SENTENCIAS BÁSICAS DEL LENGUAJE SQL

Es de eso de lo que trata el Structured Query Language que no es más que un lenguaje estándar de comunicación con bases de datos. Hablamos por tanto de un lenguaje normalizado que nos permite trabajar con cualquier tipo de lenguaje (ASP o PHP) en combinación con cualquier tipo de base de datos (MS Access, SQL Server, MySQL).

El hecho de que sea estándar no quiere decir que sea idéntico para cada base de datos. En efecto, determinadas bases de datos implementan funciones específicas que no tienen necesariamente que funcionar en otras.

Aparte de esta universalidad, el SQL posee otras dos características muy apreciadas. Por una parte, presenta una potencia y versatilidad notables que contrasta, por otra, con su accesibilidad de aprendizaje.

Diferentes tipos campos empleados en las bases de datos

Como sabemos una base de datos está compuesta de tablas donde almacenamos registros catalogados en función de distintos campos (características).
Un aspecto previo a considerar es la naturaleza de los valores que introducimos en esos campos. Dado que una base de datos trabaja con todo tipo de informaciones, es importante especificarle qué tipo de valor le estamos introduciendo de manera a, por un lado, facilitar la búsqueda posteriormente y por otro, optimizar los recursos de memoria.

Cada base de datos introduce tipos de valores de campo que no necesariamente están presentes en otras. Sin embargo, existe un conjunto de tipos que están representados en la totalidad de estas bases. Estos tipos comunes son los siguientes:

Alfanuméricos	Contienen cifras y letras. Presentan una longitud limitada (255 caracteres)
Numéricos	Existen de varios tipos, principalmente, enteros (sin decimales) y reales (con decimales).
Booleanos	Poseen dos formas: Verdadero y falso (Sí o No)
Fechas	Almacenan fechas facilitando posteriormente su explotación. Almacenar fechas de esta forma posibilita ordenar los registros por fechas o calcular los días entre una fecha y otra...
Memos	Son campos alfanuméricos de longitud ilimitada. Presentan el inconveniente de no poder ser indexados (veremos más adelante lo que esto quiere decir).
Autoincrementables	Son campos numéricos enteros que incrementan en una unidad su valor para cada registro incorporado. Su utilidad resulta más que evidente: Servir de identificador ya que resultan exclusivos de un registro.

Mostramos unas tablas con todos los tipos de datos que hay en SQL.

Los tipos de datos SQL se clasifican en 13 tipos de datos primarios y de varios sinónimos válidos reconocidos por dichos tipos de datos. Los tipos de datos primarios son:

Tipo de Datos	**Longitud**	**Descripción**
BINARY	1 byte	Para consultas sobre tabla adjunta de productos de bases de datos que definen un tipo de datos Binario.
BIT	1 byte	Valores Si/No ó True/False
BYTE	1 byte	Un valor entero entre 0 y 255.
COUNTER	4 bytes	Un número incrementado automáticamente (de tipo Long)
CURRENCY	8 bytes	Un entero escalable entre 922.337.203.685.477,5808 y 922.337.203.685.477,5807.
DATETIME	8 bytes	Un valor de fecha u hora entre los años 100 y 9999.
SINGLE	4 bytes	Un valor en punto flotante de precisión simple con un rango de - 3.402823*1038 a -1.401298*10-45 para valores negativos, 1.401298*10- 45 a 3.402823*1038 para valores positivos, y 0.

DOUBLE	8 bytes	Un valor en punto flotante de doble precisión con un rango de -1.79769313486232*10308 a -4.94065645841247*10-324 para valores negativos, 4.94065645841247*10-324 a 1.79769313486232*10308 para valores positivos, y 0.
SHORT	2 bytes	Un entero corto entre -32,768 y 32,767.
LONG	4 bytes	Un entero largo entre -2,147,483,648 y 2,147,483,647.
LONGTEXT	1 byte por carácter	De cero a un máximo de 1.2 gigabytes.
LONGBINARY	Según se necesite	De cero 1 gigabyte. Utilizado para objetos OLE.
TEXT	1 byte por carácter	De cero a 255 caracteres.

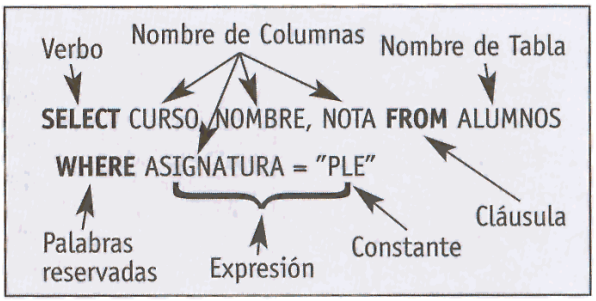
La siguiente tabla recoge los sinónimos de los tipos de datos definidos:

Tipo de Dato	**Sinónimos**
BINARY	VARBINARY
BIT	BOOLEAN LOGICAL LOGICAL1 YESNO
BYTE	INTEGER1
COUNTER	AUTOINCREMENT
CURRENCY	MONEY
DATETIME	DATE TIME TIMESTAMP
SINGLE	FLOAT4 IEEESINGLE REAL
DOUBLE	FLOAT FLOAT8 IEEEDOUBLE NUMBER NUMERIC
SHORT	INTEGER2 SMALLINT
LONG	INT INTEGER INTEGER4
LONGBINARY	GENERAL OLEOBJECT
LONGTEXT	LONGCHAR MEMO NOTE
TEXT	ALPHANUMERIC CHAR - CHARACTER STRING - VARCHAR
VARIANT (No Admitido)	VALUE

	SENTENCIA	DESCRIPCIÓN
	Manipulación de datos	
	SELECT	Recupera datos de la base de datos.
	INSERT	Añade nuevas filas de datos a la base de datos.
	DELETE	Suprime filas de datos de la base de datos.
DML	UPDATE	Modifica datos existentes en la base de datos.
	Definición de datos	
	CREATE TABLE	Añade una nueva tabla a la base de datos.
	DROP TABLE	Suprime una tabla de la base de datos.
	ALTER TABLE	Modifica la estructura de una tabla existente.
	CREATE VIEW	Añade una nueva vista a la base de datos.
	DROP VIEW	Suprime una vista de la base de datos.
	CREATE INDEX	Construye un índice para una columna.
	DROP INDEX	Suprime el índice para una columna.
	CREATE SYNOYM	Define un alias para un nombre de tabla.
DDL	DROP SYNONYM	Suprime un alias para un nombre de tabla.
	Control de acceso	
	GRANT	Concede privilegios de acceso a usuarios.
	REVOKE	Suprime privilegios de acceso a usuarios
	Control de transacciones	
	COMMIT	Finaliza la transacción actual.
DCL	ROLLBACK	Aborata la transacción actual.
	SQL Programático	
	DECLARE	Define un cursor para una consulta.
	OPEN	Abre un cursor para recuperar resultados de consulta.
	FETCH	Recupera una fila de resultados de consulta.
PLSQL	CLOSE	Cierra un cursor.

Componentes sintácticos

La mayoría de sentencias SQL tienen la misma estructura.
Todas comienzan por un verbo (select, insert, update, create), a continuación le sigue una o más cláusulas que nos dicen los datos con los que vamos a operar (from, where), algunas de estas son opcionales y otras obligatorias como es el caso del from.



Explicamos la manera de crear tablas a partir de sentencias SQL. Definimos los tipos de campos principales y la forma de especificar los índices.

En general, la mayoría de las bases de datos poseen potentes editores de bases que permiten la creación rápida y sencilla de cualquier tipo de tabla con cualquier tipo de formato.
Sin embargo, una vez la base de datos está alojada en el servidor, puede darse el caso de que queramos introducir una nueva tabla ya sea con carácter temporal (para gestionar un carrito de compra por ejemplo) o bien permanente por necesidades concretas de nuestra aplicación.

En estos casos, podemos, a partir de una sentencia SQL, crear la tabla con el formato que deseemos lo cual nos puede ahorrar más de un quebradero de cabeza.

Este tipo de sentencias son especialmente útiles para bases de datos como Mysql, las cuales trabajan directamente con comandos SQL y no por medio de editores.

Para crear una tabla debemos especificar diversos datos: El nombre que le queremos asignar, los nombres de los campos y sus características. Además, puede ser necesario especificar cuáles de estos campos van a ser índices y de qué tipo van a serlo.

La sintaxis de creación puede variar ligeramente de una base de datos a otra ya que los tipos de campo aceptados no están completamente estandarizados.

A continuación os explicamos someramente la sintaxis de esta sentencia y os proponemos una serie de ejemplos prácticos:

Sintaxis

```
Create Table nombre_tabla
(
nombre_campo_1 tipo_1
nombre_campo_2 tipo_2
nombre_campo_n tipo_n
Key(campo_x,...)
)
```

Pongamos ahora como ejemplo la creación de la tabla pedidos que hemos empleado en capítulos previos:

```
Create Table pedidos
(
id_pedido INT(4) NOT NULL AUTO_INCREMENT,
id_cliente INT(4) NOT NULL,
id_articulo INT(4)NOT NULL,
fecha DATE,
cantidad INT(4),
total INT(4), KEY(id_pedido,id_cliente,id_articulo)
)
```

En este caso creamos los campos *id* los cuales son considerados de tipo entero de una longitud especificada por el número entre paréntesis. Para *id_pedido* requerimos que dicho campo se incremente automáticamente (AUTO_INCREMENT) de una unidad a cada introducción de un nuevo registro para, de esta forma, automatizar su creación. Por otra parte, para evitar un mensaje de error, es necesario requerir que los campos que van a ser definidos como índices no puedan ser nulos (NOT NULL).
El campo *fecha* es almacenado con formato de fecha (DATE) para permitir su correcta explotación a partir de las funciones previstas a tal efecto.
Finalmente, definimos los índices enumerándolos entre paréntesis precedidos de la palabra KEY o INDEX.

Del mismo modo podríamos crear la tabla de *artículos* con una sentencia como ésta:

```
Create Table articulos
(
id_articulo INT(4) NOT NULL AUTO_INCREMENT,
titulo VARCHAR(50),
autor VARCHAR(25),
editorial VARCHAR(25),
precio REAL,
KEY(id_articulo)
)
```

En este caso puede verse que los campos alfanuméricos son introducidos de la misma forma que los numéricos. Volvemos a recordar que en tablas que tienen campos comunes es de vital importancia definir estos campos de la misma forma para el buen funcionamiento de la base.

Muchas son las opciones que se ofrecen al generar tablas. No vamos a tratarlas detalladamente pues sale de lo estrictamente práctico. Tan sólo mostraremos algunos de los tipos de campos que pueden ser empleados en la creación de tablas con sus características:

Tipo	Bytes	Descripción
INT o INTEGER	4	Números enteros. Existen otros tipos de mayor o menor longitud específicos de cada base de datos.
DOUBLE o REAL	8	Números reales (grandes y con decimales). Permiten almacenar todo tipo de número no entero.
CHAR	1/caracter	Alfanuméricos de longitud fija predefinida
VARCHAR	1/caracter+1	Alfanuméricos de longitud variable
DATE	3	Fechas, existen multiples formatos específicos de cada base de datos
BLOB	1/caracter+2	Grandes textos no indexables
BIT o BOOLEAN	1	Almacenan un bit de información (verdadero o falso)

Una base de datos en un sistema relacional está compuesta por un conjunto de tablas, que corresponden a las relaciones del modelo relacional.
En la terminología usada en SQL no se alude a las relaciones, del mismo modo que no se usa el término atributo, pero sí la palabra columna, y no se habla de tupla, sino de línea.

Creación de Tablas Nuevas

```
CREATE TABLE tabla (
campo1 tipo (tamaño) índice1,
campo2 tipo (tamaño) índice2,... ,
índice multicampo , ... )
```

En donde:

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. (Ver Tipos de Datos)
tamaño	Es el tamaño del campo sólo se aplica para campos de tipo texto.
índice1 índice2	Es una cláusula CONSTRAINT que define el tipo de índice a crear. Esta cláusula en 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 en 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,
  FechaNacimiento DATETIME
  )

CONSTRAINT
  IndiceGeneral
  UNIQUE (
     Nombre, Apellidos, FechaNacimiento
   )
```

(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 (50) y uno más llamado FechaNacimiento de tipo Fecha/Hora. También crea un índice único - no permite valores repetidos - formado por los tres campos.)

```
CREATE TABLE
    Empleados (
  IdEmpleado INTEGER CONSTRAINT IndicePrimario PRIMARY,
    Nombre TEXT,
  Apellidos TEXT,
  FechaNacimiento DATETIME
  )
```

(Crea una tabla llamada Empleados con un campo Texto de longitud predeterminada (50) llamado Nombre y otro igual llamado Apellidos, crea otro campo llamado FechaNacimiento de tipo Fecha/Hora y el campo IdEmpleado 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 si desea Crear ó Eliminar un índice de un único campo o si se trata de un campo multiíndice. Si se utiliza el motor de datos de Microsoft, sólo podrá utilizar esta cláusula con las bases de datos propias de dicho motor. 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 [,...])]}
```

En donde:

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 externos	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 CONTRAINT 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.

Indice	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, cada tabla sólo 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, predeterminado por valor, el motor Jet se comporta como si la clave principal de la tabla externa estuviera formada por los campos referenciados.

Creación de Índices

Si se utiliza el motor de datos Jet de Microsoft sólo se pueden crear índices en bases de datos del mismo motor. La sintaxis para crear un índice en ua tabla ya definida en la siguiente:
CREATE [UNIQUE] INDEX índice
ON Tabla (campo [ASC|DESC][, campo [ASC|DESC], ...])
[WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]

En donde:

índice	Es el nombre del índice a crear.
tabla	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	Prohibe 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 sólo puede existir un único índice que sea "Clave Principal". Si un índice es clave principal implica que no puede contener valores nulos ni duplicados.

En el caso de ACCESS, se puede utilizar CREATE INDEX para crear un pseudo índice sobre una tabla adjunta en una fuente de datos ODBC tal como SQL Server que no tenga todavía un índice. No necesita permiso o tener acceso a un servidor remoto para crear un pseudo índice, además la base de datos remota no es consciente y no es afectada por el pseudo índice. Se utiliza la misma sintaxis para las tablas adjuntas que para las originales. Esto es especialmente útil para crear un índice en una tabla que sería de sólo lectura debido a la falta de un índice.+

```
CREATE INDEX
  MiIndice
ON
   Empleados (Prefijo, Telefono)
```

(Crea un índice llamado MiIndice en la tabla empleados con los campos Prefijo y Teléfono.)

```
CREATE UNIQUE INDEX
   MiIndice
ON
  Empleados (IdEmpleado)
  WITH DISALLOW NULL
```

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

Modificar el Diseño de una Tabla

Modifica el diseño de una tabla ya existente, se pueden modificar los campos o los índices existentes. Su sintaxis es:

```
ALTER TABLE tabla {ADD {COLUMN tipo de campo[(tamaño)]
[CONSTRAINT índice]
CONSTRAINT índice multicampo} |
DROP {COLUMN campo I CONSTRAINT nombre del índice}}
```

En donde:

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	Es 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 CURRENCY
```
(Agrega un campo Salario de tipo Moneda 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
  (IdEmpleado)
REFERENCES
   Empleados (IdEmpleado)
```
(Agrega un índice externo a la tabla Pedidos. El índice externo se basa en el campo IdEmpleado y se refiere al campo IdEmpleado 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.)

Sintaxis y ejemplos para introducir registros en una tabla
Los registros pueden ser introducidos a partir de sentencias que emplean la instrucción Insert.
La sintaxis utilizada es la siguiente:

```
Insert Into nombre_tabla (nombre_campo1, nombre_campo2,...) Values (valor_campo1, valor_campo2...)
```
Un ejemplo sencillo a partir de nuestra tabla modelo es la introducción de un nuevo cliente lo cual se haría con una instrucción de este tipo:

```
Insert Into clientes (nombre, apellidos, direccion, poblacion, codigopostal, email, pedidos) Values ('Perico', 'Palotes', 'Percebe n°13', 'Lepe', '123456', 'perico@desarrolloweb.com', 33)
```
Como puede verse, los campos no numéricos o booleanos van delimitados por apostrofes: '. También resulta interesante ver que el código postal lo hemos guardado como un campo no numérico. Esto es debido a que en determinados paises (Inglaterra,como no) los codigos postales contienen también letras.

Nota: Si deseamos practicar con una base de datos que está vacía primero debemos crear las tablas que vamos a llenar. Las tablas también se crean con sentencias SQL y aprendemos a hacerlo en el último capítulo.
Aunque, de todos modos, puede que sea más cómodo utilizar un programa con interfaz gráfica, como Access, que nos puede servir para crear las tablas en bases de datos del propio Access o por ODBC a otras bases de datos como SQL Server o MySQL, por poner dos ejemplos.

Otra posibilidad en una base de datos como MySQL, sería crear las tablas utilizando un software como PhpMyAdmin.

Por supuesto, no es imprescindible rellenar todos los campos del registro. Eso sí, puede ser que determinados campos sean necesarios. Estos campos necesarios pueden ser definidos cuando construimos nuestra tabla mediante la base de datos.

Nota: Si no insertamos uno de los campos en la base de datos se inicializará con el valor por defecto que hayamos definido a la hora de crear la tabla. Si no hay valor por defecto, probablemente se inicialice como NULL (vacío), en caso de que este campo permita valores nulos. Si ese campo no permite valores nulos (eso se define también al crear la tabla) lo más seguro es que la ejecución de la sentencia SQL nos de un error.

Resulta muy interesante, ya veremos más adelante el por qué, el introducir durante la creación de nuestra tabla un campo autoincrementable que nos permita asignar un único número a cada uno de los registros. De este modo, nuestra tabla clientes presentaría para cada registro un número exclusivo del cliente el cual nos será muy util cuando consultemos varias tablas simultáneamente.

Sintaxis y ejemplos para borrar registros en una tabla

Para borrar un registro nos servimos de la instrucción Delete. En este caso debemos especificar cual o cuales son los registros que queremos borrar. Es por ello necesario establecer una selección que se llevara a cabo mediante la cláusula Where.
La forma de seleccionar se verá detalladamente en capítulos posteriores. Por ahora nos contentaremos de mostrar cuál es el tipo de sintaxis utilizado para efectuar estas supresiones:

Delete From nombre_tabla Where condiciones_de_selección

Nota: Si deseamos practicar con una base de datos que está vacía primero debemos crear las tablas que vamos a llenar. Las tablas también se crean con sentencias SQL y aprendemos a hacerlo en el último capítulo.

Si queremos por ejemplo borrar todos los registros de los clientes que se llamen Perico lo haríamos del siguiente modo:

Delete From clientes Where nombre='Perico'
Hay que tener cuidado con esta instrucción ya que si no especificamos una condición con Where, lo que estamos haciendo es **borrar toda la tabla**:
Delete From clientes

Sintaxis se la sentencia UPDATE del lenguaje SQL y ejemplos para editar registros en una tabla.
Update es la instrucción del lenguaje SQL que nos sirve para modificar los registros de una tabla. Como para el caso de Delete, necesitamos especificar por medio de Where cuáles son los registros en los que queremos hacer efectivas nuestras modificaciones. Además, obviamente, tendremos que especificar cuáles son los nuevos valores de los campos que deseamos actualizar.
La sintaxis es de este tipo:

Update nombre_tabla Set nombre_campo1 = valor_campo1, nombre_campo2 = valor_campo2,... Where condiciones_de_selección

Un ejemplo aplicado:

Update clientes Set nombre='José' Where nombre='Pepe'

Mediante esta sentencia cambiamos el nombre Pepe por el de José en todos los registros cuyo nombre sea Pepe.

Aquí también hay que ser cuidadoso de no olvidarse de usar Where, de lo contrario, modificaríamos todos los registros de nuestra tabla.

Update producto Set precio=990, descuento=25

Esa sentencia modificaría el campo precio y el campo descuento en todos los productos de la tabla producto. Si tenemos una tabla con miles de productos con esa sentencia se actualizarían todos, de modo que la totalidad de los

registros tendrían el mismo precio y el mismo descuento. Os aseguro que este problema de olvidarse el where no es algo extraño que ocurra, incluso para programadores experimentados y puede acarrear problemas serios.

SELECCIÓN DE TABLAS

Cómo realizar selecciones eficientemente. Ejemplos prácticos.

La selección total o parcial de una tabla se lleva a cabo mediante la instrucción Select. En dicha selección hay que especificar:

-Los campos que queremos seleccionar
-La tabla en la que hacemos la selección

En nuestra tabla modelo de clientes podríamos hacer por ejemplo una selección del nombre y dirección de los clientes con una instrucción de este tipo:

```
Select nombre, dirección From clientes
```

Si quisiésemos seleccionar todos los campos, es decir, **toda la tabla**, podríamos utilizar el comodín * del siguiente modo:

```
Select * From clientes
```

Resulta también muy útil el filtrar los registros mediante condiciones que vienen expresadas después de la **cláusula Where**. Si quisiésemos mostrar los clientes de una determinada ciudad usaríamos una expresión como esta:

```
Select * From clientes Where poblacion Like 'Madrid'
```

Además, podríamos **ordenar los resultados** en función de uno o varios de sus campos. Para este ultimo ejemplo los podríamos ordenar por nombre así:

```
Select * From clientes Where poblacion Like 'Madrid' Order By nombre
```

Teniendo en cuenta que puede haber más de un cliente con el mismo nombre, podríamos dar un segundo criterio que podría ser el apellido:

```
Select * From clientes Where poblacion Like 'Madrid' Order By nombre, apellido
```

Si invirtiésemos el orden « nombre,apellido » por « apellido, nombre », el resultado sería distinto. Tendríamos los clientes ordenados por apellido y aquellos que tuviesen apellidos idénticos se subclasificarían por el nombre.

Es posible también **clasificar por orden inverso**. Si por ejemplo quisiésemos ver nuestros clientes por orden de pedidos realizados teniendo a los mayores en primer lugar escribiríamos algo así:

```
Select * From clientes Order By pedidos Desc
```

Una opción interesante es la de efectuar **selecciones sin coincidencia**. Si por ejemplo buscásemos el saber en qué ciudades se encuentran nuestros clientes sin necesidad de que para ello aparezca varias veces la misma ciudad usaríamos una sentencia de esta clase:

```
Select Distinct poblacion From clientes Order By poblacion
```

Así evitaríamos ver repetido Madrid tantas veces como clientes tengamos en esa población.

Lista de operadores y ejemplos prácticos para realizar selecciones.

Hemos querido compilar a modo de tabla ciertos operadores que pueden resultar útiles en determinados casos. Estos operadores serán utilizados después de la cláusula Where y pueden ser **combinados hábilmente mediante paréntesis**para optimizar nuestra selección a muy altos niveles.

Operadores matemáticos:	
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<>	Distinto
=	Igual

Operadores lógicos
And
Or
Not

Otros operadores	
Like	Selecciona los registros cuyo valor de campo se asemeje, no teniendo en cuenta mayúsculas y minúsculas.
In y Not In	Da un conjunto de valores para un campo para los cuales la condición de selección es (o no) valida
Is Null y Is Not Null	Selecciona aquellos registros donde el campo especificado esta (o no) vacío.
Between...And	Selecciona los registros comprendidos en un intervalo
Distinct	Selecciona los registros no coincidentes
Desc	Clasifica los registros por orden inverso

Comodines	
*	Sustituye a todos los campos
%	Sustituye a cualquier cosa o nada dentro de una cadena
_	Sustituye un solo carácter dentro de una cadena

Veamos a continuación aplicaciones prácticas de estos operadores.

En esta sentencia seleccionamos todos los clientes de Madrid cuyo nombre no es Pepe. Como puede verse, empleamos**Like** en lugar de **=** simplemente para evitar inconvenientes debido al empleo o no de mayúsculas.

Select * From clientes Where poblacion **Like** 'madrid' **And Not** nombre **Like** 'Pepe'

Si quisiéramos recoger en una selección a los clientes de nuestra tabla cuyo **apellido comienza por A y cuyo número de pedidos esta comprendido entre 20 y 40**:

Select * From clientes Where apellidos **like 'A%'** And pedidos **Between 20 And 40**

El operador **In**, lo veremos más adelante, es muy práctico para consultas en varias tablas. Para casos en una sola tabla es empleado del siguiente modo:

Select * From clientes Where poblacion **In ('Madrid','Barcelona','Valencia')**

De esta forma **seleccionamos aquellos clientes que vivan en esas tres ciudades**.

Cómo realizar selecciones sobre varias tablas. Ejemplos prácticos basados en una aplicación de e-comercio.

Una base de datos puede ser considerada como un conjunto de tablas. Estas tablas en muchos casos están relacionadas entre ellas y se complementan unas con otras.

Refiriéndonos a nuestro clásico ejemplo de una base de datos para una aplicación de e-comercio, la tabla clientes de la que hemos estado hablando puede estar perfectamente coordinada con una tabla donde almacenamos los pedidos realizados por cada cliente. Esta tabla de pedidos puede a su vez estar conectada con una tabla donde almacenamos los datos correspondientes a cada artículo del inventario.

De este modo podríamos fácilmente obtener informaciones contenidas en esas tres tablas como puede ser la designación del artículo más popular en una determinada región donde la designación del artículo sería obtenida de la tabla de artículos, la popularidad (cantidad de veces que ese artículo ha sido vendido) vendría de la tabla de pedidos y la región estaría comprendida obviamente en la tabla clientes.

Este tipo de organización basada en múltiples tablas conectadas nos permite trabajar con tablas mucho más manejables a la vez que nos evita copiar el mismo campo en varios sitios ya que podemos acceder a él a partir de una simple llamada a la tabla que lo contiene.

En este capítulo veremos cómo, sirviéndonos de lo aprendido hasta ahora, podemos realizar fácilmente selecciones sobre varias tablas. Definamos antes de nada las diferentes tablas y campos que vamos a utilizar en nuestros ejemplos:

Tabla de clientes	
Nombre campo	**Tipo campo**
id_cliente	Numérico entero
nombre	Texto
apellidos	Texto
direccion	Texto

poblacion	Texto
codigopostal	Texto
telefono	Numérico entero
email	Texto

Tabla de pedidos	
Nombre campo	**Tipo campo**
id_pedido	Numérico entero
id_cliente	Numérico entero
id_articulo	Numérico entero
fecha	Fecha
cantidad	Numérico entero

Tabla de artículos	
Nombre campo	**Tipo campo**
id_articulo	Numérico entero
titulo	Alfanumérico
autor	Alfanumérico
editorial	Alfanumérico
precio	Numérico real

Estas tablas pueden ser utilizadas simultáneamente para extraer informaciones de todo tipo. Supongamos que queremos enviar un mailing a todos aquellos que hayan realizado un pedido ese mismo día. Podríamos escribir algo así:

Select clientes.apellidos, clientes.email From clientes,pedidos Where pedidos.fecha like '25/02/00' And pedidos.id_cliente= clientes.id_cliente

Como puede verse esta vez, después de la cláusula From, introducimos el nombre de las dos tablas de donde sacamos las informaciones. Además, el nombre de cada campo va precedido de la tabla de provenencia separados ambos por un punto. En los campos que poseen un nombre que solo aparece en una de las tablas, no es necesario especificar su origen aunque a la hora de leer la sentencia puede resultar más claro el precisarlo. En este caso el campo fecha podría haber sido designado como "fecha" en lugar de "pedidos.fecha".

Veamos otro ejemplo más para consolidar estos nuevos conceptos. Esta vez queremos ver el título del libro correspondiente a cada uno de los pedidos realizados:

Select pedidos.id_pedido, articulos.titulo From pedidos, articulos Where pedidos.id_articulo=articulos.id_articulo

En realidad la filosofía continua siendo la misma que para la consulta de una única tabla.

El empleo de funciones para la explotación de los campos numéricos y otras utilidades. Ejemplos prácticos.

Además de los criterios hasta ahora explicados para realizar las consultas en tablas, SQL permite también aplicar un conjunto de funciones predefinidas. Estas funciones, aunque básicas, pueden ayudarnos en algunos momentos a expresar nuestra selección de una manera más simple sin tener que recurrir a operaciones adicionales por parte del script que estemos ejecutando.

Algunas de estas funciones son representadas en la tabla siguiente:

Función	Descripción
Sum(campo)	Calcula la suma de los registros del campo especificado
Avg(Campo)	Calcula la media de los registros del campo especificado
Count(*)	Nos proporciona el valor del número de registros que han sido seleccionados
Max(Campo)	Nos indica cual es el valor máximo del campo
Min(Campo)	Nos indica cual es el valor mínimo del campo

Dado que el campo de la función no existe en la base de datos, sino que lo estamos generando virtualmente, esto puede crear inconvenientes cuando estamos trabajando con nuestros scripts a la hora de tratar su valor y su nombre de campo. Es por ello que el valor de la **función ha de ser recuperada a partir de un alias** que nosotros especificaremos en la sentencia SQL a partir de la instrucción **AS**. La cosa podría quedar así:

Select Sum(total) As suma_pedidos From pedidos

A partir de esta sentencia calculamos la suma de los valores de todos los pedidos realizados y almacenamos ese valor en un campo virtual llamado suma_pedidos que podrá ser utilizado como cualquier otro campo por nuestras paginas dinámicas.

Por supuesto, todo lo visto hasta ahora puede ser aplicado en este tipo de funciones de modo que, por ejemplo, podemos establecer condiciones con la cláusula Where construyendo sentencias como esta:

Select Sum(cantidad) as suma_articulos From pedidos Where id_articulo=6

Esto nos proporcionaría la cantidad de **ejemplares de un determinado libro que han sido vendidos**.
Otra propiedad interesante de estas funciones es que **permiten realizar operaciones con varios campos dentro de un mismo paréntesis**:

Select Avg(total/cantidad) From pedidos

Esta sentencia da como resultado el **precio medio al que se están vendiendo los libros**. Este resultado no tiene por qué coincidir con el del **precio medio de los libros presentes en el inventario**, ya que, puede ser que la gente tenga tendencia a comprar los libros caros o los baratos:

Select Avg(precio) as precio_venta From artículos

Una cláusula interesante en el uso de funciones es Group By. Esta cláusula nos permite agrupar registros a los cuales vamos a aplicar la función. Podemos por ejemplo calcular el **dinero gastado por cada cliente**:

Select id_cliente, Sum(total) as suma_pedidos From pedidos Group By id_cliente

O saber el **numero de pedidos que han realizado**:

Select id_cliente, Count(*) as numero_pedidos From pedidos Group By id_cliente
Las posibilidades como vemos son numerosas y pueden resultar prácticas. Todo queda ahora a disposición de nuestras ocurrencias e imaginación.

Consultas de selección

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos, esta información es devuelta en forma de conjunto de registros que se pueden almacenar en un objeto recordset.

Este conjunto de registros puede ser modificable.

Consultas básicas

La sintaxis básica de una consulta de selección es la siguiente:

```
SELECT
  Campos
FROM
  Tabla
```

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

```
SELECT
  Nombre, Teléfono
FROM
  Clientes
```

Esta sentencia devuelve un conjunto de resultados con el campo nombre y teléfono de la tabla clientes.

Devolver Literales

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 las tarifas semanales de los electricistas, podríamos realizar la siguiente consulta:

```
SELECT
  Empleados.Nombre, 'Tarifa semanal: ', Empleados.TarifaHora * 40
FROM
  Empleados
WHERE
  Empleados.Cargo = 'Electricista'
```

Ordenar los registros

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
  CodigoPostal, Nombre, Telefono
FROM
  Clientes
ORDER BY
  Nombre
```

Esta consulta devuelve los campos CodigoPostal, Nombre, Telefono de la tabla Clientes ordenados por el campo Nombre.

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

```
SELECT
  CodigoPostal, Nombre, Telefono
FROM
  Clientes
```

```
ORDER BY
  CodigoPostal, Nombre
```
Incluso se puede especificar el orden de los registros: ascendente mediante la cláusula (ASC - se toma este valor por defecto) ó descendente (DESC)

```
SELECT
  CodigoPostal, Nombre, Telefono
FROM
  Clientes
ORDER BY
  CodigoPostal DESC , Nombre ASC
```

Uso de Indices de las tablas

Si deseamos que la sentencia SQL utilice un índice para mostrar los resultados se puede utilizar la palabra reservada INDEX de la siguiente forma:

```
SELECT ... FROM Tabla (INDEX=Indice) ...
```

Normalmente los motores de las bases de datos deciden que índice se debe utilizar para la consulta, para ello utilizan criterios de rendimiento y sobre todo los campos de búsqueda especificados en la cláusula WHERE. Si se desea forzar a no utilizar ningún índice utilizaremos la siguiente sintaxis:

```
SELECT ... FROM Tabla (INDEX=0) ...
```

Consultas con Predicado

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

Predicado	**Descripción**
ALL	Devuelve todos los campos de la tabla
TOP	Devuelve un determinado número de registros de la tabla
DISTINCT	Omite los registros cuyos campos seleccionados coincidan totalmente
DISTINCTOW	Omite los registros duplicados basándose en la totalidad del registro y no sólo en los campos seleccionados.

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
FROM
  Empleados
SELECT *
FROM
  Empleados
```

TOP

Devuelve un cierto número de registros que entran entre al principio o al final de un rango especificado por una cláusula ORDER BY. Supongamos que queremos recuperar los nombres de los 25 primeros estudiantes del curso 1994:

```
SELECT TOP 25
  Nombre, Apellido
FROM
  Estudiantes
```

```
ORDER BY
  Nota DESC
```

Si no se incluye la cláusula ORDER BY, la consulta devolverá un conjunto arbitrario de 25 registros de la tabla de Estudiantes. El predicado TOP no elige entre valores iguales. En el ejemplo anterior, si la nota media número 25 y la 26 son iguales, la consulta devolverá 26 registros. Se puede utilizar la palabra reservada PERCENT para devolver un cierto porcentaje de registros que caen al principio o al final de un rango especificado por la cláusula ORDER BY. Supongamos que en lugar de los 25 primeros estudiantes deseamos el 10 por ciento del curso:

```
SELECT TOP 10 PERCENT
  Nombre, Apellido
FROM
  Estudiantes
ORDER BY
  Nota DESC
```

El valor que va a continuación de TOP debe ser un entero sin signo. TOP no afecta a la posible actualización de la consulta.

DISTINCT

Omite los registros que contienen datos duplicados en los campos seleccionados. Para que los valores de cada campo listado en la instrucción SELECT se incluyan en la consulta deben ser únicos. Por ejemplo, varios empleados listados en la tabla Empleados pueden tener el mismo apellido. Si dos registros contienen López en el campo Apellido, la siguiente instrucción SQL devuelve un único registro:

```
SELECT DISTINCT
  Apellido
FROM
  Empleados
```

Con otras palabras el predicado DISTINCT devuelve aquellos registros cuyos campos indicados en la cláusula SELECT posean un contenido diferente. El resultado de una consulta que utiliza DISTINCT no es actualizable y no refleja los cambios subsiguientes realizados por otros usuarios.

DISTINCTROW

Este predicado no es compatible con ANSI. Que yo sepa a día de hoy sólo funciona con ACCESS.

Devuelve los registros diferentes de una tabla; a diferencia del predicado anterior que sólo se fijaba en el contenido de los campos seleccionados, éste lo hace en el contenido del registro completo independientemente de los campos indicados en la cláusula SELECT.

```
SELECT DISTINCTROW
  Apellido
FROM Empleados
```

Si la tabla empleados contiene dos registros: Antonio López y Marta López el ejemplo del predicado DISTINCT devuelve un único registro con el valor López en el campo Apellido ya que busca no duplicados en dicho campo. Este último ejemplo devuelve dos registros con el valor López en el apellido ya que se buscan no duplicados en el registro completo.

ALIAS

En determinadas circunstancias es necesario asignar un nombre a alguna columna determinada de un conjunto devuelto, otras veces por simple capricho o porque estamos recuperando datos de diferentes tablas y resultan tener un campo con igual nombre. Para resolver todas ellas tenemos la palabra reservada AS que se encarga de asignar el nombre que deseamos a la columna deseada. Tomado como referencia el ejemplo anterior podemos hacer que la columna devuelta por la consulta, en lugar de llamarse apellido (igual que el campo devuelto) se llame Empleado. En este caso procederíamos de la siguiente forma:

```
SELECT DISTINCTROW
  Apellido AS Empleado
FROM Empleados
```

AS no es una palabra reservada de ANSI, existen diferentes sistemas de asignar los alias en función del motor de bases de datos. En ORACLE para asignar un alias a un campo hay que hacerlo de la siguiente forma:

```
SELECT
  Apellido AS "Empleado"
FROM Empleados
```

También podemos asignar alias a las tablas dentro de la consulta de selección, en esta caso hay que tener en cuenta que en todas las referencias que deseemos hacer a dicha tabla se ha de utilizar el alias en lugar del nombre. Esta técnica será de gran utilidad más adelante cuando se estudien las vinculaciones entre tablas. Por ejemplo:

```
SELECT
  Apellido AS Empleado
FROM
  Empleados AS Trabajadores
```

Para asignar alias a las tablas en ORACLE y SQL-SERVER los alias se asignan escribiendo el nombre de la tabla, dejando un espacio en blanco y escribiendo el Alias (se asignan dentro de la cláusula FROM).

```
SELECT
  Trabajadores.Apellido (1) AS Empleado
FROM
  Empleados Trabajadores
```

(1)Esta nomenclatura [Tabla].[Campo] se debe utilizar cuando se está recuperando un campo cuyo nombre se repite en varias de las tablas que se utilizan en la sentencia. No obstante cuando en la sentencia se emplean varias tablas es aconsejable utilizar esta nomenclatura para evitar el trabajo que supone al motor de datos averiguar en que tabla está cada uno de los campos indicados en la cláusula SELECT.

Recuperar Información de una base de Datos Externa

Para concluir este capítulo se debe hacer referencia a la recuperación de registros de bases de datos externas. Es ocasiones es necesario la recuperación de información que se encuentra contenida en una tabla que no se encuentra en la base de datos que ejecutará la consulta o que en ese momento no se encuentra abierta, esta situación la podemos salvar con la palabra reservada IN de la siguiente forma:

```
SELECT
  Apellido AS Empleado
FROM
  Empleados IN'c: databasesgestion.mdb'
```

En donde c: databasesgestion.mdb es la base de datos que contiene la tabla Empleados. Esta técnica es muy sencilla y común en bases de datos de tipo ACCESS en otros sistemas como SQL-SERVER u ORACLE, la cosa es más complicada la tener que existir relaciones de confianza entre los servidores o al ser necesaria la vinculación entre las bases de datos. Este ejemplo recupera la información de una base de datos de SQL-SERVER ubicada en otro servidor (se da por supuesto que los servidores están lincados):

```
SELECT
  Apellido
FROM
  Servidor1.BaseDatos1.dbo.Empleados
```

Criterios de selección en SQL

Estudiaremos las posibilidades de filtrar los registros con el fin de recuperar solamente aquellos que cumplan unas condiciones preestablecidas.

En el apartado anterior se vio la forma de recuperar los registros de las tablas, las formas empleadas devolvían todos los registros de la mencionada tabla. A lo largo de este apartado se estudiarán las posibilidades de filtrar los registros con el fin de recuperar solamente aquellos que cumplan unas condiciones preestablecidas.

Antes de comenzar el desarrollo de este apartado hay que recalcar tres detalles de vital importancia. El primero de ellos es que cada vez que se desee establecer una condición referida a un campo de texto la condición de búsqueda debe ir encerrada entre comillas simples; la segunda es que no es posible establecer condiciones de búsqueda en los campos memo y; la tercera y última hace referencia a las fechas. A día de hoy no he sido capaz de encontrar una sintaxis que funcione en todos los sistemas, por lo que se hace necesario particularizarlas según el banco de datos:

Banco de Datos		Sintaxis
SQL-SERVER		Fecha = #mm-dd-aaaa#
ORACLE		Fecha = to_date('YYYYDDMM','aaaammdd',)
ACCESS		Fecha = #mm-dd-aaaa#

Ejemplo

Banco de Datos Ejemplo (para grabar la fecha 18 de mayo de 1969)

SQL-SERVER		Fecha = #05-18-1969# ó Fecha = 19690518
ORACLE		Fecha = to_date('YYYYDDMM', '19690518')
ACCESS		Fecha = #05-18-1969#

Referente a los valores lógicos True o False cabe destacar que no son reconocidos en ORACLE, ni en este sistema de bases de datos ni en SQL-SERVER existen los campos de tipo "SI/NO" de ACCESS; en estos sistemas se utilizan los campos BIT que permiten almacenar valores de 0 ó 1. Internamente, ACCESS, almacena en estos campos valores de 0 ó -1, así que todo se complica bastante, pero aprovechando la coincidencia del 0 para los valores FALSE, se puede utilizar la sintaxis siguiente que funciona en todos los casos: si se desea saber si el campo es falso "... CAMPO = 0" y para saber los verdaderos "CAMPO <> 0".

Operadores Lógicos

Los operadores lógicos soportados por SQL son: AND, OR, XOR, Eqv, Imp, Is y Not. A excepción de los dos últimos todos poseen la siguiente sintaxis:

<expresión1> operador <expresión2>

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico. La tabla adjunta muestra los diferentes posibles resultados:

<expresión1>	**Operador**	**<expresión2>**	**Resultado**
Verdad	AND	Falso	Falso
Verdad	AND	Verdad	Verdad
Falso	AND	Verdad	Falso
Falso	AND	Falso	Falso
Verdad	OR	Falso	Verdad
Verdad	OR	Verdad	Verdad
Falso	OR	Verdad	Verdad
Falso	OR	Falso	Falso
Verdad	XOR	Verdad	Falso
Verdad	XOR	Falso	Verdad
Falso	XOR	Verdad	Verdad
Falso	XOR	Falso	Falso
Verdad	Eqv	Verdad	Verdad

Verdad	Eqv	Falso	Falso
Falso	Eqv	Verdad	Falso
Falso	Eqv	Falso	Verdad
Verdad	Imp	Verdad	Verdad
Verdad	Imp	Falso	Falso
Verdad	Imp	Null	Null
Falso	Imp	Verdad	Verdad
Falso	Imp	Falso	Verdad
Falso	Imp	Null	Verdad
Null	Imp	Verdad	Verdad
Null	Imp	Falso	Null
Null	Imp	Null	Null

Si a cualquiera de las anteriores condiciones le anteponemos el operador NOT el resultado de la operación será el contrario al devuelto sin el operador NOT.

El último operador denominado Is se emplea para comparar dos variables de tipo objeto <Objeto1> Is <Objeto2>. este operador devuelve verdad si los dos objetos son iguales.

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

```
SELECT *
FROM
   Empleados
WHERE
   (Edad > 25 AND Edad < 50)
  OR
  Sueldo = 100
```

```
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 la provincia de Madrid)

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 cadena modelo o campo contra el que se compara 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 puede utilizar una cadena de caracteres comodín como los reconocidos por el sistema operativo para encontrar un rango de valores (Like An*).

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. En una consulta con parámetros, puede hacer que el usuario escriba el modelo que se va a utilizar.

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.

ACCESS

Tipo de coincidencia	**Modelo Planteado**	**Coincid**e	**No coincide**
Varios caracteres	'a*a'	aa', 'aBa', 'aBBBa'	'aBC'
Carácter especial	'a[*]a'	'a*a'	'aaa'
Varios caracteres	'ab*'	'abcdefg', 'abc'	'cab', 'aab'
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'

SQL-SERVER

Ejemplo	Descripción
LIKE 'A%'	Todo lo que comience por A
LIKE '_NG'	Todo lo que comience por cualquier carácter y luego siga NG
LIKE '[AF]%'	Todo lo que comience por A ó F
LIKE '[A-F]%'	Todo lo que comience por cualquier letra comprendida entre la A y la F
LIKE '[A^B]%'	Todo lo que comience por A y la segunda letra no sea una B

En determinado motores de bases de datos, esta cláusula, no reconoce el asterisco como carácter comodín y hay que sustituirlo por el carácter tanto por ciento (%).

El Operador In

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

expresión [Not] In(valor1, valor2, . . .)

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

La cláusula WHERE

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 apartados anteriores. 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
  IdProducto, Existencias
FROM
  Productos
WHERE
   Existencias <= NuevoPedido
```

```
SELECT *
FROM
  Pedidos
WHERE
  FechaEnvio = #05-30-1994#
```

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

```
SELECT
   Apellidos, Nombre
FROM
  Empleados
WHERE
  Apellidos Like 'S*'
```

```
SELECT
   Apellidos, Salario
FROM
   Empleados
WHERE
   Salario Between 200 And 300
```

```
SELECT
   Apellidos, Salario
FROM
   Empleados
WHERE
   Apellidos Between 'Lon' And 'Tol'
```

```
SELECT
   IdPedido, FechaPedido
FROM
   Pedidos
WHERE
    FechaPedido Between #01-01-1994# And #12-31-1994#
```

```
SELECT
   Apellidos, Nombre, Ciudad
FROM
  Empleados
WHERE
  Ciudad In ('Sevilla', 'Los Angeles', 'Barcelona')
```

Seguimos con el group by, avg, sum y con el compute de sql-server.

Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro. Para cada registro se crea un valor sumario si se incluye una función SQL agregada, como por ejemplo Sum o Count, en la instrucción SELECT. Su sintaxis es:

SELECT campos FROM tabla WHERE criterio GROUP BY campos del grupo
GROUP BY es opcional. Los valores de resumen se omiten si no existe una función SQL agregada en la instrucción SELECT. Los valores Null en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores Null no se evalúan en ninguna de las funciones SQL agregadas.

Se utiliza la cláusula WHERE para excluir aquellas filas que no desea agrupar, y la cláusula HAVING para filtrar los registros una vez agrupados.
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 esta 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 bien incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada.

```
SELECT
  IdFamilia, Sum(Stock) AS StockActual
FROM
   Productos
GROUP BY
   IdFamilia
```

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.

HAVING es similar a WHERE, determina qué registros se seleccionan. Una vez que los registros se han agrupado utilizando GROUP BY, HAVING determina cuales de ellos se van a mostrar.

```
SELECT
  IdFamilia, Sum(Stock) AS StockActual
FROM
   Productos
GROUP BY
   IdFamilia
HAVING
   StockActual > 100
  AND
  NombreProducto Like BOS*
```

AVG

Calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta. Su sintaxis es la siguiente

Avg(expr)
En 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 devueltos por una consulta. Su sintaxis es la siguiente

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

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, Min

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

```
Min(expr)
Max(expr)
```

En donde expr es el campo sobre el que se desea realizar el cálculo. 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
Min(Gastos) AS ElMin
FROM
  Pedidos
WHERE
  Pais = 'España'

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

StDev, StDevP

Devuelve estimaciones de la desviación estándar para la población (el total de los registros de la tabla) o una muestra de la población representada (muestra aleatoria). Su sintaxis es:

```
StDev(expr)
StDevP(expr)
```

En donde expr representa el nombre del campo que contiene los datos que desean evaluarse 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).

StDevP evalúa una población, y StDev evalúa una muestra de la población. Si la consulta contiene menos de dos registros (o ningún registro para StDevP), estas funciones devuelven un valor Null (el cual indica que la desviación estándar no puede calcularse).

```
SELECT
  StDev(Gastos) AS Desviación
FROM
   Pedidos
```

```
WHERE
   País = 'España'
```

```
SELECT
   StDevP(Gastos) AS Desviación
FROM
  Pedidos
WHERE
   País = 'España'
```

Sum

Devuelve la suma del conjunto de valores contenido en un campo especifico 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
```

Var, VarP

Devuelve una estimación de la varianza de una población (sobre el total de los registros) o una muestra de la población (muestra aleatoria de registros) sobre los valores de un campo. Su sintaxis es:

Var(expr)
VarP(expr)

VarP evalúa una población, y Var evalúa una muestra de la población. Expr el nombre del campo que contiene los datos que desean evaluarse 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)

Si la consulta contiene menos de dos registros, Var y VarP devuelven Null (esto indica que la varianza no puede calcularse). Puede utilizar Var y VarP en una expresión de consulta o en una Instrucción SQL.

```
SELECT
  Var(Gastos) AS Varianza
FROM
   Pedidos
WHERE
   País = 'España'
```

```
SELECT
   VarP(Gastos) AS Varianza
FROM
  Pedidos
WHERE
  País = 'España'
```

COMPUTE de SQL-SERVER

Esta cláusula añade una fila en el conjunto de datos que se está recuperando, se utiliza para realizar cálculos en campos numéricos. COMPUTE actúa siempre sobre un campo o expresión del conjunto de resultados y esta expresión debe figurar exactamente igual en la cláusula SELECT y siempre se debe ordenar el resultado por la misma o al memos agrupar el resultado. Esta expresión no puede utilizar ningún ALIAS.

```
SELECT
  IdCliente, Count(IdPedido)
FROM
  Pedidos
GROUP BY
  IdPedido
HAVING
  Count(IdPedido) > 20
COMPUTE
   Sum(Count(IdPedido))

SELECT
   IdPedido, (PrecioUnidad * Cantidad - Descuento)
FROM
  [Detalles de Pedidos]
ORDER BY
   IdPedido
COMPUTE
  Sum((PrecioUnidad * Cantidad - Descuento)) // Calcula el Total
   BY IdPedido // Calcula el Subtotal
```

Subconsultas en SQL

Defenimos lo que significa subconsulta y mostramos las diferentes subconsultas que se pueden hacer.

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)

En donde:

comparación	Es una expresión y un operador de comparación que compara la expresión con el resultado de la subconsulta.
expresión	Es una expresión por la que se busca el conjunto resultante de la subconsulta.
instrucción SQL	Es una instrucción SELECT, que sigue el mismo formato y reglas que cualquier otra instrucción SELECT. Debe ir entre paréntesis.

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
  Productos
WHERE
  PrecioUnidad
  ANY
  (
  SELECT
  PrecioUnidad
  FROM
  DetallePedido
  WHERE
  Descuento = 0 .25
)
```

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. Si se cambia ANY por ALL en el ejemplo anterior, la consulta devolverá únicamente aquellos productos cuyo precio unitario sea mayor que el de todos los productos vendidos con un descuento igual o mayor al 25 por ciento. Esto es mucho más restrictivo.

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. El ejemplo siguiente devuelve todos los productos vendidos con un descuento igual o mayor al 25 por ciento:

```
SELECT *
FROM
   Productos
WHERE
  IDProducto
   IN
  (
   SELECT
  IDProducto
  FROM
   DetallePedido
  WHERE
  Descuento = 0.25
)
```

Inversamente 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.

El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro. Supongamos que deseamos recuperar todos aquellos clientes que hayan realizado al menos un pedido:

```
SELECT
Clientes.Compañía, Clientes.Teléfono
FROM
  Clientes
WHERE EXISTS (
  SELECT
  FROM
  Pedidos
   WHERE
   Pedidos.IdPedido = Clientes.IdCliente
)
```

Esta consulta es equivalente a esta otra:

```
SELECT
  Clientes.Compañía, Clientes.Teléfono
FROM
  Clientes
WHERE
  IdClientes
  IN
   (
  SELECT
  Pedidos.IdCliente
   FROM
  Pedidos
)
```

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 el 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 DISTINCT
    NombreContacto, NombreCompania, CargoContacto, Telefono
FROM
   Clientes
WHERE
   IdCliente IN (
    SELECT DISTINCT IdCliente
   FROM Pedidos
   WHERE FechaPedido <#07/01/1993#
)
```

(Obtiene una lista de las compañías y los contactos de todos los clientes que han realizado un pedido en el segundo trimestre de 1993.)

```
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 que productos no se podemos servir por falta de stock.

```
SELECT
  PedidosPendientes.Nombre
FROM
   PedidosPendientes
GROUP BY
   PedidosPendientes.Nombre
HAVING
  SUM(PedidosPendientes.Cantidad <
  (
  SELECT
  Productos.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')
```

ó lo que sería lo mismo:

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

La siguiente tabla muestra algún ejemplo del operador ANY y ALL

Valor 1	Operador	Valor 2	Resultado
3	> ANY	(2,5,7)	Cierto
3	= ANY	(2,5,7)	Falso
3	= ANY	(2,3,5,7)	Cierto
3	> ALL	(2,5,7)	Falso
3	< ALL	(5,6,7)	Falso

El operacion =ANY es equivalente al operador IN, ambos devuelven el mismo resultado.

Consultas SQL de Unión Internas

Detallamos estas consultas del lenguaje SQL tan importantes para el buen desarrollo de una base de datos.

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. Su sintaxis es:

SELECT campos FROM tb1 INNER JOIN tb2 ON
tb1.campo1 comp tb2.campo2

En donde:

tb1, tb2	Son los nombres de las tablas desde las que se combinan los registros.
campo1, campo2	Son los nombres de los campos que se combinan. Si no son numéricos, los campos deben ser del mismo tipo de datos y contener el mismo tipo de datos, pero no tienen que tener el mismo nombre.
comp	Es cualquier operador de comparación relacional: =, <,<>, <=, =>, ó >.

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; éstas 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 JOIN o todos los empleados (incluso si alguno no está asignado a ningún departamento), en este caso RIGHT JOIN.

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. Por ejemplo, puede combinar un campo Numérico para el que la propiedad Size de su objeto Field está establecida como Entero, y un campo Contador.

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

```
SELECT
   NombreCategoria, 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, incluir 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 puede 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 JOIN o un RIGHT JOIN puede anidarse dentro de un INNER JOIN, pero un INNER JOIN no puede anidarse dentro de un LEFT JOIN o un RIGHT JOIN.

Ejemplo:

```
SELECT DISTINCT
  Sum(PrecioUnitario * Cantidad) AS Sales,
(Nombre + ' ' + Apellido) AS Name
FROM
   Empleados
INNER JOIN(
   Pedidos
INNER JOIN
  DetallesPedidos
  ON
  Pedidos.IdPedido = DetallesPedidos.IdPedido)
   ON
   Empleados.IdEmpleado = Pedidos.IdEmpleado
GROUP BY
  Nombre + ' ' + Apellido
```

(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 sólo 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 y RIGHT. LEFT toma todos los registros de la tabla de la izquierda aunque no tengan ningún registro en la tabla de la izquierda. RIGHT 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.

La sintaxis expuesta anteriormente pertenece a ACCESS, en donde todas las sentencias con la sintaxis funcionan correctamente. Los manuales de SQL-SERVER dicen que esta sintaxis es incorrecta y que hay que añadir la palabra

reservada OUTER: LEFT OUTER JOIN y RIGHT OUTER JOIN. En la práctica funciona correctamente de una u otra forma.

No obstante, los INNER JOIN ORACLE no es capaz de interpretarlos, pero existe una sintaxis en formato ANSI para los INNER JOIN que funcionan en todos los sistemas. Tomando como referencia la siguiente sentencia:

```
SELECT
  Facturas.*,
  Albaranes.*
FROM
  Facturas
INNER JOIN
  Albaranes
ON
  Facturas.IdAlbaran = Albaranes.IdAlbaran
WHERE
  Facturas.IdCliente = 325
```

La transformación de esta sentencia a formato ANSI sería la siguiente:

```
SELECT
Facturas.*,
Albaranes.*
FROM
Facturas, Albaranes
WHERE
Facturas.IdAlbaran = Albaranes.IdAlbaran
AND
Facturas.IdCliente = 325
```

Como se puede observar los cambios realizados han sido los siguientes:

1. Todas las tablas que intervienen en la consulta se especifican en la cláusula FROM.
2. Las condiciones que vinculan a las tablas se especifican en la cláusula WHERE y se vinculan mediante el operador lógico AND.

Referente a los OUTER JOIN, no funcionan en ORACLE y además conozco una sintaxis que funcione en los tres sistemas. La sintaxis en ORACLE es igual a la sentencia anterior pero añadiendo los caracteres (+) detrás del nombre de la tabla en la que deseamos aceptar valores nulos, esto equivale a un LEFT JOIN:

```
SELECT
  Facturas.*,
   Albaranes.*
FROM
  Facturas, Albaranes
WHERE
  Facturas.IdAlbaran = Albaranes.IdAlbaran (+)
  AND
  Facturas.IdCliente = 325
```

Y esto a un RIGHT JOIN:

```
SELECT
  Facturas.*,
   Albaranes.*
FROM
  Facturas, Albaranes
WHERE
```

```
  Facturas.IdAlbaran (+) = Albaranes.IdAlbaran
  AND
  Facturas.IdCliente = 325
```

En SQL-SERVER se puede utilizar una sintaxis parecida, en este caso no se utiliza los caracteres (+) sino los caracteres =* para el LEFT JOIN y *= para el RIGHT JOIN.

Consultas de Autocombinación

La autocombinación 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
```

CROSS JOIN (SQL-SERVER)

Se utiliza en SQL-SERVER para realizar consultas de unión. Supongamos que tenemos una tabla con todos los autores y otra con todos los libros. Si deseáramos obtener un listado combinar ambas tablas de tal forma que cada autor apareciera junto a cada título, utilizaríamos la siguiente sintaxis:

```
SELECT
   Autores.Nombre, Libros.Titulo
FROM
   Autores CROSS JOIN Libros
```

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):

Autores	
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:

Código	Autor	Autor
B0012	1. Francisco López	1. Francisco López
B0012	1. Francisco López	2. Javier Alonso
B0012	1. Francisco López	3. Marta Rebolledo
B0012	2. Javier Alonso	2. Javier Alonso

B0012	2. Javier Alonso	1. Francisco López
B0012	2. Javier Alonso	3. Marta Rebolledo
B0012	3. Marta Rebolledo	3. Marta Rebolledo
B0012	3. Marta Rebolledo	2. Javier Alonso
B0012	3. Marta Rebolledo	1. Francisco López
C0014	1. Francisco López	1. Francisco López
C0014	1. Francisco López	2. Javier Alonso
C0014	2. Javier Alonso	2. Javier Alonso
C0014	2. Javier Alonso	1. Francisco López
D0120	2. Javier Alonso	2. Javier Alonso
D0120	2. Javier Alonso	3. Marta Rebolledo
D0120	3. Marta Rebolledo	3. Marta Rebolledo
D0120	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:

Código	**Autor**	**Autor**
B0012	1. Francisco López	2. Javier Alonso
B0012	1. Francisco López	3. Marta Rebolledo
C0014	1. Francisco López	2. Javier Alonso
D0120	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:

Empleados		
Id	Nombre	SuJefe
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
```

Consultas SQL de Unión Externas

Cómo funcionan y cómo se crean estas consultas en SQL.

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:

consulta 1,consulta 2, consulta n	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
  NuevasCuentas
UNION ALL
SELECT *
FROM
  Clientes
WHERE
  CantidadPedidos > 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 porqué 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
    NombreCompania, Ciudad
FROM
  Proveedores
WHERE
  Pais = 'Brasil'
UNION
  SELECT NombreCompania, Ciudad
  FROM Clientes
  WHERE Pais = 'Brasil'
```

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

```
SELECT
  NombreCompania, Ciudad
FROM
  Proveedores
WHERE
  Pais = 'Brasil'
UNION
  SELECT NombreCompania, Ciudad
  FROM Clientes
  WHERE Pais = '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
  NombreCompania, Ciudad
FROM
  Proveedores
WHERE
  Pais = 'Brasil'
UNION
  SELECT NombreCompania, Ciudad
  FROM Clientes
  WHERE Pais = 'Brasil'
UNION
  SELECT Apellidos, Ciudad
  FROM Empleados
  WHERE Region = '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_Clientes
UNION TABLE
  ListaProveedores
```

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

Consultas de acción

Explicamos detenidamente las sentencias delete, insert into y update, para la realización de consultas en las bases de datos.

Las consultas de acción son aquellas que no devuelven ningún registro, son las encargadas de acciones como añadir y borrar y modificar registros. Tanto las sentencias de actualización como las de borrado desencaderán (según el motor de datos) las actualizaciones en cascada, borrados en cascada, restricciones y valores por defecto definidos para los diferentes campos o tablas afectadas por la consulta.

DELETE

Crea una consulta de eliminación que elimina los registros de una o más de las tablas listadas en la cláusula FROM que satisfagan la cláusula WHERE. Esta consulta elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

```
DELETE FROM Tabla WHERE criterio
```

Una vez que se han eliminado los registros utilizando una consulta de borrado, no puede deshacer la operación. Si desea saber qué registros se eliminarán, primero examine los resultados de una consulta de selección que utilice el mismo criterio y después ejecute la consulta de borrado. Mantenga copias de seguridad de sus datos en todo momento. Si elimina los registros equivocados podrá recuperarlos desde las copias de seguridad.

```
DELETE
FROM
  Empleados
WHERE
  Cargo = 'Vendedor'
```

INSERT INTO

Agrega un registro en una tabla. Se la conoce como una consulta de datos añadidos. Esta consulta puede ser de dos tipo: Insertar un único registro ó Insertar en una tabla los registros contenidos en otra tabla.

Para insertar un único Registro:

En este caso la sintaxis es la siguiente:

```
 INSERT INTO Tabla (campo1, campo2, ..., campoN)
VALUES (valor1, valor2, ..., valorN)
```

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

Para seleccionar registros e insertarlos en una tabla nueva

En este caso la sintaxis es la siguiente:

```
SELECT campo1, campo2, ..., campoN INTO nuevatabla
FROM tablaorigen [WHERE criterios]
```

Se pueden utilizar las consultas de creación de tabla para archivar registros, hacer copias de seguridad de las tablas o hacer copias para exportar a otra base de datos o utilizar en informes que muestren los datos de un periodo de tiempo concreto. Por ejemplo, se podría crear un informe de Ventas mensuales por región ejecutando la misma consulta de creación de tabla cada mes.

Para insertar Registros de otra Tabla:

En este caso la sintaxis es:

```
INSERT INTO Tabla [IN base_externa] (campo1, campo2, , campoN)
SELECT TablaOrigen.campo1, TablaOrigen.campo2,,TablaOrigen.campoN FROM Tabla Origen
```

En este caso se seleccionarán los campos 1,2,..., n de la tabla origen 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 Tabla Origen poseen la misma estructura podemos simplificar la sintaxis a:

```
INSERT INTO Tabla SELECT Tabla Origen.* FROM Tabla Origen
```

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

En este tipo de consulta 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.

Se puede utilizar la instrucción INSERT INTO para agregar un registro único a una tabla, utilizando la sintaxis de la consulta de adición de registro único tal y como se mostró anteriormente. En este caso, su código especifica el nombre y el valor de cada campo del registro. Debe especificar cada uno de los campos del registro al que se le va a asignar un valor así como el valor para dicho campo. Cuando no se especifica dicho campo, se inserta el valor predeterminado o Null. Los registros se agregan al final de la tabla.

También se puede utilizar INSERT INTO para agregar un conjunto de registros pertenecientes a otra tabla o consulta utilizando la cláusula SELECT... FROM como se mostró anteriormente en la sintaxis de la consulta de adición de múltiples registros. En este caso la cláusula SELECT especifica los campos que se van a agregar en la tabla destino especificada.

La tabla destino u origen puede especificar una tabla o una consulta. Si la tabla destino contiene una clave principal, hay que asegurarse que es única, y con valores no nulos; si no es así, no se agregarán los registros. Si se agregan registros a una tabla con un campo Contador, no se debe incluir el campo Contador en la consulta. Se puede emplear la cláusula IN para agregar registros a una tabla en otra base de datos.

Se pueden averiguar los registros que se agregarán en la consulta ejecutando primero una consulta de selección que utilice el mismo criterio de selección y ver el resultado. Una consulta de adición copia los registros de una o más tablas en otra. Las tablas que contienen los registros que se van a agregar no se verán afectadas por la consulta de adición. En lugar de agregar registros existentes en otra tabla, se puede especificar los valores de cada campo en un nuevo registro utilizando la cláusula VALUES. Si se omite la lista de campos, la cláusula VALUES debe incluir un valor para cada campo de la tabla, de otra forma fallará INSERT.

Ejemplos

```
INSERT INTO
   Clientes
SELECT
   ClientesViejos.*
FROM
   ClientesNuevos
```

```
SELECT
   Empleados.*
INTO Programadores
FROM
  Empleados
WHERE
  Categoria = 'Programador'
```

Esta consulta crea una tabla nueva llamada programadores con igual estructura que la tabla empleado y copia aquellos registros cuyo campo categoria se programador

```
INSERT INTO
  Empleados (Nombre, Apellido, Cargo)
VALUES
  (
    'Luis', 'Sánchez', 'Becario'
   )
INSERT INTO
   Empleados
SELECT
  Vendedores.*
FROM
  Vendedores
WHERE
   Provincia = 'Madrid'
```

UPDATE

Crea una consulta de actualización que cambia los valores de los campos de una tabla especificada basándose en un criterio específico. Su sintaxis es:

```
  UPDATE Tabla SET Campo1=Valor1, Campo2=Valor2, CampoN=ValorN
  WHERE Criterio
```

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez. El ejemplo siguiente incrementa los valores Cantidad pedidos en un 10 por ciento y los valores Transporte en un 3 por ciento para aquellos que se hayan enviado al Reino Unido.:

```
UPDATE
   Pedidos
SET
   Pedido = Pedidos * 1.1,
  Transporte = Transporte * 1.03
WHERE
   PaisEnvío = 'ES'
```

UPDATE no genera ningún resultado. Para saber qué registros se van a cambiar, hay que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

```
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
```

Optimizar prestaciones I

Formas de indexar los campos y organizar los enlaces entre tablas para mejorar el rendimiento de las consultas.

Las bases de datos (BD) cuanto más extensas requieren una mayor atención a la hora de organizar sus contenidos. Cuando se trabaja con tablas de miles o decenas de miles de registros la búsqueda de un determinado dato puede resultar un proceso largo que ralentiza enormemente la creación de nuestra página.

Es por ello importante tener en cuenta una serie de aspectos indispensables para el mejor funcionanmiento de la base.

Gestión y elección de los índices

Los índices son campos elegidos arbitrariamente por el constructor de la BD que permiten la búsqueda a partir de dicho campo a una velocidad notablemente superior. Sin embargo, esta ventaja se ve contrarrestada por el hecho de ocupar mucha más memoria (el doble más o menos) y de requerir para su inserción y actualización un tiempo de proceso superior.

Evidentemente, **no podemos indexar todos los campos** de una tabla extensa ya que doblamos el tamaño de la BD. Igualmente, tampoco sirve de mucho el indexar todos los campos en una tabla pequeña ya que las selecciones pueden efectuarse rápidamente de todos modos.
Un caso en el que los índices pueden resultar muy útiles es cuando realizamos peticiones simultáneas sobre varias tablas. En este caso, el proceso de selección puede acelerarse sensiblemente si **indexamos los campos que sirven de nexo entre las dos tablas**. En el ejemplo de nuestra librería virtual estos campos serían id_cliente e id_articulo.
Los índices pueden resultar contraproducentes si los introducimos sobre campos triviales a partir de los cuales no se realiza ningún tipo de petición ya que, además del problema de memoria ya mencionado, estamos ralentizando otras tareas de la base de datos como son la edición, inserción y borrado. Es por ello que vale la pena pensarselo dos veces antes de indexar un campo que no sirve de criterio para búsquedas de los internautas y que es usado con muy poca frecuencia por razones de mantenimiento.

Gestión de los nexos entre tablas

El enlace entre tablas es uno de los puntos más peliagudos y que puede llevar a la absoluta ralentización de la base de datos a causa "pequeños" detalles que resultan ser fatales.

Imaginemos que trabajamos con una pequeña BD constituida por dos tablas de 1000 registros cada una. Imaginemos ahora una selección simultánea en la que imponemos la condición de que el valor un campo de la primera sea igual a de una segunda, algo que se realiza con mucha frecuencia. En este tipo de casos, la BD leerá y comparará cada valor de campo de una con cada valor de campo de la otra. Esto representaría un millón de lecturas. Este hecho podría agravarse si consultamos una tercera tabla al mismo tiempo y podría llegar a ser catastrófico si tenemos en cuenta que la BD esta siendo consultada por varios internautas al mismo tiempo.

Para evitar situaciones de colapso, es necesario **indexar cada uno de los campos que sirven de enlace entre esas tablas**. En el ejemplo de nuestra librería virtual, ya lo hemos dicho, estos campos serían id_cliente e id_articulo.
Además, resulta también de vital importancia el **definir esos campos de una forma estrictamente idéntica en cada una de las tablas**, es decir, el campo ha de ser de la misma naturaleza y características. No vale definirlo como real en una tabla y entero en otra o cambiar la longitud máxima para los alfanuméricos o que en una tabla sea de longitud constante y en otra variable...
El gestionar inteligentemente estos aspectos puede solucionarnos muchos quebraderos de cabeza y permitir a los internautas navegar más agradablemente por nuestro sitio.

Los campos, su naturaleza y dimensiones. Cómo gestionarlos para mejorar la eficiencia de la base de datos.

Gestión de los campos

Ya hemos comentado por encima los diferentes tipos de campo existentes en una base de datos. La elección del tipo de campo apropiado para cada caso puede ayudarnos también a optimizar el tamaño y rapidez de nuestra base de datos.

La preguntas que hay que hacerse a la hora de elegir la naturaleza y dimensiones del campo son:

-¿Qué tipo de dato voy a almacenar en el campo? Números, texto, fechas...

-¿Cuál es el tamaño máximo que espero que pueda alcanzar alguno de los registros del campo?

Hay que tener en cuenta que cuanto más margen le demos al valor máximo del campo, más aumentará el tamaño de nuestra base de datos y más tiempo tardara en realizar las consultas. Además, el factor tamaño puede verse agravado si estamos definiendo un campo indexado, para los cuales, el espacio ocupado es aproximadamente del doble.

Un consejo práctico es que las fechas sean almacenadas en formato de fecha ya que ello nos permite reducir el espacio que ocupan en memoria de más del doble y por otro lado, podremos aprovechar las prestaciones que SQL y nuestro lenguaje de servidor nos ofrecen. Podremos calcular la diferencia de días entre dos fechas, ordenar los registros por fecha, mostrar los registros comprendidos en un intervalo de tiempo...

Existe la posibilidad para los campos de texto de fijar una cierta longitud para el campo o dejar que cada registro tenga una longitud variable en función del número de carácteres que posea. Elegir campos de longitud variable nos puede ayudar a optimizar los recursos de memoria de la BD, no obstante, es un arma de doble filo ya que las consultas se realizan más lentamente puesto que obligamos a la tabla a establecer cuál es el tamaño de cada registro que se está comparando en lugar de saberlo de antemano. Es por tanto aconsejable, para los campos indexados de pequeño tamaño, atribuirles una longitud fija.

Eliminar palabras cortas y repeticiones

En situaciones en la que nuestra base de datos tiene que almacenar campos de texto extremadamente largos y dichos campos son requeridos para realizar selecciones del tipo LIKE '%algo%', los recursos de la BD pueden verse sensiblemente mermados. Una forma de ayudar a gestionar este tipo búsquedas es incluyendo un campo adicional.

Este campo adicional puede ser creado automáticamente por medio de scripts y en él incluiríamos el texto original, del cual habremos eliminado palabras triviales como artículos, preposiciones o posesivos. Nos encargaremos además de eliminar las palabras que estén repetidas. De esta forma podremos disminuir sensiblemente el tamaño del campo que va a ser realmente consultado.

Hemos comentado en otros capítulos que los campos texto de más de 255 caracteres denominados memo no pueden ser indexados. Si aún después de esta primera filtración nuestro campo continua siendo demasiado largo para ser indexado, lo que se puede hacer es cortarlo en trozos de 255 caracteres de manera a que lo almacenemos en distintos campos que podrán ser indexados y por tanto consultados con mayor rapidez.

SQL con Oracle

Antes de empezar me gustaría decir que este curso esta basado en Oracle, es decir los ejemplos expuestos y material se han utilizado sobre Oracle. Por otro lado decir que pienso que es interesante saber algo de SQL antes de comenzar con MYSQL, ya que, aunque existen algunos cambios insignificantes, sabiendo manejar SQL sabes manejar MYSQL.

Caracteristicas:

SQL: Structured query language.

- Permite la comunicación con el sistema gestor de base de datos.
- En su uso se puede especificar que quiere el usuario.
- Permite hacer consulta de datos.

Tipos de datos:

CHAR:

- Tienen una longitud fija.
- Almacena de 1 a 255.
- Si se introduce una cadena de menos longitud que la definida se rellenara con blancos a la derecha hasta quedar completada.
- Si se introduce una cadena de mayor longitud que la fijada nos dará un error.

VARCHAR:

- Almacena cadenas de longitud variable.
- La longitud máxima es de 2000 caracteres.
- Si se introduce una cadena de menor longitud que la que está definida, se almacena con esa longitud y no se rellenara con blancos ni con ningún otro carácter a la derecha hasta completar la longitud definida.
- Si se introduce una cadena de mayor longitud que la fijada, nos dará un error

NUMBER:

- Se almacenan tanto enteros como decimales.
- Number (precisión, escala)
- Ejemplo:
  X=number (7,2)
  X=155'862 à Error ya que solo puede tomar 2 decimales
  X= 155'86 à Bien

Nota: El rango máximo va de 1 a 38.

LONG:

- No almacena números de gran tamaño, sino cadenas de caracteres de hasta 2 GB

DATE:

- Almacena la fecha. Se almacena de la siguiente forma:

Siglo/Año/Mes/Día/Hora/Minutos/Segundos

RAW:

- Almacena cadenas de Bytes (gráficos, sonidos…)

LONGRAW:

- Como el anterior pero con mayor capacidad.

ROWID:

- Posición interna de cada una de las columnas de las tablas.
- Sentencias de consultas de datos
  Select:
  Select [ALL | Distinct] [expresión_columna1, expresión_columna2, …., | *]
  From [nombre1, nombre_tabla1, …, nombre_tablan]
  {[Where condición]
  [Order By expresión_columna [Desc | Asc]…]};

Vamos a explicar como leer la consulta anterior y así seguir la pauta para todas las demás. Cuando ponemos [] significa que debemos la que va dentro debe existir, y si además ponemos | significa que deberemos elegir un valor de los que ponemos y no mas de uno. En cambio si ponemos {} significa que lo que va dentro de las llaves puede ir o no, es decir es opcional y se pondrá según la consulta.

Nota: En el select el valor por defecto entre ALL y DISTINCT es ALL.

- Alias = El nuevo nombre que se le da a una tabla. Se pondrá entre comillas
- Order By = Ordena ascendentemente (Asc) (valor por defecto) o descendentemente (Desc).
- All = Recupera todas las filas de la tabla aunque estén repetidas.
- Distinct = Solo recupera las filas que son distintas.
- Desc Emple; = Nos da un resumen de la tabla y sus columnas. En este caso de la tabla Emple.
- Not Null= Si aparece en una lista de una columna significa que la columna no puede tener valores nulos.
- Null= Si está nulo.

Nota: Nótese que cada consulta de SQL que hagamos hemos de terminarla con un punto y coma ";".

Varios ejemplos para verlo mas claro:

```
SELECT JUGADOR_NO, APELLIDO, POSICION, EQUIPO
FROM JUGADORES
WHERE EQUIPO_NO = 'VALENCIA'
ORDER BY APELLIDO;
```

Este ejemplo mostrar el número de jugador (jugador_no) el apellido (Apellido), la posición en la que juega (Posición), y el equipo (Equipo) al que pertenece.
Seleccionara todos los datos de la tabla jugadores donde (Where) el nombre de equipo (Equipo_No) sea igual que la palabra 'Valencia' y se ordenara (order by) apellido. Notemos también que no pone ni 'Distinct' ni 'All'. Por defecto generara la sentencia con ALL.

```
SELECT *
FROM JUGADORES
WHERE POSICION = 'DELANTERO'
ORDER BY JUGADOR_NO;
```

Este ejemplo muestra todos los campos de la tabla jugadores donde (Where) la posición sea igual que 'Delantero' y lo ordena por número de jugador. Al no poner nada se presupone que es ascendentemente (Asc).

```
SELECT *
FROM JUGADORES
WHERE EQUIPO_NO = 'VALENCIA' AND POSICION = 'DELANTERO'
ORDER BY APELLIDO DESC, JUGADOR_NO ASC;
```

En este ejemplo selecciona todos los campos de la tabla jugadores donde (Where) el nombre del equipo sea igual a 'Valencia' y la posición de los jugadores sea igual a 'Delantero'. Por último los ordena por 'Apellido' descendentemente y por número de jugador ascendentemente.