiene el logaritmo base 10.
MOD(x,y)	Obtiene el resto de dividir x entre y.
PI	Obtiene el valor de la constante pi.
POWER(x,y)	Obtiene el valor de x elevado a y.
ROUND(x,y)	Redondea x a y lugares decimales. Si se omite y, x se redondea al entero más próximo.
SIGN(x)	Devuelve +1 si la x es positivo, 0 si es cero, y -1 si es negativo.
SIN(x)	Obtiene el seno trigonométrico.
SQUARE(x)	Obtiene el cuadrado.
SQRT(x)	Obtiene la raíz cuadrada.
TAN(x)	Obtiene la tangente.

- **Funciones de cadena**

FUNCIÓN	DESCRIPCIÓN
ASCII(x)	Obtiene el código ASCII de x.
CHAR(x)	Obtiene el carácter ASCII cuyo código entero corresponde a x.
LEFT(x,y)	Obtiene los y caracteres de la izquierda de x.
LEN(x)	Obtiene el número de caracteres de x.
LOWER(x)	Obtiene x con todos sus caracteres convertidos a minúsculas.
LTRIM(x)	Obtiene x quitándoles los espacios iniciales.
REPLACE(x,y,z)	Encuentra todas las apariciones de y en x, reemplazándolas por z.
RIGHT(x,y)	Obtiene los y caracteres de la derecha de x.
RTRIM(x)	Obtiene x sin espacios a la derecha.
SPACE(x)	Obtiene x espacios.
SUBSTRING(x,y,z)	Obtiene z caracteres de x, comenzando en la posición y.
UPPER(x)	Obtiene x con todos sus caracteres convertidos a mayúsculas.

- **Funciones de conversación**

FUNCIÓN	DESCRIPCIÓN
CTOD(x)	Convierte una cadena de caracteres a una fecha. Se puede utilizar un segundo parámetro para especificar el formato de la fecha devuelta: 0 (por defecto devuelve MM/DD/YY, 1 devuelve DD/MM/YY y 2 devuelve YY/MM/DD).
CAST(x,y)	Convierte la cadena y, pasada como argumento, en el tipo especificado x (si es posible).
DTOC(x)	Convierte una fecha x, a una cadena de caracteres. Un segundo parámetro opcional determina el formato del resultado: 0 (por defecto) devuelve MM/DD/YY,1 devuelve DD/MM/YY, 2 devuelve YY/MM/DD, 10 devuelve MM/DD/YYYY, 11 devuelve DD/MM/YYYY, 12 devuelve YYYY/MM/DD. Puede existir un tercer parámetro opcional para determinar el carácter que se quiere utilizar como separador. Si no se especifica se toma el (/).
STR(x,y)	Convierte un número x en una cadena. Devuelve y posiciones (incluyendo el punto decimal). Opcionalmente se puede incluir un tercer parámetro para indicar el número de dígitos a la derecha del punto decimal.
STRVAL(x)	Convierte un valor de cualquier tipo a una cadena de caracteres.

5. Mantenimiento de los datos. DML

El Lenguaje de Manipulación de Datos (DML) se compone de las instrucciones para crear y recuperar datos. Son sentencias que no devuelven ningún registro. Son las encargadas de mantener actualizados los datos que están almacenados en las tablas.

- **DELETE**

  La sentencia DELETE se utiliza para borrar registros de una tabla de la base de datos. No es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

  DELETE FROM Tabla [**WHERE** { condición }]

  La cláusula WHERE sigue el mismo formato que la vista en la sentencia SELECT y determina qué registros se borrarán.

  Cada sentencia **DELETE** borra los registros que cumplen la condición impuesta o todos si no se indica cláusula WHERE.

  DELETE FROM Empleados

Con el ejemplo anterior se borrarían todos los registros de la tabla Empleados. Se llama vaciado y solo quedaría la estructura de la tabla.

DELETE FROM Empleados WHERE Cargo = 'Vendedor'

- **INSERT**

La sentencia **INSERT** se utiliza para añadir registros a las tablas de la base de datos. Puede ser de dos tipos: insertar un único registro o insertar en una tabla los registros contenidos en otra tabla.

Para insertar un único registro, la sintaxis es la siguiente:

INSERT INTO Tabla (campo1, campo2, ..., campoN)

VALUES (valor1, valor2, ..., valorN)

Esta sentencia graba en el campo1 el valor1, en el campo2 el valor2 y así sucesivamente.

Para insertar registros de otra tabla, la sintaxis es la siguiente:

INSERT INTO Tabla (campo1, campo2, , campoN)

SELECT TablaOrigen.campo1, TablaOrigen.campo2,,TablaOrigen.campoN

FROM TablaOrigen

En este caso se seleccionarán los campos 1,2,..., n de la TablaOrigen y se grabarán en los campos 1,2,.., n de la Tabla. La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si Tabla y TablaOrigen poseen la misma estructura podemos simplificar la sintaxis a:

INSERT INTO Tabla **SELECT** TablaOrigen.* **FROM** TablaOrigen

De esta forma los campos de TablaOrigen se grabarán en Tabla, para realizar esta operación es necesario que todos los campos de TablaOrigen estén contenidos con igual nombre en Tabla. Con otras palabras, que Tabla posea todos los campos de TablaOrigen (igual nombre e igual tipo).

En este tipo de sentencia hay que tener especial atención con los campos contadores o autonuméricos puesto que al insertar un valor en un campo de este tipo se escribe el valor que contenga su campo homólogo en la tabla origen, no incrementándose como le corresponde.

Si la tabla destino contiene una clave principal, hay que asegurarse de que es única, y con valores no nulos; si no es así, no se agregarán los registros.

INSERT INTO Empleados (Nombre, Apellido, Cargo)

VALUES ('Luis', 'Sánchez', 'Becario')

```
INSERT INTO Empleados SELECT * FROM Vendedores
    WHERE Provincia = 'Madrid'
INSERT INTO Oils (OilName, Latiname, Simple)
    VALUES('Super', NULL, NULL)
INSERT INTO MyOils (OilName, LatinName)
    SELECT OilName, LatinName FROM Oils
        WHERE (LEFT(OilName), 2) = '98')
```

- **UPDATE**

La sentencia **UPDATE** se utiliza para cambiar el contenido de los registros de una tabla de la base de datos. Su sintaxis es:

UPDATE Tabla **SET** Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN

[WHERE { condición }]

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando estos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez.

La cláusula WHERE sigue el mismo formato que la vista en la sentencia SELECT y determina qué registros se modificarán.

El ejemplo siguiente incrementa los valores Cantidad pedidos en un 10% y los valores Transporte en un 3% para aquellos que se hayan enviado al Reino Unido:

```
UPDATE Pedidos SET Pedido = Pedidos * 1.1, Transporte = Transporte * 1.03
    WHERE PaisEnvío = 'ES'
UPDATE Empleados SET Grado = 5 WHERE Grado = 2
UPDATE Productos SET Precio = Precio * 1.1
    WHERE Proveedor = 8 AND Familia = 3
```

Si en una consulta de actualización suprimimos la cláusula WHERE todos los registros de la tabla señalada serán actualizados.

```
UPDATE Empleados SET Salario = Salario * 1.1
UPDATE Libros SET Precio = (SELECT AVG(Precio) FROM Libros
    WHERE Precio IS NOT NULL)
        WHERE Precio IS NULL
```

Con esta última sentencia se ha puesto precio a todos los libros que no lo tenían. Ese precio ha sido el resultante de calcular la media entre los libros que sí lo tenían.

- **SELECT...INTO**

  Esta sentencia se utiliza para seleccionar registros e insertarlos en una tabla nueva. Su sintaxis es:

  SELECT campo1, campo2, ..., campoN **INTO** NuevaTabla

  FROM TablaOrigen [**WHERE** { condición }]

  Las columnas de la nueva tabla tendrán el mismo tipo y tamaño que las columnas origen, y se llamarán con el nombre de alias de la columna origen, o en su defecto, con el nombre de la columna origen, pero no se transfiere ninguna otra propiedad del campo o de la tabla como, por ejemplo, las claves e índices.

  La sentencia **SELECT** puede ser cualquier sentencia **SELECT** sin ninguna restricción, puede ser una consulta multitabla, una consulta de resumen, una UNION, etc.

  SELECT * INTO Programadores FROM Empleados

  WHERE Categoria = 'Programador'

  Esta consulta crea una tabla nueva llamada Programadores con igual estructura que la tabla Empleados y copia aquellos registros cuyo campo Categoria sea "Programador".

  Por ejemplo: queremos enviarle a un representante una tabla con todos los datos personales de sus clientes para que les pueda enviar cartas, etc...

  SELECT Numclie AS Codigo, Nombre, Direccion, Telefono INTO Susclientes

  FROM Clientes WHERE Repclie = '103'

  En el ejemplo anterior la nueva tabla tendrá cuatro columnas llamadas Codigo, Nombre, Direccion, Telefono y contendrá las filas correspondientes a los clientes del representante 103.

- **MERGE**

  Se utiliza para seleccionar filas de una o más fuentes para la actualización o inserción en una tabla o vista. Puede especificar las condiciones para determinar si se debe actualizar o insertar en la tabla de destino o vista. MERGE es una afirmación determinista. No se puede actualizar la misma fila de la tabla de destino varias veces en la misma instrucción MERGE. Esta sentencia se utiliza para seleccionar registros e insertarlos en una tabla nueva. Su sintaxis es:

```
MERGE INTO tabla_destino USING tabla_origen ON (condición)
    WHEN MATCHED THEN
    UPDATE SET
        campo1 = valor1,
        campoN = valorN
    WHEN NOT MATCHED THEN
    INSERT (campo1, ...., campoN)
    VALUES (valor1, ..., valorN);
```

Veamos un ejemplo:

```
MERGE INTO clientes cli USING datos_cli dac
ON (cli.cliente_id = dac.cliente_id)
    WHEN MATCHED THEN
    UPDATE SET
        cli.nombre = dac.nombre,
        cli.direccion = dac.direccion
    WHEN NOT MATCHED THEN
    INSERT (cliente_id, nombre, direccion)
    VALUES (dac.cliente_id, dac.nombre, dac.direccion);
```

En este ejemplo cuando encontremos un cliente que exista en ambas tablas mediante su "id" actualizará su nombre y dirección. En caso de que no exista alguno de los clientes que exista en la tabla "datos_cli" que no exista en la tabla "clientes" los insertará.

6. Definición de los datos. DDL

El Lenguaje de Definición de Datos (DDL) consta de las sentencias utilizadas para crear los objetos dentro de la base de datos y cambiar las propiedades y atributos de la propia base de datos.

COMANDO	DESCRIPCIÓN
CREATE	Utilizado para crear objetos de base de datos.
DROP	Utilizado para eliminar un objeto de la base de datos.
ALTER	Utilizado para modificar o alterar un objeto de la base de datos.
TRUNCATE	Elimina el contenido de una tabla, la vacía pero no modifica su estructura.

6.1. Crear objetos

Los objetos de base de datos se crean utilizando la sentencia CREATE. Su sintaxis exacta varía para cada objeto.

SINTAXIS	OBJETO CREADO
CREATE DATABASE nombre	Crea una base de datos.
CREATE DEFAULT nombre AS expresión	Crea una propiedad determinada.
CREATE FUNCTION nombre RETURNS valor AS sentencias	Crea una función definida por el usuario.
CREATE INDEX nombre ON tabla (columnas)	Crea un índice sobre una tabla.
CREATE PROCEDURE nombre AS sentencias	Crea un procedimiento almacenado.
CREATE RULE nombre AS expresión	Crea una regla de base de datos.
CREATE TABLE nombre (definición)	Crea una tabla.
CREATE TRIGGER nombre {FOR | AFTER | INSTEAD OF} acción AS sentencias	Crea un desencadenador.
CREATE VIEW nombre AS sentencia_select	Crea una vista.

- **Crear una tabla**

  De las sentencias **CREATE** descritas, la más compleja es la sentencia **CREATE TABLE**, a causa del número de elementos diferentes que comprenden una definición de tabla. Debe añadir columnas, por supuesto, y cada definición de columna debe tener al menos un nombre y un tipo de datos. Opcionalmente puede especificar si la columna admite nulos, su valor por defecto, cualquier restricción aplicable a la columna...etc. Su sintaxis simplificada es:

CREATE TABLE Tabla (campo1 tipo (tamaño) índice1 ,

campo2 tipo (tamaño) índice2 , ..., índice multicampo , ...)

En donde:

PARTE	DESCRIPCIÓN
Tabla	Es el nombre de la tabla que se va a crear.
campo1 campo2	Es el nombre del campo o de los campos que se van a crear en la nueva tabla. La nueva tabla debe contener, al menos, un campo.
Tipo	Es el tipo de datos de campo en la nueva tabla.
Tamaño	Es el tamaño del campo y solo se aplica para campos de tipo texto.
índice1 índice2	Es una cláusula **CONSTRAINT** que define el tipo de índice a crear. Esta cláusula es opcional.
índice multicampos	Es una cláusula **CONSTRAINT** que define el tipo de índice multicampos a crear. Un índice multicampo es aquel que está indexado por el contenido de varios campos. Esta cláusula es opcional.

CREATE TABLE Empleados (Nombre TEXT (25) , Apellidos TEXT (50))

Crea una nueva tabla llamada Empleados con dos campos, uno llamado Nombre de tipo texto y longitud 25 y otro llamado apellidos con longitud 50.

CREATE TABLE Empleados (Nombre TEXT (10), Apellidos TEXT,

Fecha_Nacimiento DATETIME) CONSTRAINT IndiceGeneral UNIQUE

([Nombre], [Apellidos], [Fecha_Nacimiento])

Crea una nueva tabla llamada Empleados con un campo Nombre de tipo texto y longitud 10, otro con llamado Apellidos de tipo texto y longitud predeterminada y uno más llamado Fecha_Nacimiento de tipo Fecha/Hora. También crea un índice único (no permite valores repetidos) formado por los tres campos.

CREATE TABLE Empleados (ID INT CONSTRAINT IndicePrimario PRIMARY,

Nombre TEXT, Apellidos TEXT, Fecha_Nacimiento DATETIME)

Crea una tabla llamada Empleados con un campo Texto de longitud predeterminada llamado Nombre y otro igual llamado Apellidos, crea otro campo llamado Fecha_Nacimiento de tipo Fecha/Hora y el campo ID de tipo entero el que establece como clave principal.

- **La cláusula CONSTRAINT**

  Se utiliza la cláusula CONSTRAINT en las instrucciones **ALTER TABLE y CREATE TABLE** para crear o eliminar índices. Existen dos sintaxis para esta cláusula dependiendo de si desea crear o eliminar un índice de un único campo o si se trata de un campo multiíndice.

  Para los índices de campos únicos:

  CONSTRAINT nombre

  {PRIMARY KEY | UNIQUE | REFERENCES tabla externa

  [(campo externo1, campo externo2)]}

  Para los índices de campos múltiples:

  CONSTRAINT nombre

  {**PRIMARY KEY** (primario1[, primario2 [, ...]]) |

  UNIQUE (único1[, único2 [, ...]]) |

  FOREIGN KEY (ref1[, ref2 [, ...]])

  REFERENCES tabla externa [(campo externo1

  [,campo externo2 [, ...]])]}

PARTE	DESCRIPCIÓN
nombre	Es el nombre del índice que se va a crear.
primarioN	Es el nombre del campo o de los campos que forman el índice primario.
únicoN	Es el nombre del campo o de los campos que forman el índice de clave única.
refN	Es el nombre del campo o de los campos que forman el índice externo (hacen referencia a campos de otra tabla).
tabla externa	Es el nombre de la tabla que contiene el campo o los campos referenciados en refN
campos externosN	Es el nombre del campo o de los campos de la tabla externa especificados por ref1, ref2, ..., refN

Si se desea crear un índice para un campo cuando se esta utilizando las instrucciones ALTER TABLE o CREATE TABLE la cláusula **CONSTRAINT** debe aparecer inmediatamente después de la especificación del campo indexado.

Si se desea crear un índice con múltiples campos cuando se está utilizando las instrucciones ALTER TABLE o CREATE TABLE la cláusula **CONSTRAINT** debe aparecer fuera de la cláusula de creación de tabla.

TIPO DE ÍNDICE	DESCRIPCIÓN
UNIQUE	Genera un índice de clave única. Lo que implica que los registros de la tabla no pueden contener el mismo valor en los campos indexados.
PRIMARY KEY	Genera un índice primario el campo o los campos especificados. Todos los campos de la clave principal deben ser únicos y no nulos, y cada tabla solo puede contener una única clave principal.
FOREIGN KEY	Genera un índice externo (toma como valor del índice campos contenidos en otras tablas). Si la clave principal de la tabla externa consta de más de un campo, se debe utilizar una definición de índice de múltiples campos, listando todos los campos de referencia, el nombre de la tabla externa, y los nombres de los campos referenciados en la tabla externa en el mismo orden que los campos de referencia listados. Si los campos referenciados son la clave principal de la tabla externa, no tiene que especificar los campos referenciados, el motor se comporta como si la clave principal de la tabla externa fueran los campos referenciados.

```
CREATE TABLE Empleados (DNI CHAR(10), nombemp CHAR(15),
    Apellemp CHAR(40), sueldo DECIMAL,
        CONSTRAINT pk_dni PRIMARY KEY (DNI))
```

Donde pk_dni es el nombre de la restricción (nombre de uso interno para el gestor) y DNI es el campo a vincular (índice primario).

```
CREATE TABLE Ventas (numventa INT IDENTITY (100,1), DNI CHAR(10),
    Fecha DATETIME, codart CHAR(5),
        CONSTRAINT pk_dni FOREIGN KEY (DNI))
```

Esta orden crea una tabla Ventas en la cual se otorga al campo DNI el atributo de clave externa (índice externo). El campo numventa es autonumérico, empieza a contar desde 100 con un incremento de 1.

- **Crear una vista**

  Es una sentencia SELECT a la que se otorga un nombre y se guarda en el catálogo. El resultado es una tabla y pueden realizarse operaciones sobre ella, pero no es propiamente una tabla, ni tiene datos propios. Se pueden utilizar sobre ella los mandatos SELECT habituales. Sirve para preservar los datos de la tabla "verdadera".

```
CREATE VIEW suspensos AS SELECT * FROM alumnos WHERE nota < 5
```

Crea una vista con los suspensos de la tabla alumnos.

```
CREATE VIEW VistaSimple
AS SELECT IDRelacionada, DescripcionSimple, DescripcionRelacionada
    FROM TablaRelacionada
    INNER JOIN TablaSimple
    ON TablaRelacionada.IDSimple = TablaSimple.IDSimple
```

- **Crear un índice**

  Los índices son unos atributos de ordenación interna para campos de tipo fecha, numéricos y alfanuméricos (no memo).

  Las búsquedas basadas en campos indexados se realizan de una forma más rápida, entre otras ventajas.

  No es bueno que existan muchos campos indexados en una misma tabla, pues debido a la longitud que tomaría el registro interno, puede incluso hacer que una búsqueda sea más tediosa o lenta, justo el efecto contrario a lo que se pretende.

  Existen dos tipos de índices: **índice único**, para campos que no permiten valores iguales en una misma columna, también llamado sin duplicados (UNIQUE) y los llamados **índices con duplicados**. Su sintaxis es:

  CREATE [UNIQUE] INDEX índice

  ON tabla (campo [**ASC** | **DESC**][, campo [**ASC** | **DESC**], ...])

  [WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]

En donde:

PARTE	DESCRIPCIÓN
ÍNDICE TABLA	Es el nombre del índice a crear. Es el nombre de una tabla existente en la que se creará el índice.
CAMPO	Es el nombre del campo o lista de campos que constituyen el índice.
ASC | DESC	Indica el orden de los valores de los campos. ASC indica un orden ascendente (valor predeterminado) y DESC un orden descendente.
UNIQUE	Indica que el índice no puede contener valores duplicados.
DISALLOW NULL	Prohíbe valores nulos en el índice.
IGNORE NULL	Excluye del índice los valores nulos incluidos en los campos que lo componen.
PRIMARY	Asigna al índice la categoría de clave principal, en cada tabla solo puede existir un único índice que sea "Clave Principal". Si un índice es clave principal implica que no puede contener valores nulos ni duplicados.

CREATE INDEX MiIndice ON Empleados (Prefijo, Telefono)

Crea un índice llamado MiIndice en la tabla Empleados con los campos Prefijo y Telefono.

CREATE UNIQUE INDEX MiIndice ON Empleados (ID) WITH DISALLOW NULL

Crea un índice en la tabla Empleados utilizando el campo ID, obligando que el campo ID no contenga valores nulos ni repetidos.

6.2. Modificar objetos

Del mismo modo que la sentencia CREATE crea un nuevo objeto, la sentencia **ALTER** proporciona el mecanismo para alterar una definición de objeto. No todos los objetos creados por una instrucción CREATE tienen su correspondiente sentencia **ALTER**. Su sintaxis exacta varía para cada objeto.

SINTAXIS	OBJETO CREADO
ALTER DATABASE nombre especif_archivo	Modifica los archivos utilizados para almacenar la base de datos.
ALTER FUNCTION nombre RETURNS valor AS sentencias	Cambia las sentencias SQL que componen la función.
ALTER PROCEDURE nombre AS sentencias	Cambia las sentencias SQL que componen el procedimiento almacenado.
ALTER TABLE nombre (definición)	Cambia la definición de una tabla.
ALTER TRIGGER nombre {FOR | AFTER | INSTEAD OF} acción AS sentencias	Cambia las sentencias SQL que componen el desencadenador.
ALTER VIEW nombre AS sentencia_select	Cambia la sentencia SELECT que crea la vista.

Modificar una tabla

La sentencia **ALTER TABLE** es compleja por la misma razón que la sentencia CREATE TABLE: hay varias partes diferentes en una definición de tabla. Su sintaxis simplificada es:

ALTER TABLE Tabla {**ADD** | **ALTER** {**COLUMN** tipo de campo[(tamaño)]

[**CONSTRAINT** índice] **CONSTRAINT** índice multicampo} |

DROP {**COLUMN** campo [**CONSTRAINT** nombre del índice]} }

En donde:

PARTE	DESCRIPCIÓN
tabla	Es el nombre de la tabla que se desea modificar.
campo	Es el nombre del campo que se va a añadir o eliminar.
tipo	Es el tipo de campo que se va a añadir.
tamaño	El tamaño del campo que se va a añadir (sólo para campos de texto).
índice	Es el nombre del índice del campo (cuando se crean campos) o el nombre del índice de la tabla que se desea eliminar.
índice multicampo	Es el nombre del índice del campo multicampo (cuando se crean campos) o el nombre del índice de la tabla que se desea eliminar.

OPERACIÓN	DESCRIPCIÓN
ADD COLUMN	Se utiliza para añadir un nuevo campo a la tabla, indicando el nombre, el tipo de campo y opcionalmente el tamaño (para campos de tipo texto).
ADD	Se utiliza para agregar un índice de multicampos o de un único campo.
DROP COLUMN	Se utiliza para borrar un campo. Se especifica únicamente el nombre del campo.
DROP	Se utiliza para eliminar un índice. Se especifica únicamente el nombre del índice a continuación de la palabra reservada CONSTRAINT.

ALTER TABLE Empleados ADD COLUMN Salario DECIMAL

Agrega un campo Salario de tipo Numérico a la tabla Empleados.

ALTER TABLE Empleados DROP COLUMN Salario.

Elimina el campo Salario de la tabla Empleados.

ALTER TABLE Pedidos ADD CONSTRAINT RelacionPedidos FOREIGN KEY

(ID_Empleado) REFERENCES Empleados (ID_Empleado)

Agrega un índice externo a la tabla Pedidos. El índice externo se basa en el campo ID_Empleado y se refiere al campo ID_Empleado de la tabla Empleados. En este ejemplo no es necesario indicar el campo

junto al nombre de la tabla en la cláusula REFERENCES, pues ID_Empleado es la clave principal de la tabla Empleados.

ALTER TABLE Pedidos DROP CONSTRAINT RelacionPedidos

Elimina el índice de la tabla Pedidos.

ALTER TABLE TablaSimple ADD COLUMN ColumnaNueva VARCHAR(20)

Agrega un campo ColumnaNueva de tipo carácter variable a la tabla TablaSimple.

ALTER TABLE TablaSimple ALTER COLUMN ColumnaNueva VARCHAR(10)

Modifica la longitud del campo ColumnaNueva de la tabla TablaSimple.

ALTER TABLE TablaSimple DROP COLUMN ColumnaNueva

Elimina el campo ColumnaNueva de la tabla TablaSimple.

6.3. Eliminar objetos

La sentencia **DROP** elimina un objeto de la base de datos. Al contrario que las sentencias CREATE y ALTER, todas las sentencias **DROP** tienen la misma sintaxis:

DROP tipo_objeto nombre

Donde tipo_objeto puede ser cualquier objeto de la base de datos.

DROP DATABASE Alumnos

DROP TABLE Productos

DROP INDEX MiIndice

DROP VIEW Suspensos

La sentencia TRUNCATE elimina el contenido de una tabla dejándola vacía de filas sin modificar su estructura. La sintaxis es la siguiente:

TRUNCATE tabla;

7. Conceptos de interés

7.1. Variables

Las variables se identifican por el prefijo @; por ejemplo, @miVariable. Tienen dos niveles de ámbito: local y global, identificando las variables globales con una doble@: @@VERSION.

- **Variables locales**

  Las variables locales se crean utilizando la sentencia **DECLARE**, con la siguiente sintaxis:

  DECLARE @variable_local tipo_datos

  Se pueden crear varias variables locales con una sola instrucción **DECLARE** separándolas con comas:

  DECLARE @var1 INT, @var2 INT

  Cuando se crea una variable local, inicialmente tiene el valor NULL. Puede asignar un valor a una variable de las siguientes formas:

  — Utilizando el comando **SET** con una constante o expresión:

    SET @miVariableChar = 'Hola, mundo'

  — Utilizando el comando **SELECT** con una constante o expresión:

    SELECT @miVariableChar = 'Hola, mundo'

  — Utilizando el comando **SELECT** con otra sentencia **SELECT**:

    SELECT @miVariableChar = MAX(OilName) FROM Oils

  Observe que en la tercera forma (el SELECT con otro SELECT), el operador de asignación (=) reemplaza a la segunda palabra reservada SELECT; no se repite.

- **Variables globales**

  Las variables globales, identificadas con un doble signo @ (@@VERSION) las proporciona SQL y el usuario no puede crearlas. Existen docenas de variables globales. La mayoría de ellas proporcionan información sobre el estado actual de SQL.

- **Utilizar variables**

  Las variables pueden utilizarse en expresiones a lo largo y ancho del lenguaje SQL. En cualquier caso, no pueden utilizarse en lugar de un nombre de objeto o palabra reservada. Así las siguientes sentencias son correctas:

  DECLARE @elAceite CHAR(20)

  SET @elAceite = 'Basil'

  -- Se ejecutará este comando

  SELECT OilName, Descripction FROM Oils WHERE OilName = @elAceite

Sin embargo, las siguientes sentencias SELECT provocarán errores:

```
DECLARE @elComando CHAR(10), @elCampo CHAR(10)
SET @elComando = 'SELECT'
SET @elCampo = 'OilName'
-- Este comando fallará
@elComando * from Oils
-- Igual que éste
SELECT @elCampo FROM Oils
```

7.2. Control de ejecución

A menos que especifique lo contrario, SQL procesa las sentencias desde el comienzo del script hasta el final, pasando por todas ellas. Sin embargo, unas veces interesará ejecutar una instrucción solo si son verdaderas ciertas condiciones y, otras veces, que una instrucción se ejecute un número de veces, o se repita hasta que se cumpla alguna condición.

Los comandos de flujo de SQL le proporcionan la posibilidad de controlar la ejecución de esta forma.

Cuando comience a manipular el modo en que SQL ejecuta las instrucciones, es conveniente tratar un conjunto de sentencias en bloque. SQL se lo permite mediante la pareja de comando BEGIN...END.

Escribir el comando BEGIN tras cualquier comando de control de flujo provoca que SQL aplique el comando a todas las sentencias entre el BEGIN y su correspondiente END.

- **Procesamiento condicional**

  IF...ELSE

  La sentencia IF es la más sencilla entre los comandos de flujo condicionales. Si la expresión lógica que sigue al comando IF se evalúa a TRUE, se ejecutarán la sentencia o bloque de sentencias que lo siguen. Si la expresión lógica se evalúa a FALSE, se salta la sentencia o bloque de sentencias que los sigan.

  El comando opcional ELSE le permite especificar una sentencia o grupo de sentencias a ejecutar solo si la expresión lógica se evalúa a FALSE.

```
DECLARE @primeraLetra CHAR(2)

SELECT @primeraLetra = LEFT(MIN(OilName), 1) FROM Oils

IF @primeraLetra = 'A'

    PRINT 'Es una A'

ELSE

    PRINT 'No es una A'
```

CASE

En la mayoría de los lenguajes de programación, CASE es una forma sofisticada de la sentencia IF que le permite especificar múltiples expresiones lógicas en una única sentencia. En SQL, CASE es una función, no un comando. No se utiliza por sí mismo como IF; en su lugar, se utiliza como parte de una sentencia SELECT o UPDATE.

Las sentencias que incluyen CASE pueden hacerlo en una de sus dos formas sintácticas, dependiendo de si la expresión a evaluar cambia. La forma más simple asume que la expresión lógica a evaluar siempre tiene la siguiente forma:

Valor = expresión

El valor puede ser tan complejo como quiera. Puede utilizar una constante, un nombre de columna o una expresión compleja, o cualquier cosa que necesite. El operador de comparación siempre es la igualdad. La sintaxis simple de CASE es:

```
CASE valor

    WHEN expresión_uno THEN resultado_expresión_uno

    WHEN expresión_dos THEN resultado_expresión_dos

...

    WHEN expresión_n THEN resultado_expresión_n

    [ELSE resultado_expresión_else]

END
```

En esta forma del CASE, se obtiene solo el resultado_expresión si la expresión que sigue a la palabra clave WHEN es lógicamente igual al valor especificado. Puede tener cualquier número de cláusulas WHEN en la expresión.
La cláusula ELSE es opcional y actúa como un resultado "comodín" (se ejecuta solo si todas las cláusulas WHEN se evalúan a FALSE).

Comparar un valor contra varios valores diferentes es extremadamente común, pero en algunas ocasiones necesitará más flexibilidad. En este caso, puede utilizar la llamada sintaxis CASE de búsqueda, con esta forma:

```
CASE
WHEN expresión_lógica_uno THEN resultado_expresión_uno
WHEN expresión_ lógica_dos THEN resultado_expresión_dos
...
WHEN expresión_ lógica_n THEN resultado_expresión_n
[ELSE resultado_expresión_else]
END
```

Utilizar un CASE simple:

```
SELECT OilName,
CASE PlantPartID
    WHEN 1 THEN 'Uno'
    WHEN 2 THEN 'Dos'
    WHEN 3 THEN 'Tres'
    WHEN 7 THEN 'Siete'
    WHEN 8 THEN 'Ocho'
END AS Categoria
FROM Oils ORDER BY Categoria
```

Utilizar un CASE de búsqueda:

```
SELECT TOP 10 OilName, LatinName
CASE
    WHEN LEFT(OilName,1) = 'B' THEN 'Nombre B'
    WHEN LEFT(LatinName,1) = 'C' THEN 'Nombre Latino C'
    ELSE 'Ninguno de los dos'
FROM Oils ORDER BY OilName
```

- **BUCLES**

  El último comando de control de flujo le permite hacer que una sentencia o bloque de sentencias se ejecuten hasta que se cumpla determinada condición.

  Bucle WHILE simple

  La forma más simple del bucle WHILE especifica una expresión lógica y una sentencia o bloque de sentencias. Las instrucciones se repiten hasta que la expresión lógica se evalúa a FALSE. Si la expresión lógica es FALSE la primera vez que se evalúa la sentencia WHILE, la instrucción o grupo de instrucciones no se ejecutará nunca.

```
DECLARE @contador INT
SET @contador = 1
WHILE @contador < 11
    BEGIN
    PRINT @contador
    SET @contador = @contador + 1
END
```

  Bucle WHILE complejo

  La sintaxis de la sentencia WHILE puede realizar procesos más complejos que el mostrado en el ejemplo anterior. La cláusula BREAK sale del bucle; la ejecución continúa por la sentencia que sigue a la cláusula END del bloque de sentencias WHILE. La cláusula CONTINUE devuelve la ejecución al comienzo del bucle, ocasionando que las sentencias que le siguen en el bloque de instrucciones no se ejecuten. Ambas sentencias BREAK y CONTINUE se suelen ejecutar condicionalmente, dentro de una instrucción IF.

  Utilizar WHILE...BREAK:

```
DECLARE @contador INT
SET @contador = 1
WHILE @contador < 25
    BEGIN
        PRINT @contador
        SET @contador = @contador + 1
        IF @contador > 10 BREAK
    END
```

```
Utilizar WHILE...CONTINUE:
DECLARE @contador INT
SET @contador = 0
WHILE @contador < 11
    BEGIN
        SET @contador = @contador + 1
        IF (@contador % 2) = 0 CONTINUE
        PRINT @contador
END
```

7.3. Transacciones

Una transacción es una serie de cambios en la base de datos que deben ser tratadas como una sola. En otras palabras, que se realicen todos o que no se haga ninguno, pues de lo contrario se podrían producir inconsistencias en la base de datos.

Cuando no se tiene activada una transacción el gestor de base de datos ejecuta inmediatamente cada sentencia INSERT, UPDATE o DELETE que se le encomiende, sin posibilidad de deshacer los cambio en caso de ocurrir cualquier percance. Cuando se activa una transacción los cambios que se van realizando quedan en un estado de provisionalidad hasta que se realiza un **COMMIT**, el cual hará definitivos los cambios o hasta realizar un **ROLLBACK** que deshará todos los cambios producidos desde que se inició la transacción.

7.4. Cursores

Una de las características que definen las bases de datos relacionales es que las operaciones se ejecutan sobre un conjunto de filas. Un conjunto puede estar vacío, o contener una sola fila, pero aún así se considera un conjunto. Esto es necesario y útil en operaciones relacionales, pero en algunas ocasiones puede no ser conveniente para las aplicaciones.

Por ejemplo, dado que no hay un modo de apuntar a una fila específica de un conjunto, mostrar cada vez una fila al usuario puede ser difícil.

Para manejar estas situaciones, SQL admite los cursores. Un cursor es un objeto que apunta a una fila específica dentro de un conjunto. Dependiendo de la naturaleza del cursor que cree, puede mover el cursor por el conjunto y modificar o borrar datos. Su sintaxis es:

DECLARE nombre-cursor **CURSOR FOR** especificación-consulta

```
DECLARE MiCursor CURSOR FOR
    SELECT num_emp, nombre, puesto, salario FROM Empleados
        WHERE num_dept = 'informatica'
```

Este comando es meramente declarativo, simplemente especifica las filas y columnas que se van a recuperar. La consulta se ejecuta cuando se abre o se activa el cursor.

- **Variables cursor**

  SQL permite declarar variables de tipo CURSOR. En este caso, la sintaxis estándar **DECLARE** no crea el cursor, debe explícitamente establecer (**SET**) la variable al cursor.

  DECLARE MiCursor CURSOR FOR

  SELECT OilName FROM Oils

  DECLARE @miVariableCursor CURSOR

  SET @miVariableCursor = MiCursor

- **Abrir un cursor**

  Al declarar un cursor crea el objeto cursor, pero no crea el conjunto de registros que lo manipulará. El conjunto de cursor no se crea hasta que abre el cursor. Para abrirlo o activarlo se utiliza el comando **OPEN**, la sintaxis es la siguiente:

  OPEN nombre_cursor

  Al abrir el cursor se evalúa la consulta que aparece en su definición, utilizando los valores actuales de cualquier parámetro referenciado en la consulta, para producir una colección de filas. El puntero se posiciona delante de la primera fila de datos (registro actual), esta sentencia no recupera ninguna fila.

- **Cerrar un cursor**

  Una vez ha terminado de utilizar un cursor, deberá cerrarlo. La sentencia **CLOSE** libera los recursos utilizados en mantener el conjunto de cursor. Este comando hace desaparecer el puntero sobre el registro actual. La sintaxis es:

  CLOSE nombre_cursor

- **Liberar un cursor**

  Para liberar un cursor, se utiliza la sentencia **DEALLOCATE**. Este comando borra el identificador del cursor o la variable cursor, pero no borra necesariamente el cursor. El cursor en sí mismo no se elimina hasta que todos los identificadores que lo referencian se hayan liberado o salgan fuera de su ámbito o se elimine el cursor.

  La sintaxis es:

  DEALLOCATE nombre_cursor

Por ejemplo:

```
-- Crea el cursor
DECLARE MiCursor CURSOR FOR SELECT * FROM Oils
-- Crea una variable de cursor
DECLARE @variableCursor CURSOR
-- Crea el conjunto de cursor
OPEN MiCursor
-- Asigna la variable al cursor
SET @variableCursor = MiCursor
-- Liberar el cursor
DEALLOCATE MiCursor
```

Después de liberar el cursor, el identificador MiCursor deja de estar asociado con el conjunto de cursor, pero dado que el conjunto de cursor aun está referenciado por la variable @variableCursor, el cursor y conjunto de cursor no se liberan. A menos que explícitamente libere también la variable cursor, el cursor y conjunto de cursor continuarán existiendo hasta que la variable salga de su ámbito o se elimine definitivamente el cursor.

- **Eliminar un cursor**

  Para eliminar el cursor se utiliza el comando **DROP CURSOR**. Su sintaxis es la siguiente:

  ```
  DROP CURSOR nombre_cursor
  ```

- **Manipular filas mediante un cursor**

  Los cursores no serían interesantes si no pudiera hacer algo con ellos. Existen tres comandos diferentes para trabajar con cursores: **FETCH, UPDATE y DELETE**.

  El comando **FETCH** recupera una fila específica del conjunto del cursor. En su forma más simple el comando **FETCH** tiene la sintaxis:

  ```
  FETCH cursor_o_variable
  ```

  Este comando obtiene la fila en la cual está posicionado el cursor (la fila actual).

  En lugar de obtener una fila directamente, el comando FETCH le permite almacenar los valores obtenidos de las columnas en variables. Para almacenar los resultados del FETCH en una variable, utilice la siguiente sintaxis:

  ```
  FETCH cursor_o_variable INTO lista_variables
  ```

La lista_variables es una lista separada por comas de identificadores de variable. Debe declarar las variables antes de ejecutar el comando FETCH. La lista_variables debe incluir una variable por cada columna de la sentencia SELECT que define el cursor, y los tipos de datos de las variables deben ser igual o compatibles con los tipos de datos de la columna.

```
— Crea el cursor y algunas variables

DECLARE CursorSimple CURSOR FOR SELECT OilName, Latíname FROM Oils

DECLARE @Nombre CHAR(20), @NombreLatin CHAR(50)

— Crea el conjunto de cursor

OPEN CursorSimple

— Recupera los valores en variables

FETCH CursorSimple INTO @Nombre, @NombreLatin

— Muestra los resultados

PRINT RTRIM(@Nombre) + ‘ es el nombre’

PRINT RTRIM(@NombreLatin) + ‘ es el nombre latín’

— Cierra el conjunto de resultados

CLOSE CursorSimple

— Libera el cursor

DEALLOCATE CursorSimple
```

En el ejemplo anterior hemos utilizado la sentencia **FETCH** para obtener la fila actual. La sintaxis de la sentencia **FETCH** proporciona también un número de palabras reservadas para especificar una fila diferente. Cuando utiliza una de estas palabras clave la sentencia **FETCH** obtendrá la fila especificada y convierte esa fila en la actual.

Tres palabras clave le permiten especificar una posición absoluta en el conjunto de cursor. Las palabras reservadas **FIRST** y **LAST** obtienen la primera y última fila respectivamente, mientras que **ABSOLUTE n** especifica una fila **n** filas desde el comienzo (si n es positivo) o el final (si n es negativo) del conjunto de cursor. Puede expresar el valor de **n** como una constante (3) o una variable (@laFila).

```
— Crea el cursor y algunas variables

DECLARE CursorSimple CURSOR FOR SELECT OilName FROM Oils

DECLARE @Nombre CHAR(20)
```

```
-- Crea el conjunto de cursor

OPEN CursorSimple

-- Recupera la primera fila en la variable

FETCH FIRST FROM CursorSimple INTO @Nombre

-- Muestra los resultados

PRINT RTRIM(@Nombre) + 'es el primer nombre'

-- Recupera la quinta fila

FETCH ABSOLUTE 5 FROM CursorSimple INTO @Nombre

-- Muestra los resultados

PRINT RTRIM(@Nombre) + 'es el quinto nombre'

-- Cierra el conjunto de resultados

CLOSE CursorSimple

-- Libera el cursor

DEALLOCATE CursorSimple
```

Además de las palabras clave que le permiten recuperar una fila basándose en su posición absoluta, la sentencia **FETCH** le proporciona tres palabras clave que le permiten recuperar una fila basándose en su posición relativa con respecto a la fila actual. **FETCH NEXT** obtiene la siguiente fila, **FETCH PRIOR** obtiene la fila anterior, y **FETCH RELATIVE** n obtiene una fila n filas desde la fila actual. Como **FETCH ABSOLUTE** *n*, **FETCH RELATIVE** *n* puede especificar el número de filas antes de la fila actual, si n es negativo, o después de la fila actual, si n es positivo.

```
DECLARE CursorSimple CURSOR FOR SELECT OilName FROM Oils

DECLARE @Nombre CHAR(20)

OPEN CursorSimple

-- Recupera la fila en la variable

FETCH FIRST FROM CursorSimple INTO @Nombre

-- Muestra los resultados

PRINT RTRIM(@Nombre) + 'es el primer nombre'
```

```
-- Recupera la siguiente fila
FETCH RELATIVE 1 FROM CursorSimple INTO @Nombre
-- Muestra los resultados
PRINT RTRIM(@Nombre) + 'es el siguiente nombre'
CLOSE CursorSimple
DEALLOCATE CursorSimple
```

- **Monitorizar un cursor**

@@FETCH_STATUS obtiene información sobre el último comando FETCH ejecutado.

VALOR DE RETORNO	SIGNIFICADO
0	El comando FETCH se ejecutó correctamente.
-1	El comando FETCH falló.
-2	La fila leída desapareció.

```
--Abrir un cursor y recorrerlo
DECLARE EmployeeCursor CURSOR FOR SELECT LastName, FirstName
                                  FROM Employees
                                  WHERE LastName LIKE 'B*'
OPEN EmployeeCursor
FETCH NEXT FROM EmployeeCursor
WHILE @@FETCH_STATUS = 0
BEGIN
FETCH NEXT FROM EmployeeCursor
END
CLOSE EmployeeCursor
DEALLOCATE EmployeeCursor
```

```
--Recorrer un cursor guardando los valores en variables
DECLARE @au_lname VARCHAR(40)
DECLARE @au_fname VARCHAR(20)
DECLARE authors_cursor CURSOR FOR SELECTau_lname, au_fname
                                  FROM authors
                                  WHERE au_lname LIKE "B*"
                                  ORDER BY au_lname, au_fname
OPEN authors_cursor
FETCH NEXT FROM authors_cursor INTO @au_lname, @au_fname
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT "Author: " + @au_fname + " " + @au_lname
    FETCH NEXT FROM authors_cursor INTO @au_lname, @au_fname
END
CLOSE authors_cursor
DEALLOCATE authors_cursor
```

7.5. Procedimientos almacenados *(Store procedure)*

Es una colección de sentencias SQL precompiladas que pueden devolver y tomar parámetros, algo así como un fichero de ejecución por lotes. Son lotes de sentencias almacenadas en el servidor.

Los procedimientos almacenados no son la única forma de ejecutar sentencias SQL. Hemos visto los script SQL, pero los procedimientos almacenados se ejecutan de forma optimizada, dando lugar a una ejecución más rápida.

Los procedimientos almacenados proporcionan dos métodos de comunicación con procesos externos: parámetros y valores de retorno.

Los parámetros son una clase especial de variable local declarada como parte del procedimiento almacenado. Puede utilizar parámetros para pasar información al procedimiento almacenado (parámetros de entrada) o recibir valores desde el procedimiento almacenado (parámetros de salida).

Un valor de retorno es similar al resultado de una función y pueden asignarse a una variable local de la misma forma. Los valores de retorno siempre

son enteros. Pueden utilizarse teóricamente para devolver cualquier resultado, pero por convención se utilizan para devolver el estado de la ejecución del procedimiento almacenado. Por ejemplo, un procedimiento almacenado podría devolver 0 si todo fue bien, o -1 si hubo algún error. Los procedimientos almacenados más sofisticados pueden devolver valores de retorno diferentes para indicar la naturaleza del error encontrado.

Es importante no confundir los parámetros y los códigos de retorno con cualquier otro conjunto de resultados que podría devolver un procedimiento almacenado. Un procedimiento almacenado puede contener cualquier número de sentencias SELECT que devolverían conjuntos de resultados. No tiene que utilizar un parámetro para recibirlos, se devuelven a la aplicación de forma independiente.

Los procedimientos almacenados vienen de dos formas: los procedimientos de sistema creados por el propio SQL (todos ellos comienzan con los caracteres sp_) que nos devuelven información acerca del sistema, sus tablas, contenidos y estructura de los campos, almacenamiento de los datos, etc. (por ejemplo, sp_tables, sp_columns, sp_spaceused, sp_who, sp_helpdb, etc.) y los procedimientos almacenados definidos por el usuario.

- **Utilizar procedimientos almacenados**

  Se utiliza la sentencia **EXECUTE** para invocar un procedimiento almacenado tanto de sistema como definido por el usuario. Si el procedimiento almacenado no tiene parámetros o si no devuelve ningún resultado, la sintaxis es muy simple:

  EXECUTE nombre_procedimiento

  Por ejemplo:

  EXECUTE sp_helpdb

  Si el procedimiento almacenado acepta parámetros de entrada puede indicárselos por posición o por nombre. Para indicar parámetros por posición, simplemente lístelos después del nombre del procedimiento almacenado, separando cada parámetro individual con comas:

  EXECUTE nombre_procedimiento parámetro [, parámetro...]

  Por ejemplo:

  EXECUTE sp_dboption 'MiBasedeDatos', 'read only'

- **Crear procedimientos almacenados**

  Los procedimientos almacenados se crean utilizando la sentencia CREATE PROCEDURE. Su sintaxis es:

  CREATE PROCEDURE nombre_procedimiento [lista_parámetros]

  AS sentencias_procedimiento

Cada parámetro en la lista_parámetros tiene la estructura:

@nombre_parámetro tipo_dato [= valor_defecto] [OUTPUT]

Los nombres de parámetros comienzan siempre con @, como una variable local. De hecho, los parámetros son variables locales; solo son visibles dentro del procedimiento almacenado.

El valor_defecto es el valor que utilizará el procedimiento almacenado si el usuario no especifica el valor del parámetro de entrada en la llamada al procedimiento almacenado. La palabra reservada OUTPUT, también opcional, define los parámetros que se devolverán al script de llamada.

Las sentencias_procedimiento que siguen al AS en la sentencia CREATE definen las acciones a ejecutar cuando se llame al procedimiento almacenado.

Los procedimientos almacenados pueden llamar a otros procedimientos almacenados, en un proceso conocido como **anidamiento**.

Crear un procedimiento almacenado simple

CREATE PROCEDURE SPSimple AS SELECT OilName, LatinName FROM Oils

Para ejecutar este procedimiento almacenado:

EXECUTE SPSimple

Crear un procedimiento almacenado con un parámetro de entrada

CREATE PROCEDURE SPInput @OilName CHAR(50)

AS SELECT OilName, LatinName FROM Oils WHERE OilName = @OilName

Para ejecutar este procedimiento almacenado:

EXECUTE SPInput 'Basil'

Crear un procedimiento almacenado con un valor por defecto

CREATE PROCEDURE SPDefault @OilName CHAR(50) = 'Fennel'

AS SELECT OilName, LatinName FROM Oils WHERE OilName = @OilName

Para ejecutar este procedimiento almacenado:

EXECUTE SPDefault

Crear un procedimiento almacenado con un parámetro de salida

CREATE PROCEDURE SPOutput @VarSalida CHAR(6) OUTPUT

AS SET @VarSalida = 'Salida'

Para ejecutar este procedimiento almacenado:

```
DECLARE @miSalida CHAR(6)

EXECUTE SPOutput @miSalida OUTPUT

SELECT @miSalida
```

Los valores de retorno se definen utilizando la sentencia **RETURN**, que tiene la forma:

RETURN (int)

En la sentencia RETURN, int es un valor entero. Como vimos anteriormente, los valores de retorno se utilizan la mayoría de las veces para devolver el estado de ejecución de un procedimiento almacenado, con o indicando ejecución correcta, y cualquier otro número indicando un error. Puede comprobar los errores utilizando la variable global @@ERROR, que devuelve el estado de ejecución del comando SQL más reciente: 0 para ejecución correcta, o un número distinto de 0 indicando el error que ha ocurrido.

Crear un procedimiento almacenado con un parámetro de entrada

```
CREATE PROCEDURE SPError AS

-- Crea una variable para almacenar el código de error

DECLARE @codigoRetorno INT

SELECT OilName, LatinName FROM Oils

-- Atrapa cualquier error

SET @codigoRetorno = @@ERROR

RETURN (@codigoRetorno)
```

Para ejecutar este procedimiento almacenado:

```
DECLARE @elError INT

EXECUTE @elError = SPError

SELECT @elError AS 'Valor retorno'
```

7.6. Desencadenadores *(triggers)*

Un desencadenador o trigger es un tipo especial de procedimiento almacenado que se ejecuta desatendidamente y automáticamente cuando un usuario realiza una acción con la tabla de una base de datos que lleve asociado este trigger. Se pueden crear triggers para las sentencias de SQL INSERT, UPDATE Y DELETE.

SQL impone algunas restricciones en el proceso que pueden ejecutar los desencadenadores. No puede CREATE, ALTER o DROP una base de datos utilizando un desencadenador; ni restaurar una base de datos o archivo de transacciones; y no puede ejecutar ciertas operaciones que cambien la configuración de SQL.

- **Utilizar el comando CREATE TRIGGER**

  Los desencadenadores se crean utilizando la sentencia CREATE TRIGGER. Su sintaxis es:

  CREATE TRIGGER nombre_desencadenador

  ON tabla **FOR** lista_comandos **AS** sentencias_sql

  La Lista de comandos es cualquier combinación de los comandos INSERT, UPDATE o DELETE. Si indica más de un comando, sepárelos con comas.

  Las sentencias_sql que siguen a la palabra reservada AS definen el proceso a ejecutar por el desencadenador, igual que en los procedimientos almacenados excepto que un desencadenador no admite parámetros.

  ```
  CREATE TRIGGER afterUpdate
      ON Oils
      FOR UPDATE AS INSERT INTO TriggerMessages (TriggerName, MessageText)
      VALUES ('afterUpdate', 'enviado por el desencadenador afterUpdate')
  ```

- **Utilizar la función UPDATE**

  SQL proporciona una función especial, **UPDATE**, que puede utilizarse dentro de un desencadenador para comprobar si se ha modificado una columna específica de una fila. La sintaxis es:

  UPDATE (nombre_columna)

  Devolverá TRUE si se ha modificado el valor de los datos para la columna especificada para cualquiera de los comandos INSERT o UPDATE.

  ```
  CREATE TRIGGER UpdateFunc
    ON Oils
    FOR UPDATE AS
        IF UPDATE (Descripcion)
            INSERT INTO TriggerMessages (TriggerName, MessageText)
            VALUES ('UpdateFunc', 'Descripción modificada')
  ```

```
IF UPDATE (OilName)
        INSERT INTO TriggerMessages (TriggerName, MessageText)
        VALUES ('UpdateFunc', 'OilName modificado')
```

- **Utilizar las tablas Inserted y Deleted**

  SQL crea dos tablas para ayudarle a manipular los datos durante la ejecución del desencadenador. Las tablas **Inserted** y **Deleted** son tablas temporales residentes en memoria que contienen los valores de las filas afectadas por el comando que invocó al desencadenador.

  Cuando se llama un desencadenador desde un comando DELETE, la tabla **Deleted** contendrá las filas que se borraron de la tabla. Para un comando INSERT, la tabla **Inserted** contendrá una copia de las nuevas filas. Físicamente, una sentencia UPDATE es un DELETE seguido de un INSERT, así que la tabla **Deleted** contendrá los valores antiguos, y la tabla **Inserted** los valores nuevos. Puede hacer referencia a los contenidos de estas tablas desde dentro del desencadenador pero no modificarlas.

7.7. Bloqueos

Los bloqueos nos proporcionan información acerca de qué recursos individuales están bloqueados. Los bloqueos en filas leídas o modificadas durante una transacción se utilizan para evitar que varias transacciones utilicen simultáneamente los mismos recursos y puedan estropear los datos. Por ejemplo, si una transacción mantiene un bloqueo exclusivo en una fila de una tabla ninguna otra transacción podrá modificar esa fila hasta que se libere el bloqueo.

Para lograr estos objetivos el SQL tiene cuatro modos de aislamiento *(isolation levels).*

- **¿Qué se bloquea?**

  SQL dispone de varios niveles de bloqueo lo que permite a una transacción bloquear diferentes tipos de recursos. Para minimizar el costo de los bloqueos y aumentar la simultaneidad, SQL bloquea automáticamente los recursos en el nivel apropiado para la tarea. El bloqueo de menor granularidad, como es el caso de las filas, aumenta la simultaneidad. Sin embargo, se produce una sobrecarga mayor porque cuantas más filas se bloquean, más bloqueos se deben mantener y esto requiere que nuestro servidor utilice recursos adicionales del sistema. Bloquear con una granularidad mayor, como las tablas, es costoso en términos de simultaneidad debido a que bloquear una tabla completa restringe los accesos de las demás transacciones a cualquier parte de la tabla, pero produce una sobrecarga menor (menos recursos utilizados) debido a que se mantienen menos bloqueos.

Veamos qué tipo de recursos puede bloquear SQL:

RID	Identificador de fila. Se utiliza para bloquear una sola fila de una tabla.
CLAVE	Bloqueo de una fila en un índice. Se utilizan únicamente en transacciones que operan en el nivel de transacción serializable.
PÁGINA	La página de datos o página de índices (8 kb).
EXTENSIÓN	Grupo contiguo de ocho páginas de datos o páginas de índice.
TABLA	Tabla completa, con todos los datos e índices.
BASE DE DATOS	Base de datos.

Estos recursos se pueden bloquear con diferentes modos de bloqueo que determinarán cómo transacciones simultáneas pueden tener acceso a esos recursos.

- **¿Cómo se bloquea?**

Una vez que hemos visto los recursos sobre los que SQL mantiene bloqueos, vamos a ver los tipos de bloqueo de los que disponemos:

— **SHARED (S):** Compartido.

— **UPDATE (U):** Actualizar.

— **EXCLUSIVE (X):** Exclusivo.

— **BULK UPDATE (BU):** Actualización masiva.

— **SCHEMA:** Esquema.

Estos tipos de bloqueo son los que SQL utiliza para alcanzar los cuatro niveles de aislamiento. El nivel de aislamiento estándar es el **COMMITED READ** aunque con la siguiente instrucción podemos cambiar este comportamiento:

```
SET TRANSACTION ISOLATION LEVEL
{ COMMITTED READ | UNCOMMITTED READ | REPEATABLE READ
    | SERIALIZABLE }
```

- **Bloqueo compartido**

  Los bloqueos compartidos (**SHARED**, S) se utilizan para operaciones de lectura de datos.

  Durante los bloqueos compartidos (S) varias transacciones concurrentes pueden leer un recurso pero no pueden modificar ese recurso mientras ese bloqueo compartido exista. Si no hemos cambiado el nivel de aislamiento de nuestra transacción, cosa que en general no haremos, en cuanto se haya producido la lectura de los datos los recursos bloqueados quedan libres.

  Si hemos colocado el nivel de aislamiento de nuestra transacción en REPEATABLE READ o en SERIALIZABLE, el recurso quedará bloqueado hasta que termine la transacción en la que estamos trabajando.

- **Bloqueo de actualización**

  Los bloqueos de actualización (**UPDATE**, U) se utilizan cuando el SQL tiene intención de modificar una fila o una página y posteriormente promociona este bloqueo a un bloqueo exclusivo (X). Este tipo de bloqueos se utiliza para evitar el problema de los interbloqueos.

  Veámoslo con un ejemplo:

  Supongamos que tenemos dos transacciones que intentarán actualizar la misma fila. Cada una de nuestras transacciones obtendrá un bloqueo compartido (S) sobre la fila, la leerá y posteriormente intentará obtener sobre esa fila un bloqueo exclusivo. Pero la obtención de un bloqueo exclusivo no es compatible con la existencia de un bloqueo compartido, así que la primera transacción esperará a que la segunda libere su bloqueo compartido, y la segunda espera a que la primera libere su bloqueo compartido para obtener uno exclusivo. Este es un ejemplo típico de interbloqueo.

  Para evitar esta situación tenemos este tipo de bloqueos de actualización (U).

  Dos transacciones no pueden obtener simultáneamente un bloqueo de actualización (U) para un recurso, y si una transacción modifica un recurso, el bloqueo de actualización (U) se convierte en bloqueo exclusivo (X). En caso contrario, el bloqueo se convierte en bloqueo de modo compartido.

- **Bloqueos exclusivos**

  Los bloqueos exclusivos (**EXCLUSIVE**, X) se utilizan para realizar modificaciones con sentencias INSERT, UPDATE y DELETE. La principal característica de este bloqueo es que otras transacciones no pueden leer ni modificar los registros bloqueados. Asimismo si hay un bloqueo compartido(S) sobre un recurso ninguna transacción puede obtener un bloqueo exclusivo sobre ese recurso.

- **Bloqueos de actualización masiva**

  Los bloqueos de actualización masiva (**BULK UPDATE**, BU) se utilizan durante la inserción masiva de datos en una tabla. Este tipo de bloqueos permiten que se copien datos concurrentemente en la misma tabla mientras que se impide que otros procesos accedan a esa tabla.

- **Bloqueos de Esquema**

  Los bloqueos de esquema (**SCHEMA**) se usan cuando se realiza una operación que modifica el esquema de nuestra base de datos.

  Por ejemplo al ejecutar una sentencia DDL como ALTER TABLE se adquiere un bloqueo de modificación de esquema en la tabla para garantizar que ninguna otra conexión haga referencia ni siquiera a los metadatos de la tabla durante el cambio.