Base de Datos

4. Consultas de Seleccion SImple

Curso 2022-2023

Tabla de contenido

La mejor forma de iniciar el estudio del lenguaje SQL es ejecutar consultas sencillas en la base de datos. Sin embargo, antes conviene conocer los orígenes y evolución que ha tenido este lenguaje, los diferentes tipos de sentencias SQL existentes (subdivisión del lenguaje SQL), así como los diferentes tipos de datos (números, fechas, cadenas…) que nos podemos encontrar almacenadas en las bases de datos. Y, entonces, ya podremos iniciarnos en la ejecución de consultas simples.

1. ORÍGENES Y EVOLUCIÓN DEL LENGUAJE SQL BAJO LA GUÍA DE LOS SGBD

El modelo relacional en el que se basan los SGBD actuales fue presentado en 1970 por el matemático Edgar Frank Codd, que trabajaba en los laboratorios de investigación de la empresa de informática IBM. Uno de los primeros SGBD relacionales en aparecer fue el System R de IBM, que se desarrolló como prototipo para probar la funcionalidad del modelo.

Una vez comprobada la eficiencia del modelo relacional y del lenguaje SQL, se inició una dura carrera entre distintas marcas comerciales. Así, tenemos lo siguiente:

- IBM comercializa varios productos relacionales con el lenguaje SQL: System/38 en 1979, SQL/DS en 1981, y DB2 en 1983.
- Relational Software, Inc. (actualment, Oracle Corporation) crea la su propia versió de SGBD relacional para a la Marina de los EEUU, la CIA y otros.

El lenguaje SQL evolucionó (cada marca comercial seguía su propio criterio) hasta que los principales organismos de estandarización intervinieron para obligar a los diferentes SGBD relacionales a implementar una versión común del lenguaje y, así, en 1986 el ANSI (American National Standards Institute) publica el estándar SQL-86, que en 1987 es ratificado por el ISO (Organización Internacional para la Normalización, o International Organization for Standardization en inglés).

2. TIPO DE SENTENCIAS SQL

Términos en inglés

En la jerga informática se utilizan los siguientes términos:

- Lenguaje de definición de datos, abreviado **DDL**
- Lenguaje de control de datos, abreviado **DCL**
- Query language, abreviado QL
- Lenguaje de manipulación de datos, abreviado **DML**

Los SGBD relacionales incorporan el lenguaje SQL para ejecutar distintos tipos de tareas en las bases de datos: definición de datos, consulta de datos, actualización de datos, definición de usuarios, concesión de privilegios… Por este motivo, las sentencias que aporta el lenguaje SQL suelen agruparse en las siguientes:

- Sentencias destinadas a la definición de los datos (**DDL**), que permiten definir los objetos (tablas, campos, valores posibles, reglas de integridad referencial, restricciones…).
- Sentencias destinadas al control sobre los datos (**DCL**), que permiten conceder y retirar permisos sobre los distintos objetos de la base de datos.
- Sentencias destinadas a la consulta de los datos (**QL**), que permiten acceder a los datos en modo consulta.
- Sentencias destinadas a la manipulación de los datos (**DML**), que permiten actualizar la base de datos (altas, bajas y modificaciones).

En algunos SGBD no existe distinción entre **QL** y **DML**, y únicamente se habla de **DML** para las consultas y actualizaciones. Del mismo modo, en ocasiones se incluyen las sentencias de control (**DCD**) junto con las de definición de datos (**DDL**). No tiene importancia que se incluyan en un grupo o que sean un grupo propio: es una simple clasificación.

3. TIPO DE DATOS

La evolución anárquica que ha seguido el lenguaje SQL ha hecho que cada SGBD haya tomado sus decisiones en cuanto a los tipos de datos permitidos. Ciertamente, los diferentes estándares SQL que han ido apareciendo han marcado una cierta línea y los SGBD se acercan a ellos, pero tampoco pueden dejar de apoyar los tipos de datos que han proporcionado a lo largo de su existencia, ya que hay muchas bases de datos repartidas por el mundo que las utilizan.

De todo esto debemos deducir que, para trabajar con un SGBD, debemos conocer los principales tipos de datos que facilita (numéricas, alfanuméricas, momentos temporales…) y debemos hacerlo centrándonos en un SGBD concreto (la nuestra elección ha sido MySQL) teniendo en cuenta que el resto de SGBD también incorpora tipos de datos similares y, en caso de tener que trabajar, siempre deberemos echar un vistazo a la documentación que cada SGBD facilita.

Cada valor manipulado por un determinado SGBD corresponde a un tipo de dato que asocia un conjunto de propiedades al valor. Las propiedades asociadas a cada tipo de dato hacen que un SGBD concreto trate de forma diferente los valores de distintos tipos de datos.

En el momento de creación de una tabla, debe especificarse un tipo de dato para cada una de las columnas. En la creación de una acción o función almacenada en la base de datos, es necesario especificar un tipo de dato para cada argumento. La correcta asignación del tipo de dato es fundamental porque los tipos de datos definen el dominio de valores que cada columna o argumento puede contener. Así, por ejemplo, las columnas tipo DATE no podrán aceptar el valor '30 de febrero ' ni el valor 2 ni la cadena Hola .

Dentro de los tipos de datos básicos, podemos distinguir los siguientes:

- Tipo de datos para gestionar información alfanumérica.
- Tipo de datos para gestionar información numérica.
- Tipo de datos para gestionar momentos temporales (fechas y tiempo).
- Otros tipos de datos.

MySQL es el SGBD con el que se trabaja en estos materiales y el lenguaje SQL de MySQL lo que se describe. La notación que se utiliza, en cuanto a sintaxis de definición del lenguaje, habitual, consiste en poner entre corchetes ([ ]) los elementos opcionales, y separar con el carácter | los elementos alternativos.

3.1. Tipo de datos string

Los tipos de datos string almacenan datos alfanuméricos en el conjunto de caracteres de la base de datos. Estos tipos son menos restrictivos que otros tipos de datos y, en consecuencia, tienen menos propiedades. Así, por ejemplo, las columnas de tipo carácter pueden almacenar valores alfanuméricos -letras y cifras-, pero las columnas de tipo numérico sólo pueden almacenar valores numéricos.

MySQL proporciona los siguientes tipos de datos para gestionar datos alfanuméricos:

- CHAR
- VARCHAR

- BINARY
- VARBINARY
- BLOB
- TEXT
- ENUM
- SET

3.1.1. El tipo CHAR [(largo)]

Este tipo especifica una cadena de longitud fija (indicada por longitud) y, por tanto, MySQL asegura que todos los valores almacenados en la columna tienen la longitud especificada. Si se inserta una cadena de longitud más corta, MySQL la rellena con espacios en blanco hasta la longitud indicada. Si se intenta insertar una cadena de longitud más larga, se trunca.

La longitud mínima y por defecto (no es obligatoria) para una columna de tipos CHAR es de 1 carácter, y la longitud máxima permitida es de 255 caracteres.

Para indicar la longitud, debe especificarse con un número entre paréntesis, que indica el número de caracteres, que tendrá el string. Por ejemplo CHAR (10).

3.1.2. El tipo VARCHAR (largo)

Este tipo especifica una cadena de longitud variable que puede ser, como máximo, la indicada por longitud, valor que es obligatorio introducir.

Los valores de tipo VARCHAR almacenan el valor exacto que indica el usuario sin añadir espacios en blanco. Si se intenta insertar una cadena de longitud más larga, VARCHAR devuelve un error.

La longitud máxima de este tipo de datos es de 65.535 caracteres.

La longitud puede indicarse con un número, que indica el número de caracteres máximo que contendrá el string. Por ejemplo: VARCHAR (10).

El tipo de datos alfanumérico más habitual para almacenar strings en bases de datos MySQL es **VARCHAR**.

3.1.3. El tipo BINARY (largo)

El tipo de dato BINARYes similar al tipo CHAR, pero almacena caracteres en binario. En este caso, la longitud mpre se indica en bytes. La longitud mínima para una columna BINARY es de 1 byte. La longitud máxima permitida es de 255.

3.1.4. El tipo VARBINARY (largo)

El tipo de dato VARBINARYes similar al tipo VARCHAR, pero almacena caracteres en binario. En ese caso, la longitud siempre se indica en bytes. Los bytes que no se rellenan explícitamente se rellenan con '@IOCCONTENT@' .

Así, por ejemplo, una columna definida como a VARBINARY (4)la que se asigne el valor 'a' contendrá, realmente, 'a@IOCCONTENT@@IOCCONTENT@@IOCCONTENT@' y habrá que tenerlo en cuenta en el ' hora de hacer, por ejemplo, comparaciones, ya que no será lo mismo comparar la columna con el valor 'a' que con el valor 'a@IOCCONTENT@@IOCCONTENT@@IOCCONTENT@' .

El valor '@IOCCONTENT@' en hexadecimal se corresponde con 0x00 .

3.1.5. El tipo BLOB

El tipo de datos BLOB es un objeto que permite contener una gran cantidad y variable de datos de tipo binario.

De hecho, se puede considerar un dato de tipo BLOB como un dato de tipo VARBINARY, pero sin limitación en cuanto al número de bytes. De hecho, los valores de tipos se BLOB almacenan en un objeto separado del resto de columnas de la tabla, debido a sus requerimientos de espacio.

Realmente, hay varios subtipos de BLOG: TINYBLOB, BLOB, MEDIUMBLOBy LONGBLOB.

- TINYBLOB puede almacenar hasta 2 ^ 8 + 2 bytes
- BLOB puede almacenar hasta 2 ^ 16 + 2 bytes
- MEDIUMBLOB puede almacenar hasta 2 ^ 24 + 3 bytes
- LONGBLOB puede almacenar hasta 2 ^ 32 + 4 bytes

3.1.6. El tipo TEXTO

El tipo de datos TEXTes un objeto que permite contener una cantidad grande y variable de datos de tipo carácter.

De hecho, se puede considerar un dato de tipo TEXTcomo un dato de tipo VARCHAR, pero sin limitación en cuanto al número de caracteres. De forma similar al tipo BLOB, los valores tipos TEXTtambién se almacenan en un objeto separado del resto de columnas de la tabla, debido a sus requerimientos de espacio.

Realmente, hay varios subtipos de TEXT: TINYTEXT, TEXT, MEDIUMTEXTy LONGTEXT:

- TINYTEXT puede almacenar hasta2 ^ 8 + 2 bytes
- TEXT puede almacenar hasta2 ^ 16 + 2 bytes
- MEDIUMTEXT puede almacenar hasta2 ^ 24 + 3 bytes
- LONGTEXT puede almacenar hasta2 ^ 32 + 4 bytes

3.1.7. El tipo ENUM ('cadena1' [,'cadena2'] ... [,'cadena_n'])

El tipo ENUMdefine un conjunto de valores tipo string con una lista prefijada de cadenas que se definen en el momento de la definición de la columna y que se corresponderán con los valores válidos de la columna.

Ejemplo de columna tipo ENUM

```
CREATE TABLE sizes (name ENUM ('small', 'medium', 'large'));
```

El conjunto de valores son obligatoriamente literales entre comillas simples.

Si una columna se declara de tipo ENUM y se especifica que no admite valores nulos, entonces, el valor predeterminado será el primero de la lista de cadenas.

El número máximo de cadenas distintas que puede soportar el tipo ENUM es 65535.

3.1.8. El tipo SET ('cadena1' [,'cadena2'] ... [,'cadena_n'])

Una columna tipo SET puede contener cero o más valores, pero todos los elementos que contenga deben pertenecer a una lista especificada en el momento de la creación.

El número máximo de valores distintos que puede soportar el tipo SET es 64.

Por ejemplo, se puede definir una columna tipo SET('one', 'two'). Y un elemento concreto puede tener cualquiera de los siguientes valores:

- ' '
- 'una'
- 'dos'
- 'uno,dos' (el valor 'dos,uno' no se prevé porque el +++++++ de los elementos no afecta a las listas)

3.2. Tipo de datos numéricos

MySQL soporta todos los tipos de datos numéricos de SQL estándar:

- INTEGER (también abreviado por INT)
- SMALLINT
- DECIMAL(también abreviado por DEC o FIXED)
- NUMERIC

También soporta:

- FLOAT
- REAL
- DOUBLE PRECISION (también llamado DOUBLE, simplemente, o REAL)
- BIT
- BOOLEAN

3.2.1. Los tipos de datos INTEGER

El tipo INTEGER (comúnmente abreviado como INT) almacena valores enteros. Existen varios subtipos de enteros en función de los valores admitidos (vea la siguiente tabla).

Tipo de entero	Almacena (en bytes)	Valor mínimo (con signo/sin signo)	Valor máximo (con signo/sin signo)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	8	-9223372036854775808 / 0	9223372036854775807/ 18446744073709551615

Los tipos de datos enteros admiten la especificación del número de dígitos que se deben mostrar de un valor concreto, utilizando la sintaxis:

INT (N), donde N es el número de dígitos visibles.

Así pues, si se especifica una columna de tipo INT (4), en el momento de seleccionar un valor concreto, se mostrarán tan sólo 4 dígitos. Hay que tener en cuenta que esta especificación no condiciona el valor almacenado, tan sólo fija el valor que se debe mostrar.

También se puede especificar el número de dígitos visibles en los subtipos de INTEGER, utilizando la misma sintaxis.

Para almacenar datos de tipo entero en bases de datos MySQL lo habitual es utilizar el tipo de datos INT.

3.2.2. Tipo FLOAT, REAL y DOUBLE

FLOAT, REALy DOUBLE son los tipos de datos numéricos que almacenan valores numéricos reales (es decir, que admiten decimales).

Los tipos FLOAT y REAL se almacenan en 4 bytes y los DOUBLE en 8 bytes.

Los tipos FLOAT, REAL o DOUBLE PRECISION admiten que se especifiquen los dígitos de la parte entera (E) y los dígitos de la parte decimal (D, que pueden ser 30 como máximo, y nunca mayores que E-2). La sintaxis para esta especificación sería:

FLOAT(E,D)

REAL(E,D)

DOUBLE PRECISION(E,D)

Ejemplo de almacenamiento en FLOAT

Por ejemplo, si se define una columna como FLOAT(7,4) y se desea almacenar el valor 999.00009, el valor almacenado realmente será 999.0001, que es el valor más cercano (aproximado) al original.

3.2.3. Tipo de datos DECIMAL y NUMERIC

DECIMAL y NUMERIC son los tipos de datos reales de punto fijo que admite MySQL. Son sinónimos y, por tanto, se pueden utilizar indistintamente.

Los valores en punto fijo no se almacenarán nunca de forma redondeada, es decir, que si es necesario almacenar un valor en un espacio que no es adecuado, emitirá un error. Este tipo de datos permite asegurar que el valor es exactamente el que se ha introducido. No se ha redondeado. Por tanto, se trata de un tipo de datos muy adecuado para representar valores monetarios, por ejemplo.

Ambos tipos de datos permiten especificar el total de dígitos (T) y la cantidad de dígitos decimales (D), con la siguiente sintaxis:

DECIMAL (T,D)

NUMERIC (T,D)

Valores posibles para NUMERIC

Por ejemplo uno NUMERIC (5,2) podría contener valores desde -999.99 hasta 999.99 .

También se admite la sintaxis DECIMAL (T) y NUMERIC (T) que es equivalente a DECIMAL (T,0) y NUMERIC (T,0).

El valor predeterminado de T es 10 y su valor máximo es 65.

3.2.4. Tipo de datos BIT

BIT es un tipo de datos que permite almacenar bits, desde 1 (por defecto) hasta 64. Para especificar el número de bits que almacenará es necesario definirlo, siguiendo la siguiente sintaxis:

BIT (M), en el que M es el número de bits que se almacenarán.

Los valores literales de los bits se dan siguiendo el siguiente formato: b'valor_binario'. Por ejemplo, un valor binario admisible para un campo tipo BIT sería b'0001' .

Si damos el valor b'1010' a un campo definido como BIT (6) el valor que almacenará será b'001010'. Añadirá, pues, ceros a la izquierda hasta completar el número de bits definidos en el campo.

BIT es un sinónimo de TINYINT(1).

3.2.5. Otros tipos numéricos

BOOL o BOOLEAN es el tipo de datos que permite almacenar tipos de datos booleanos (que permiten los valores cierto o falso). BOOL o BOOLEAN es sinónimo de TINYINT (1). Almacenar un valor de cero se considera falso. En cambio, un valor distinto a cero se interpreta como cierto.

3.2.6. Modificadores de tipos numéricos

Hay algunas palabras clave que pueden añadirse a la definición de una columna numérica (entera o real) para condicionar los valores que contendrán.

- UNSIGNED: con este modificador sólo se admitirán los valores de tipo numérico no negativos.
- ZEROFILL: se añadirán ceros a la izquierda hasta completar el total de dígitos del valor numérico, si es necesario.
- AUTO_INCREMENT: cuando se añade un valor 0 o NULL en esa columna, el valor que se almacena es el valor más alto incrementado en 1. El primer valor por defecto es 1.

3.3. Tipo de datos para momentos temporales

El tipo de datos que MySQL dispone para almacenar datos que indiquen momentos temporales son:

- DATETIME
- DATE
- TIMESTAMP
- TIME
- YEAR

Tenga en cuenta que cuando hace falta referirse a los datos referentes a un año es importante explicitar sus cuatro dígitos. Es decir, que para expresar en el año 98 del siglo XX lo mejor es referirse a ello explícitamente así: 1998.

Si se utilizan dos dígitos en lugar de cuatro para expresar años, debe tenerse en cuenta que MySql los interpretará de la siguiente manera:

Los valores de los años entre 00-69 se interpretan como los años: 2000-2069.

Los valores de los años entre 70-99 se interpretan como los años: 1970-1999.

3.3.1. El tipo de dato DATE

DATE permite almacenar fechas. El formato de una fecha en MySQL es 'AAAA-MM-DD', en el que AAAA indica el año expresado en cuatro dígitos, MM indica el mes expresado en dos dígitos y DD indica el día expresado en dos dígitos.

La fecha mínima soportada por el sistema es '1000-01-01'. Y la fecha máxima admisible en MySQL es '9999-12-31'.

3.3.2. El tipo de dato DATETIME

DATETIME es un tipo de dato que permite almacenar combinaciones de días y horas.

El formato de DATETIME en MySQL es 'AAAA-MM-DD HH:MM:SS', en el que AAAA-MM-DD es el año, el mes y el día, y HH:MM:SS indican la hora, minuto y segundo, expresados en dos dígitos, separados por ':'.

Los valores válidos para los campos de tipos DATETIME van desde '1000-01-01 00:00:00' hasta '9999-12-31 23:59:59'.

3.3.3. El tipo de dato TIMESTAMP

TIMESTAMP es un tipo de dato similar a DATETIME y tiene el mismo formato por defecto: 'AAAA-MM-DD HH:MM:SS'. TIMESTAMP sin embargo, permite almacenar la hora y fecha actuales en un momento determinado.

Si se asigna el valor NULL a una columna TIMESTAMP o no se le asigna ninguna explícitamente, el sistema almacena por defecto la fecha y hora actuales. Si se especifica una fecha-hora concreta en la columna, entonces la columna tomará ese valor, como si se tratara de una columna DATETIME.

El rango de valores que admite TIMESTAMP es de '1970-01-01 00:00:01' a '2038-01-19 03:14:07'.

Una columna TIMESTAMP es útil cuando se desea almacenar la fecha y hora en el momento de añadir o modificar un dato en la base de datos, por ejemplo.

Si en una tabla hay más de una columna de tipos TIMESTAMP, el funcionamiento de la primera columna TIMESTAMP es el esperable: se almacena la fecha y hora de la operación más reciente, salvo que explícitamente se asigne un valor concreto, y, entonces, prevalece el valor especificado. El resto de columnas TIMESTAMP de una misma tabla no se actualizará con ese valor. En ese caso, si no se especifica un valor concreto, el valor almacenado será cero.

Ejemplo de creació de tabla utilizando TIMESTAMP

Para crear una tabla con una columna tipo TIMESTAMP son equivalentes las siguientes sintaxis:

- CREATE TABLE t (ts TIMESTAMP);
- CREATE TABLE t (ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
- CREATE TABLE t (ts TIMESTAMP ON UPDATE CURRENT_TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

3.3.4. Tipo de datos TIME

TIME es un tipo de dato específico que almacena la hora en el formato 'HH:MM:SS'.

TIME sin embargo, también permite expresar el tiempo transcurrido entre dos momentos (diferencia de tiempo). Por eso, los valores que permite almacenar son de '-838:59:59' a '838:59:59'. En este caso, el formato será 'HHH:MM:SS'.

Otros formatos también admitidos para un dato de tipo TIMEson: 'HH:MM', 'D HH:MM:SS', 'D HH:MM', 'D HH', o 'SS', donde D indica los días. Es posible, también, un formato que admite microsegundos 'HH:MM:SS.uuuuuu' en el que uuuuuu son los microsegundos.

3.3.5. Tipo de datos YEAR

YEAR es un dato de tipos BYTE que almacena datos de tipo año. El formato por defecto es AAAA (el año expresado en cuatro dígitos) o bien AAAA, expresado como string.

También se pueden utilizar los tipos YEAR (2) o YEAR (4), para especificar columnas de tipo año expresado con dos dígitos o año con cuatro dígitos.

Se admiten valores desde 1901 hasta 2155. También se admite 0000. En el formato de dos dígitos, se admiten los valores del 70 al 69, que representan los años de 1970 a 2069.

3.4. Otros tipos de datos

En MySQL existen extensiones que permiten almacenar otros tipos de datos: los datos poligonales.

Así, MySQL permite almacenar datos de tipo objeto poligonal y da implementación al modelo geométrico del estándar OpenGIS.

Por tanto, podemos definir columnas MySQL de tipo Polygon, Point, Curve o Line, entre otras.

4. Consultas simples

Una vez ya conocemos los diferentes tipos de datos que podemos encontrarnos almacenados en una base de datos (nosotros nos hemos centrado en MySQL, pero en el resto de SGBD es similar), estamos en condiciones de iniciar la explotación de la base de datos, es decir, empezar la gestión de los datos. Evidentemente, para poder gestionar datos, previamente es necesario haber definido las tablas que deben contener los datos, y para poder consultar datos es necesario haberlos introducido antes. Sin embargo, el aprendizaje del lenguaje SQL se efectúa en sentido inverso; es decir, empezaremos conociendo las posibilidades de consulta de datos sobre tablas ya creadas y con datos ya introducidos. Sin embargo, necesitamos conocer la estructura de las tablas que deben gestionarse y las relaciones existentes entre ellas.

Las tablas que gestionaremos forman parte de dos diseños distintos, es decir, son tablas de temas disjuntos. Veámoslas.

1. ESQUEMA EMPRESA

La figura 1 muestra el diseño del esquema de la Compañía, implementado mediante el uso de la opción "Diseñador" de modelado de fechas *phpMyAdmin*, que utiliza un modo de notación intuitiva.

> **Nota:**
>
> Los datos del diseño Entidad-Relación del tema ***empresa***, los gestionaremos a lo largo de esta unidad.

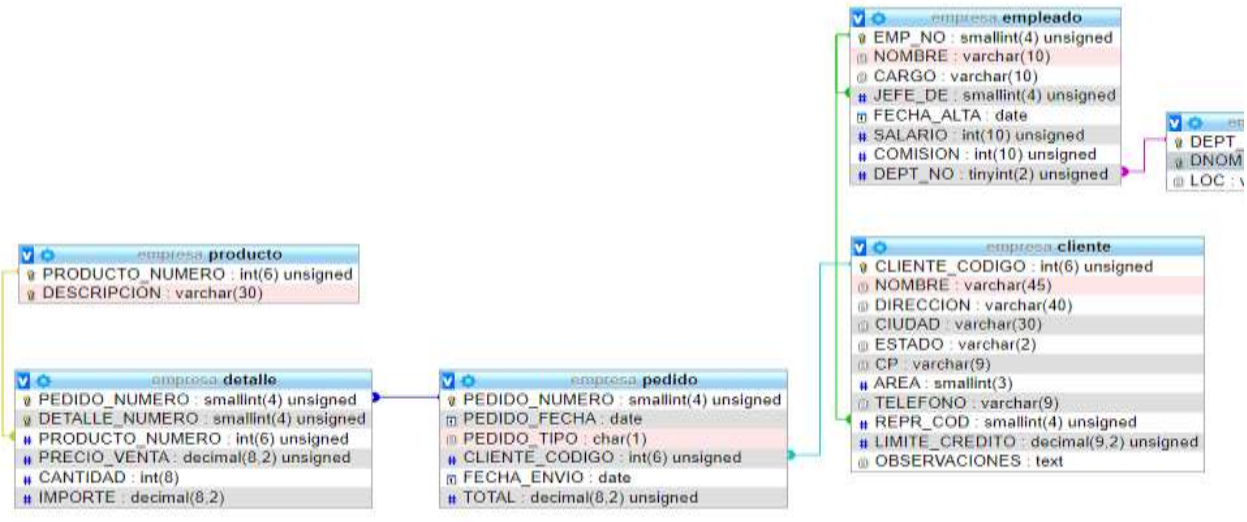

Figura 1 Diseño del Esquema Empresa

Tenga presente que la figura 1 muestra un esquema similar a los diagramas Entidad-Relación o diseño Chen. Echando un vistazo rápido a este diseño, debemos interpretar lo siguiente:

- Tenemos seis entidades diferentes: departamentos (DEPARTAMENTO), empleados (EMPLEADO), clientes (CLIENTE), productos (PRODUCTO), órdenes (PEDIDO) y detalle de las órdenes (DETALLE).
- Entre las seis entidades se establecen relaciones:

- Entre DEPARTAMENTO y EMPLEADO (relación 1:N), puesto que un empleado es asignado obligatoriamente a un departamento, y un departamento tiene asignados cero o varios empleados.
- Entre EMPLEADO y EMPLEADO (relación reflexiva 1:N), ya que un empleado puede depender de otro empleado de la empresa, y un empleado puede ser jefe de cero o varios empleados.
- Entre EMPLEADO y CLIENTE (relación 1:N), puesto que un empleado puede ser el representante de cero o varios clientes, y un cliente puede tener asignado un representante que debe ser un empleado de la empresa.
- Entre CLIENTE y PEDIDO (relación 1:N), ya que un cliente puede tener cero o varios pedidos en la empresa, y un pedido es obligatoriamente de un cliente.
- Entre PEDIDO y DETALLE (relación fuerte-débil 1:N), ya que un pedido está formado por varias líneas, llamadas detalle del pedido.
- Entre DETALLE y PRODUCTO (relación N:1), puesto que cada línea de detalle corresponde a un producto.

En ocasiones, algún alumno no experto en diseños Entidad-Relación se pregunta el porqué de la entidad DETALLE y piensa que no debería estar, y la sustituye por una relación N:N entre las entidades PEDIDO y PRODUCTO. Gran error. El error radica en que en una relación N:N entre PEDIDO y PRODUCTO, un mismo producto no puede estar más de una vez en el mismo pedido. En ciertos negocios, esto puede ser una decisión acertada, pero no siempre es así, puesto que se pueden dar situaciones similares a las siguientes:

- Por razones comerciales o de otra índole, en un misma pedido existe cierta cantidad de un producto con un precio y descuentos determinados, y otra cantidad del mismo producto con unas condiciones de venta (precio y/o descuentos) diferentes.
- Puede que una cantidad de producto deba entregarse en una fecha, y otra cantidad del mismo producto en otra fecha. En esta situación, la fecha de envío debería residir en cada línea de detalle.

La traducción correspondiente al modelo relacional, considerando los atributos subrayados como clave primaria y el símbolo (VN) que indica que admite valores nulos, es la siguiente:

```
DEPARTAMENTO (Dept no, Dnom, Loc (VN) )
EMPLEADO (Emp No, Nombre, Cargo (VN), Jefe_De (VN), Fecha_Alta (VN),
Salario (VN), Comisión (VN), Dept_No )  DÓNDE Dept_No REFERENCIA
DEPARTAMENTO
CLIENTE (Client Codigo, Nombre, Direccion, Ciudad, Estado(VN), CP, Área
(VN), Teléfono (VN), Repr_Cod (VN), Límite_Credito (VN), Observaciones
(VN))  DONDE Repr_Cod REFERENCIA Repr_Cod
PRODUCTO (Producto Numero, Descripción )
PEDIDO (Pedido Numero, Pedido_Fecha (VN), Pedido_Tipo (VN),
Cliente_Codigo, Fecha_Envío (VN), Total (VN)) DONDE  Cliente_Codigo
REFERENCIA CLIENTE
DETALLE (Pedido Numero, Detalle Numero, Producto_Numero, Precio_Venta (
VN ), Cantidad ( VN ), Importe (VN))  DONDE Pedido_Numero REFERENCIA
PEDIDO y Producto_Numero REFERENCIA PRODUCTO
```

La implementación de este modelo relacional en MySQL ha provocado las siguientes tablas:

Tabla **DEPARTAMENTO**, que contiene los departamentos de la empresa

Nombre	Null?	Tipo	Descripción
DEPT_NO	NOT NULL	INT(2)	Número de departamento de la empresa
DNOM	NOT NULL	VARCHAR(14)	Descripción del departamento
LOC		VARCHAR(14)	Localidad del departamento

Tabla **EMPLEADO**, que contiene los empleados de la empresa

Nombre	Null?	Tipo	Descripción
EMP_NO	NOT NULL	INT(4)	Número de empleado de la empresa
NOMBRE	NOT NULL	VARCHAR(10)	Apellido del empleado
CARGO		VARCHAR(10)	Oficio del empleado
JEFE_DE		INT(4)	Número del empleado que es el jefe directo (tabla EMPLEADO)
FECHA_ALTA		DATE	Fecha_Alta
SALARIO		INT(10)	Salario mensual
COMISION		INT(10)	Importe de las comisiones
DEPTA_NO	NOT NULL	INT(2)	Departamento al que pertenece (tabla DEPARTAMENTO)

Tabla **CLIENTE**, que contiene los clientes de la empresa

Nombre	Null?	Tipo	Descripción
CLIENTE_CODIGO	NOT NULL	INT(6)	Código de cliente
NOMBRE	NOT NULL	VARCHAR(45)	Nombre del cliente
DIRECCIÓN	NOT NULL	VARCHAR(40)	Dirección del cliente
CIUDAD	NOT NULL	VARCHAR(30)	Ciudad del cliente
ESTADO		VARCHAR(2)	País del cliente
CP	NOT NULL	VARCHAR(9)	Código postal del cliente
AREA		INT(3)	Área telefónica
TELEFONO		VARCHAR(9)	Teléfono del cliente
REPR_COD		INT(4)	Código del representante del cliente. Es uno de los empleados de la empresa (tabla EMPLEADO)
LIMITE_CREDITO		DECIMAL(9,2)	Límite de crédito de que dispone el cliente

OBSERVACIONES		TEXT	Observaciones

Tabla **PRODUCTO**, que contiene los productos a vender

Nombre	Null?	Tipo	Descripción
PRODUCTO_NUMERO	NOT NULL	INT(6)	Código de producto
DESCRIPCIÓN	NOT NULL	VARCHAR(30)	Descripción del producto

Tabla **PEDIDO**, que contiene las órdenes de venta

Nombre	Null?	Tipo	Descripción
PEDIDO_NUMERO	NOT NULL	INT(4)	Número de orden de venta
PEDIDO_FECHA		DATE	Fecha de la orden de venta
PEDIDO_TIPO		VARCHAR(1)	Tipo de Pedido - Valores válidos: A, B, C
CLIENTE_CODIGO	NOT NULL	INT(6)	Código del cliente que efectúa el comando (tabla CLIENTE)
FECHA_ENVÍO		DATE	Fecha de envío del comando
TOTAL		DECIMAL(8,2)	Importe total del pedido

Tabla **DETALLE**, que contiene el detalle de las órdenes de venta

Nombre	Null?	Tipo	Descripción
PEDIDO_NUMERO	NOT NULL	INT(4)	Número de orden de venta (Tabla Pedido)
DETALLE_NUMERO	NOT_NULL	INT(4)	Número de línea de detalle
PRODUCTO_NUMERO	NOT NULL	INT(6)	Código del producto de la línea (tabla PRODUCTO)
PRECIO_VENTA		DECIMAL(8,2)	Precio de venta del producto
CANTIDAD		INT(8)	Cantidad de producto a vender
IMPORTE		DECIMAL(8,2)	Importe total de la línea

2. ESQUEMA SANIDAD

La figura 2 muestra el diseño del tema Sanidad. Este diseño se realiza con la herramienta de análisis y diseño Data Mode-ling de MySQL-bench Work

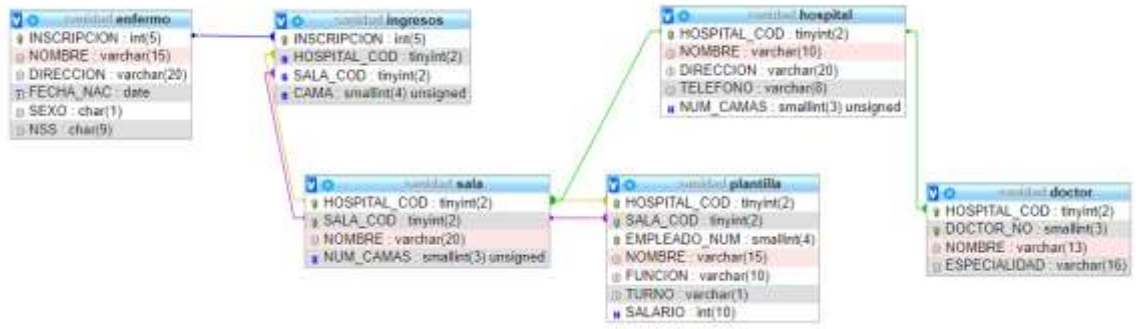

Figura 2 Diseño del Esquema Empresa

Los datos sobre el Esquema "sanidad" gestionarlos a lo largo de este módulo.

Echando un vistazo rápido a este diseño, debemos interpretar lo siguiente:

- Tenemos seis entidades diferentes: hospitales (HOSPITAL), salas de los hospitales (SALA), doctores de los hospitales (DOCTOR), empleados de las salas de los hospitales (PLANTILLA), enfermos (ENFERMO) y enfermos ingresados actualmente (INGRESOS).
- Entre las seis entidades se establecen relaciones:
  - Entre HOSPITAL y SALA (relación fuerte-débil 1:N), ya que las salas se identifican con un código de sala en cada hospital; es decir, podemos tener una sala identificada con el código 1 en el hospital X, y una sala identificada también con el código 1 en el hospital Y.
  - Entre HOSPITAL y DOCTOR (relación fuerte-débil 1:N), ya que los doctores se identifican con un código de doctor en cada hospital; es decir, podemos tener un doctor identificado con el código 10 en el hospital X, y un doctor identificado también con el código 10 en el hospital Y.
  - Entre SALA y PLANTILLA (relación fuerte-débil 1:N), puesto que los empleados se identifican con un código dentro de cada sala; es decir, podemos tener un empleado identificado con el código 55 en la sala 10 del hospital X y un empleado identificado también con el código 55 en otra sala de cualquier hospital.
  - Entre ENFERMO e INGRESOS (relación fuerte-débil 1:1), ya que un enfermo puede estar ingresado o no.
  - Entre SALAe INGRESOS (relación 1:N), ya que en una sala puede haber cero o varios enfermos ingresados, y un enfermo sólo puede estar en una única sala.
- Seguro que no es el mejor diseño para una correcta gestión de hospitales, pero nos interesa mantener este diseño para las posibilidades que nos dará de cara al aprendizaje del lenguaje SQL. Sin embargo, aprovechamos la ocasión para comentar los puntos oscuros en el diseño:
  - Quizás no sea muy normal que los empleados de un hospital se identifiquen dentro de cada sala. Es decir, en el diseño, la entidad PLANTILLA es débil de la entidad SALA y,

quizás, sería más lógico que fuera débil de la entidad HOSPITAL de forma similar a la entidad DOCTOR.

- Para poder gestionar los pacientes (ENFERMO) que actualmente están ingresados, es necesario establecer una relación entre ENFERMO y SALA, la cual sería de orden N:1, ya que en una sala puede haber varios pacientes ingresados y un paciente, si está ingresado, lo está en una sala. La traducción correspondiente al modelo relacional provocaría lo siguiente:

```
SALA (Hospital_Cod, Sala_Cod, Nombre, Num_Camas(VN) donde Sala_Cod
REFERENCIA HOSPITAL
ENFERMO (Inscripción, Apellido, Dirección(VN), Fecha_Nac(VN), Sexo,
Nss(VN), Hosp_Ingreso(VN), Sala_Ingreso(VN)) on {Hosp_Ingreso,
Sala_Ingreso} REFERENCIA SALA
```

Fijémonos en que la relación (tabla) ENFERMO contiene la pareja de atributos (Hosp_Ingreso, Sala_ingreso) que conjuntamente son clave foránea de la relación (tabla) SALA y que pueden tener valores nulos (VN), ya que un paciente no debe estar necesariamente ingresado. Si pensamos un poco en la gestión real de estas tablas, nos encontraremos con que la tabla ENFERMO normalmente contendrá muchas filas y que, por suerte para los pacientes, muchas de estas tendrán vacíos los campos Hosp_Ingreso y Sala_Ingreso, ya que, del total de pacientes que pasan por un hospital, un conjunto muy pequeño está ingresado en un momento determinado. Esto puede provocar una pérdida grave de espacio en la base de datos.

En estas situaciones es lícito pensar en una entidad que aglutine a los pacientes que están ingresados actualmente (INGRESOS), la cual debe ser débil de la entidad que engloba a todos los pacientes (ENFERMO). Ésta es la opción adoptada en este diseño.

También sería adecuado disponer de una entidad que aglutinara las diferentes especialidades médicas existentes, de modo que pudiéramos establecer una relación entre esta entidad y la entidad DOCTOR. No es el caso y, por tanto, la especialidad de cada doctor se introduce como un valor alfanumérico.

Asimismo, de forma similar, sería adecuado disponer de una entidad que aglutinara las diferentes funciones que puede llevar a cabo el personal de la plantilla, de modo que pudiéramos establecer una relación entre esta entidad y la entidad PLANTILLA. Tampoco es el caso y, por tanto, la función que cada empleado lleva a cabo se introduce como un valor alfanumérico.

La traducción correspondiente al modelo relacional es la siguiente:

```
HOSPITAL (Hospital_Cod, Nombre, Dirección(VN), Teléfono(VN),
Num_Camas(VN))
SALA (Hospital_Cod, Sala_Cod, Nombre, Num_Camas(VN)) donde Hospital_Cod
REFERENCIA HOSPITAL Y Sala_Cod REFERNCIA SALA
PLANTILLA (Hospital_Cod, Sala_Cod, Empleado_No, Apellido, Función(VN),
Turno(VN), Salario(VN)) DONDE {Hospital_Cod, Sala_Cod} REFERENCIA SALA
ENFERMO (Inscripción, Apellido, Dirección(VN), Fecha_Nace(VN), Sexo,
Nss(VN))
INGRESOS (Inscripción, Hospital_Cod, Sala_Cod, Cama(VN)) DONDE
Inscripción REFERENCIA ENFERMO, {Hospital_Cod, Sala_Cod} REFERENCIA SALA
DOCTOR (Hospital_Cod, Doctor_No, Apellido, Especialidad) DONDE
Hospital_Cod REFERENCIA HOSPITAL
```

La implementación de este modelo relacional en MySQL ha provocado las siguientes tablas:

Tabla **HOSPITAL**, que contiene la enumeración de los hospitales

Nombre	**Null?**	**Tipo**	**Descripción**
HOSPITAL_COD	NOT NULL	INT(2)	Código del hospital
NOMBRE	NOT NULL	VARCHAR(10)	Nombre del hospital
DIRECCIÓN		VARCHAR(20)	Dirección del hospital
TELEFON		VARCHAR(8)	Teléfono del hospital
NUM_CAMAS		INT(3)	Cantidad de camas del hospital

Tabla **SALA**, que contiene las salas de cada hospital

Nombre	**Null?**	**Tipo**	**Descripción**
HOSPITAL_COD	NOT NULL	INT(2)	Código del hospital
SALA_COD	NOT NULL	INT(2)	Código de la sala en cada hospital
NOMBRE	NOT NULL	VARCHAR(20)	Nombre de la sala
NUM_CAMAS		INT(3)	Cantidad de camas del hospital

Tabla **DOCTOR**, que contiene los doctores de los diferentes hospitales

Nombre	**Null?**	**Tipo**	**Descripción**
HOSPITAL_COD	NOT NULL	INT(2)	Código del hospital
DOCTOR_COD	NOT NULL	INT(3)	Código del Doctor en cada hospital
NOMBRE	NOT NULL	VARCHAR(20)	Doctor
ESPECIALIDAD	NOT NULL	VARCHAR(16)	Cantidad de camas del hospital

Tabla **PLANTILLA**, que contiene a los trabajadores no doctores de las salas de los hospitales

Nombre	**Null?**	**Tipo**	**Descripción**
HOSPITAL_COD	NOT NULL	INT(2)	Código del hospital
SALA_COD	NOT NULL	INT(2)	Código de la sala en cada hospital. La pareja (HOSPITAL_COD, SALA_COD) es clave foránea de

			la tabla SALA
EMPLEADO_NUM	NOT NULL	INT(4)	Código del empleado (independiente de hospital y sala)
NOMBRE	NOT NULL	VARCHAR(15)	Nombre del empleado
FUNCION		VARCHAR(10)	Tarea del empleado
TURNO		VARCHAR(1)	Turno del empleado. Valores posibles: (M)añana - (T)arde - (N)oche
SALARIO		INT(10)	Salario anual del empleado

Tabla **ENFERMO**, que contiene los enfermos

Nombre	**Null?**	**Tipo**	**Descripción**
INSCRIPCIÓN	NOT NULL	INT(5)	Identificación del enfermo
APELLIDO	NOT NULL	VARCHAR(15)	Apellido del enfermo
DIRECCIÓN		VARCHAR(20)	Dirección del enfermo
FECHA_NACIMIENTO		DATE	Fecha de nacimiento del enfermo
SEXO	NOT NULL	VARCHAR(1)	Sexo del enfermo. Valores posibles: (H)ombre - (M)ujer
NSS		CHAR(9)	Número de Seguridad Social del enfermo

Tabla **INGRESOS**, que contiene a los enfermos ingresados en los hospitales

Nombre	**Null?**	**Tipo**	**Descripción**
INSCRIPCIÓN	NOT NULL	INT(5)	Código de enfermo
HOSPITAL_COD	NOT NULL	INT (2)	Código del Hospital
SALA_COD	NOT NULL	INT(2)	Código de sala de hospital
CAMA		INT(4)	Número de cama que ocupa en la sala

Así pues, ya estamos en condiciones de introducirnos en el aprendizaje de las instrucciones de consulta SQL, que practicaremos en las tablas de los temas empresa y sanidad presentados.

Todas las consultas en el lenguaje SQL se realizan con una única sentencia, llamada SELECT, que se puede utilizar con diferentes niveles de complejidad. Y todas las instrucciones SQL **finalizan**, obligatoriamente, con **un punto y coma**.

Tal y como lo indica su nombre, esta sentencia permite **seleccionar** lo que el usuario pide, el cual no debe indicar dónde debe ir a buscarlo ni cómo debe hacerlo.

La instrucción SELECT consta de varias secciones que generalmente se denominan **cláusulas**. Dos de estos apartados son siempre obligatorios y son los primeros en presentar. El resto de cláusulas deben utilizarse según los resultados que se quieran obtener.

La sintaxis de SELECT es compleja, pero en esta Unidad no explicaremos todas sus opciones, si no las más simples. Por ello la unidad se llama "**Consultas se Selección Simple**".

Una forma más general consiste en la siguiente sintaxis:

```
SELECT  expresion_select,...
   FROM referencias_de_tablas
   WHERE condiciones;
```

4.1. CLAUSULAS SELECT Y FROM

Es muy importante que practique sobre un SGBD MySQL todas las instrucciones SQL que se van explicando. Para ello, siga las instrucciones del Anexo e instale, si no lo ha hecho todavía, el software **MySQL**, e importa las bases de datos de ejemplo empresa y sanidad para poder probar los ejemplos que se van mostrando a lo largo del material.

La sintaxis más simple de la sentencia SELECTutiliza estas dos cláusulas de forma obligatoria:

SELECT < expresión / columna >, < expresión / columna >,... **FROM** < tabla >, < tabla >,... ;

La cláusula **SELECT** permite escoger columnas y/o valores (resultados de las expresiones) derivados de éstas.

La cláusula **FROM** permite especificar las tablas en las que se deben buscar las columnas o sobre las que se calcularán los valores resultantes de las expresiones.

Una sentencia SQL puede escribirse en una única línea, pero para hacer la sentencia más legible se suelen utilizar diferentes líneas para las diferentes cláusulas.

Ejemplo de utilización simple de las cláusulas select y from

En el Esquema ***empresa***, se quieren mostrar los códigos, apellidos y oficios de los empleados.

Éste es un ejemplo claro de las consultas más simples: hay que indicar las columnas a visualizar y la tabla de dónde visualizarlas. La sentencia es la siguiente:

```
mysql> SELECT emp_no, nombre, cargo FROM empleado;
```

El resultado que se obtiene es el siguiente:

```
+--------+--------+------------+
| emp_no | nombre | cargo      |
+--------+--------+------------+
|   7369 | S      | EMPLEADO   |
|   7499 | ARROYO | VENDEDOR   |
|   7521 | SALA   | VENDEDOR   |
|   7566 | JIM    | DIRECTOR   |
|   7654 | MART   | VENDEDOR   |
|   7698 | NEGRO  | DIRECTOR   |
|   7782 | CEREZO | DIRECTOR   |
|   7788 | GIL    | ANALISTA   |
|   7839 | REY    | PRESIDENTE |
|   7844 | TOVAR  | VENDEDOR   |
|   7876 | ALONSO | EMPLEADO   |
|   7900 | JIMENO | EMPLEADO   |
|   7902 | FERN   | ANALISTA   |
|   7934 | MU     | EMPLEADO   |
+--------+--------+------------+
14 rows in set (0.00 sec)
```

Ejemplo de utilización de expresiones en la cláusula SELECT

En el Esquema ***empresa***, se quieren mostrar los códigos, apellidos y salario anual de los empleados.

Como saben que en la tabla EMPLEADO consta el salario mensual de cada empleado, sabemos calcular, mediante el producto por el número de pagas mensuales en un año (12, 14, 15…?), su salario anual. Supondremos que el empleado tiene catorce pagas mensuales, ¡como la mayoría de los mortales! Por tanto, en este caso, alguna de las columnas a visualizar es el resultado de una expresión:

```
mysql> SELECT emp_no, nombre, salario*14 FROM empleado;
```

El resultado que se obtiene es el siguiente:

```
+--------+--------+--------------+
| emp_no | nombre | salario * 14 |
+--------+--------+--------------+
|   7369 | S      |      1456000 |
|   7499 | ARROYO |      2912000 |
|   7521 | SALA   |      2275000 |
|   7566 | JIM    |      5414500 |
|   7654 | MART   |      2275000 |
|   7698 | NEGRO  |      5187000 |
|   7782 | CEREZO |      4459000 |
|   7788 | GIL    |      5460000 |
|   7839 | REY    |      9100000 |
|   7844 | TOVAR  |      2730000 |
|   7876 | ALONSO |      2002000 |
|   7900 | JIMENO |      1729000 |
|   7902 | FERN   |      5460000 |
|   7934 | MU     |      2366000 |
```

```
+--------+--------+-------------+
14 rows in set (0.00 sec)
```

Fijémonos en que el lenguaje SQL utiliza los nombres reales de las columnas como títulos en la presentación del resultado y, en caso de columnas que correspondan a expresiones, nos muestra la expresión como título.

El lenguaje SQL permite dar un nombre alternativo (llamado **alias**) a cada columna. Para ello, se puede utilizar la siguiente sintaxis:

SELECT < expresión / columna > [**AS** alias], < expresión / columna > [**AS** alias] ,...

FROM < tabla >, < tabla >,... ;

Tenga en cuenta lo siguiente:

- Si el alias está formado por varias palabras, se debe cerrar entre comillas dobles.
- Hay algunos SGBD que permiten la no utilización de la partícula ***as*** (como Oracle y MySQL) pero en otros es obligatoria (como MS-Access).

Ejemplo de utilización de sobrenombre en la cláusula SELECT

En el tema empresa, se quieren mostrar los códigos, apellidos y salario anual de los empleados.

La instrucción para alcanzar el objetivo puede ser, con la utilización de sobrenombre:

```
mysql> SELECT emp_no, nombre, salario * 14  AS "Salario Anual"  FROM
empleado;
```

Obtendríamos el mismo resultado sin la partícula as:

```
mysql> SELECT emp_no, nombre, salario * 14  "Salario Anual"  FROM
empleado;
```

El resultado que se obtiene en este caso es el siguiente:

```
+--------+--------+---------------+
| emp_no | nombre | Salario Anual |
+--------+--------+---------------+
|   7369 | S      |       1456000 |
|   7499 | ARROYO |       2912000 |
|   7521 | SALA   |       2275000 |
|   7566 | JIM    |       5414500 |
|   7654 | MART   |       2275000 |
|   7698 | NEGRO  |       5187000 |
|   7782 | CEREZO |       4459000 |
|   7788 | GIL    |       5460000 |
|   7839 | REY    |       9100000 |
|   7844 | TOVAR  |       2730000 |
|   7876 | ALONSO |       2002000 |
|   7900 | JIMENO |       1729000 |
|   7902 | FERN   |       5460000 |
|   7934 | MU     |       2366000 |
+--------+--------+---------------+
```

```
14 rows in set (0.00 sec)
```

El lenguaje SQL facilita una manera sencilla de mostrar todas las columnas de las tablas seleccionadas en la cláusula from (y pierde la posibilidad de utilizar un sobrenombre) y consiste en utilizar un asterisco en la cláusula select.

Ejemplo de uso de asterisco (*) en la cláusula SELECT

Se nos pide mostrar, en el tema empresa, toda la información que hay en la tabla que contiene los departamentos.

La instrucción que nos permite alcanzar el objetivo es la siguiente (fijémonos en que la instrucción SELECT con asterisco nos muestra los datos de todas las columnas de la tabla):

```
mysql> SELECT *  FROM departamento;
```

Y obtenemos el resultado esperado:

```
+---------+---------------+-----------+
| DEPT_NO | DNOM          | LOC       |
+---------+---------------+-----------+
|      10 | CONTABILIDAD  | SEVILLA   |
|      20 | INVESTIGACION | MADRID    |
|      30 | VENTAS        | BARCELONA |
|      40 | PRODUCCON     | BILBAO    |
+---------+---------------+-----------+
4 rows in set (0.00 sec)
```

Aunque disponemos del asterisco para visualizar todas las columnas de las tablas de la cláusula from, a veces nos interesará conocer las columnas de una tabla para diseñar una sentencia SELECT adecuada a las necesidades y no utilizar el asterisco para visualizar todas las columnas.

Los SGBD suelen facilitar mecanismos para visualizar un breve descriptor de una tabla. En MySQL (y también en Oracle), disponemos del comando desc (no es sentencia del lenguaje SQL) para ello. Hay que utilizarla acompañando el nombre de la tabla.

Ejemplo de obtención del descriptor de una tabla

Si necesitamos conocer las columnas que forman una tabla determinada (y sus características básicas), podemos obtener el descriptor: **DESC**

```
mysql> desc departamento;
+---------+---------------------+------+-----+---------+-------+
| Field   | Type                | Null | Key | Default | Extra |
+---------+---------------------+------+-----+---------+-------+
| DEPT_NO | tinyint(2) unsigned | NO   | PRI | NULL    |       |
| DNOM    | varchar(14)         | NO   | UNI | NULL    |       |
| LOC     | varchar(14)         | YES  |     | NULL    |       |
+---------+---------------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

El comando **DESC** no es exactamente un comando SQL, sino un comando que facilitan los SGBD para visualizar la estructura o el diccionario de los datos almacenados en una BD concreta. Por tanto, al no tratarse estrictamente de un comando SQL, admite la no

utilización del punto y coma (;) al final de la sentencia. De este modo, los siguientes códigos son equivalentes:

```
SQL> desc departamento;
SQL> desc departamento
```

Supongamos que estamos conectados con el SGBD en la base de datos o el esquema (**schema**, en inglés) por defecto que contiene las tablas correspondientes al tema empresa y que necesitamos acceder a tablas de una base de datos que contiene las tablas correspondientes a esquema **sanidad**. ¿Lo podemos conseguir?

La cláusula **from** puede hacer referencia a tablas de otra base de datos. En esta situación, es necesario anotar la tabla como **<nom_esquema>.<nom_tabla>.**

El acceso a objetos de otros esquemas (bases de datos) para un usuario conectado a un esquema sólo es posible si tiene concedidos los permisos de acceso correspondientes.

Bases de datos gestionadas por una instancia de un SGBD corporativo

Un SGBD es un conjunto de programas encargados de manejar bases de datos.

En los SGBD ofimáticas (MS-Access) podemos poner en marcha el SGBD sin la obligación de abrir (arrancar) ninguna base de datos. En un momento concreto, podemos tener diferentes ejecuciones de SGBD (instancias), cada una de las cuales puede dar servicio a una única base de datos.

Los DBMS corporativos (Oracle, MSSQLServer, MySQL, PostgreSQL ...), los pusieron en marcha, obligados a tener definido el conjunto de bases de datos que prestan el servicio, en conjunto normalmente denominado base de datos de clúster. En una máquina, tenemos diferentes ejecuciones del mismo DBMS (instancias), cada una de las cuales sirve a un conjunto diferente de bases de datos (base de datos de clúster). Por lo tanto, cada instancia de un DBMS para administrar una base de datos de clúster .

En las máquinas en las que existen SGBD corporativos instalados, se suele configurar un servicio de sistema operativo para cada instancia configurada para que sea gestionada por el SGBD. Así, en máquinas con sistema operativo MS-Windows, esta situación se puede constatar rápidamente echando un vistazo a los servicios instalados (Panel de control\Herramientas administrativas\Servicios), en los que podríamos encontrar varios servicios del Oracle, MySQL, SQLServer … identificats per noms que es decideixen en el moment de creació de la instància (cluster database).

Así, por ejemplo, en una empresa en la que es muy usual tener una base de datos para la gestión comercial, una base de datos para el control de la producción, una base de datos para la gestión de personal, una base de datos para la gestión financiera…, en SGBD como MySQL, PostgreSQL y SQLServer podrían ser distintas bases de datos gestionadas por una misma instancia del SGBD.

Ejemplo de acceso a tablas de otros esquemas

Si, estando conectados al esquema (base de datos) empresa, queremos mostrar los hospitales existentes en el esquema sanidad, habrá que hacer lo siguiente:

```
SELECT  * FROM  sanidad.hospital;
```

El resultado que se obtiene es el siguiente:

```
+--------------+------------+----------------------+----------+-----------+
| HOSPITAL_COD | NOMBRE     | DIRECCION            | TELEFONO | NUM_CAMAS |
+--------------+------------+----------------------+----------+-----------+
|           13 | Provincial | O Donell 50          | 964-4264 |        88 |
|           18 | General    | Atocha s/n           | 595-3111 |        63 |
|           22 | La Paz     | Castellana 1000      | 923-5411 |       162 |
|           45 | San Carlos | Ciudad Universitaria | 597-1500 |        92 |
+--------------+------------+----------------------+----------+-----------+
4 rows in set (0.03 sec)
```

El lenguaje SQL efectúa el producto cartesiano de todas las tablas que encuentra en la cláusula from. En este caso, puede haber columnas con el mismo nombre en diferentes tablas y, si es así y seleccionar una, utilizar obligatoriamente la sintaxis **<nom_tabla>. <nom_columna>**, incluso, la sintaxis **<nom_esquema>. <nom_tabla>. <nom_columna>** si se accede a una tabla de otro esquema.

El lenguaje SQL permite definir sobrenombre para una tabla. Para conseguirlo, es necesario escribir el sobrenombre en la cláusula from después del nombre de la tabla y antes de la coma que la separa de la tabla siguiente (si existe) de la cláusula from.

Ejemplo de utilización de sobrenombre para nombres de tablas

Si estamos conectados al esquema **empresa**, para obtener el producto cartesiano de todas las filas de la tabla DEPARTAMENTO con todas las filas de la tabla SALA del esquema **sanidad**, que muestre únicamente las columnas que forman las claves primarias respectivas, podríamos ejecutar la siguiente instrucción:

```
SQL > SELECT d.dept_no, s.hospital_cod, s.sala_cod FROM departamento d,
sanidad.sala s;
```

MySQL (al igual que Oracle) incorpora una tabla ficticia, llamada **DUAL**, para efectuar cálculos independientes de cualquier tabla de la base de datos aprovechando la potencia de la sentencia SELECT.

Así pues, podemos utilizar esta tabla para hacer lo siguiente:

1. **Efectuar cálculos matemáticos**

```
SQL > SELECT 4*3-8/2 AS "resultado" FROM dual;
```

```
RESULTADOS
----------------------
8
1  FILAS seleccionadas
```

2. **Obtener la fecha del sistema, sabiendo que la proporciona la función sysdate()**

```
SQL > SELECT sysdate() FROM dual;
```

```
SYSDATE()
------------------------
09/02/08
1  FILAS seleccionadas
```

La tabla DUAL también puede ser eludida. De este modo, las siguientes sentencias serían equivalentes a las expuestas anteriormente:

```
SQL > SELECT 4*3-8/2 AS "resultado";
SQL > SELECT sysdate();
```

4.2. Cláusula ORDER BY

La sentencia SELECT tiene más cláusulas aparte de las conocidas select y from. Así, posee una cláusula **order by** que permite ordenar el resultado de la consulta.

Formato de consulta con ordenación:

SELECT [**ALL**/**DISTINCT**] ExpresionColumna [,ExpresionColumna....] **FROM** NombreTabla [**WHERE** CondicionSeleccion] [**ORDER BY** {ExpresionColumna\Posicion} [**ASC**|**DESC**] [,{ExpresionColumna\Posicion} [**ASC**|**DESC**]]] ;

Notación: la cláusula **ORDER BY** es **opcional**, por eso aparece toda ella entre corchetes.

Dentro de ella expresión de columna y posición van entre llaves porque hay que elegir una de ellas y ASC y DESC entre corchetes porque son opcionales donde:

- **ASC| DESC**: ASC (ascendente) o DESC (descendente): indica la forma de ordenación para esa expresión. Por omisión es ASC.
- **ExpresionColumna**: conjunto de nombres de columna con literales, operadores y/o funciones. También admite alias.
- **Posicion**: si queremos ordenar por expresiones que se muestran en el **select** la **ExpresionColumna** puede ser sustituida por el número (*Posicion)*, que corresponde al número de orden que ocupa en la lista de expresiones visualizadas en la **select**.

Si existe más de una expresión por la que ordenar estas aparecen separadas por comas y el orden en que se realizan las clasificaciones es de izquierda a derecha, es decir, a igualdad de la expresión más a la izquierda ordena por la siguiente expresión y así sucesivamente.

Ejemplo de ordenación de los datos utilizando la cláusula ORDER BY

Si se quieren obtener todos los datos de la tabla departamentos, ordenados por el nombre de la localidad, podemos ejecutar la siguiente sentencia:

```
mysql> SELECT * FROM DEPARTAMENTO ORDER  BY  loc;
```

Y obtendremos el siguiente resultado:

```
+---------+---------------+-----------+
| DEPT_NO | DNOM          | LOC       |
+---------+---------------+-----------+
|      30 | VENTAS        | BARCELONA |
```

```
|      40 | PRODUCCON      | BILBAO    |
|      20 | INVESTIGACION | MADRID    |
|      10 | CONTABILIDAD   | SEVILLA   |
+---------+---------------+-----------+
4 rows in set (0.01 sec)
```

4.3. CLÁUSULA WHERE

La cláusula **where** se añade detrás de la cláusula **from** de modo que ampliamos la sintaxis de la sentencia SELECT:

SELECT < expresión / columna >, < expresión / columna >,... **FROM** < tabla >, < tabla >,... [**WHERE** < condición_de_búsqueda >] ;

La cláusula **where** permite establecer los criterios de búsqueda sobre las filas generadas por la cláusula from.

La complejidad de la cláusula **where** es prácticamente ilimitada gracias a la abundancia de operadores disponibles para efectuar operaciones.

1. Operadores aritméticos

Son los típicos operadores +, -, *, /utilizables para formar expresiones con constantes, valores de columnas y funciones de valores de columnas.

2. Operadores de fecha

Para obtener la diferencia entre dos fechas:

- Operador -, para restar dos fechas y obtener el número de días que las separan.

3. Operadores de comparación

Disponemos de diferentes operadores para efectuar comparaciones:

- Los típicos operadores =, !=, >, <, >=, <= , para efectuar comparaciones entre datos y obtener un resultado booleano: cierto o falso.
- El operador **[NOT] LIKE**, para comparar una cadena (parte izquierda del operador) con una cadena patrón (parte derecha del operador) que puede contener los siguientes caracteres especiales:
    - **%** para indicar cualquier cadena de cero o más caracteres.
    - **_** para indicar cualquier carácter.

Así:

```
LIKE 'Torres'   compara con la cadena 'Torres'.
LIKE 'Torr%'    compara con cualquier cadena iniciada por 'Torr'.
LIKE '%S%'      compara con cualquier cadena que contenga 'S'.
LIKE '_o%'      compara con cualquier cadena que tenga por segundo
carácter una 'o'.
LIKE '%%__%%'   compara con cualquier cadena de dos caracteres.
```

Un último conjunto de operadores lógicos:

```
[NOT] BETWEEN valor_1 AND valor_2
```

que permite efectuar la comparación entre dos valores.

```
[NOT] IN (lista_valores_separados_por_comas)
```

que permite comparar con una lista de valores.

```
IS [NOT] NULL
```

que permite reconocer si estamos ante un valor null.

```
<comparador genérico> ANY (lista_valores)
```

que permite efectuar una comparación genérica (=, !=, >, <, >=, ⇐) con cualquiera de los valores de la derecha. Los valores de la derecha serán el resultado de ejecución de otra consulta (SELECT), por ejemplo:

```
SQL > SELECT * FROM emp WHERE nombre != ANY (SELECT 'Alonso' FROM dual);
```

```
<comparador genérico> ALL (lista_valores)
```

que permite efectuar una comparación genérica (=, !=, >, <, >=, <=) con todos los valores de la derecha. Los valores de la derecha serán el resultado de ejecución de otra consulta (SELECT), por ejemplo:

```
SELECT * FROM empleado WHERE nombre != ALL (SELECT 'Alonso' FROM
dual) ;
```

Fijémonos en que:

- =ANY es equivalente a IN.
- =ANY es equivalente a NOT IN.
- = ALL siempre es falso si la lista tiene más de un elemento distinto.
- != ANY siempre es cierto si la lista tiene más de un elemento distinto.

Ejemplo de filtrado simple en la cláusula where

En el esquema ***empresa***, se quieren mostrar a los empleados (código y apellido) que tienen un salario mensual igual o superior a 200.000 y también su salario anual (supongamos que en un año hay catorce pagas mensuales).

La instrucción que permite alcanzar el objetivo es:

```
SQL > SELECT emp_no AS  "Codi", nombre AS "Empleado", salario*14 AS
"Salario anual"  FROM empleado WHERE salario >=200000;
```

El resultado obtenido es éste:

```
+------+----------+---------------+
| Codi | Empleado | Salario anual |
```

```
+------+----------+---------------+
| 7499 | ARROYO   |       2912000 |
| 7566 | JIM      |       5414500 |
| 7698 | NEGRO    |       5187000 |
| 7782 | CEREZO   |       4459000 |
| 7788 | GIL      |       5460000 |
| 7839 | REY      |       9100000 |
| 7902 | FERN     |       5460000 |
+------+----------+---------------+
7 rows in set (0.00 sec)
```

Ejemplo de filtrado de fechas utilizando la especificación ANSI para indicar una fecha

En el tema empresa, se quieren mostrar a los empleados (código, apellido y fecha de contratación) contratados a partir del mes de junio de 1981.

La instrucción que permite alcanzar el objetivo es:

```
SELECT emp_no AS "Código", nombre AS "Empleado", fecha_alta AS
"Contrato" FROM empleado WHERE fecha_alta >= '1981-06-01';
```

El resultado obtenido es éste:

```
+---------+----------+------------+
| Código  | Empleado | Contrato   |
+---------+----------+------------+
|    7654 | MART     | 1981-09-29 |
|    7782 | CEREZO   | 1981-06-09 |
|    7788 | GIL      | 1981-11-09 |
|    7839 | REY      | 1981-11-17 |
|    7844 | TOVAR    | 1981-09-08 |
|    7876 | ALONSO   | 1981-09-23 |
|    7900 | JIMENO   | 1981-12-03 |
|    7902 | FERN     | 1981-12-03 |
|    7934 | MU       | 1982-01-23 |
+---------+----------+------------+
9 rows in set (0.00 sec)
```

Ejemplo de utilización de operaciones lógicas en la cláusula where

En el tema empresa, se quieren mostrar a los empleados (código, apellido) resultado de la intersección de los dos últimos ejemplos, es decir, empleados que tienen un sueldo mensual igual o superior a 200.000 y contratados a partir del mes de junio de 1981.

La instrucción para conseguir lo que se nos pide es ésta:

```
SELECT emp_no AS "Código", nombre AS "Empleado" FROM empleado WHERE
fecha_alta >= '1981-06-01' AND salario >= 200000;
```

El resultado obtenido es éste:

```
+---------+----------+
| Código  | Empleado |
+---------+----------+
|    7782 | CEREZO   |
```

```
|    7788 | GIL       |
|    7839 | REY       |
|    7902 | FERN      |
+---------+-----------+
4 rows in set (0.00 sec)
```

Ejemplo 1 de uso del operador LIKE

En el tema empresa, se quieren mostrar a los empleados que tienen como inicial del apellido una 'S'.

La instrucción pedida puede ser ésta:

```
SELECT emp_no AS "Código", nombre AS "Empleado", dept_no AS
"Departamento" FROM empleado WHERE dept_no !=10 AND dept_no != 30;
```

El resultado obtenido es éste:

```
+---------+----------+--------------+
| Código  | Empleado | Departamento |
+---------+----------+--------------+
|    7369 | S        |           20 |
|    7566 | JIM      |           20 |
|    7788 | GIL      |           20 |
|    7876 | ALONSO   |           20 |
|    7902 | FERN     |           20 |
+---------+----------+--------------+
5 rows in set (0.00 sec)
```

Esta instrucción muestra a los empleados con el apellido comenzado por la letra 'S' mayúscula, y se supone que los apellidos están introducidos con la inicial en mayúscula, pero, para asegurar la solución, en el enunciado se puede utilizar la función incorporada upper(), que devuelve una cadena en mayúsculas:

```
SELECT nombre AS "Empleado" FROM empleado WHERE UPPER(nombre)LIKE 'S%';
```

Ejemplo 2 de utilización del operador LIKE

En el tema empresa, se quieren mostrar a los empleados que tienen alguna S en el apellido.

La instrucción pedida puede ser ésta:

```
SELECT nombre AS  "Empleado" FROM empleado WHERE  UPPER (nombre)  LIKE
'%S%' ;
```

Ejemplo 3 de uso del operador LIKE

En el esquema empresa, se quieren mostrar a los empleados que no tienen la R como tercera letra del apellido.

La instrucción pedida puede ser ésta:

```
SELECT nombre AS  "Empleado"  FROM empleado WHERE  UPPER (nombre) NOT
LIKE '__R%';
```

Ejemplo de utilización del operador BETWEEN

En el esquema empresa, se quieren mostrar a los empleados que tienen un salario mensual entre 100.000 y 200.000.

La instrucción pedida puede ser ésta:

```
SELECT emp_no AS "Codi", nombre AS "Empleado", salario AS "Salario"
FROM empleado WHERE salario >=  100000 AND salario <=  200000;
```

Sin embargo, podemos utilizar el operador **BETWEEN** :

```
SELECT emp_no AS "Codi", nombre AS "Empleado", salario AS "Salario"
FROM empleado WHERE salario BETWEEN 100000 AND 200000;
```

Ejemplo de utilización de los operadores IN ó =ANY

En el esquema **empresa**, se quieren mostrar a los empleados de los departamentos 10 y 30.

La instrucción pedida puede ser ésta:

```
SELECT emp_no AS "Codigo", nombre AS "Empleado", dept_no AS
"Departamento" FROM empleado WHERE dept_no = 10 OR dept_no = 30;
```

Sin embargo, podemos utilizar el operador **IN**:

```
SELECT emp_no AS "Codigo", nombre AS "Empleado", dept_no AS
"Departamento" FROM empleado WHERE dept_no IN (10,30);
```

Ejemplo de utilización del operador "NOT IN"

En el tema empresa, se quieren mostrar a los empleados que no trabajan en los departamentos 10 y 30.

La instrucción pedida puede ser ésta:

```
SELECT emp_no AS "Codigo", cognom AS "Empleado", dept_no AS
"Departamento" FROM empleado WHERE dept_no !=10 AND dept_no != 30;
```

4.4. CLÁUSULA LIMIT

Nos va a permitir limitar el número de filas que se visualicen como resultado de una sentencia select.

Formato de consulta con limitación de las filas que se visualizan dentro de las filas seleccionadas con el WHERE:

Formato de consulta con LIMIT:

```
SELECT [ALL/DISTINCT] ExpresionColumna [,ExpresionColumna....] FROM NombreTabla [WHERE CondicionSeleccion] [ORDER BY {ExpresionColumna\Posicion} [ASC|DESC] [,{ExpresionColumna\Posicion} [ASC|DESC] ] ] [ LIMIT [m, ] n ] ;;
```

Nota: la cláusula LIMIT es opcional, por eso aparece toda ella entre corchetes. Dentro de ella también lo es indicar el valor de m

Donde:

- m es el número de fila por el que se comienza la visualización. Las filas se empiezan la numerar por 0. Es opcional y en caso de omitirse se supone el valor 0 (1ª fila)
- n indica el número de filas que se quieren visualizar.