1. Consultas de selección simples

La mejor manera de iniciar el estudio del lenguaje SQL es ejecutar consultas sencillas en la base de datos. Antes, sin embargo, nos 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 almacenados en las bases de datos. Y, entonces, ya nos podremos iniciar en la ejecución de consultas simples.

1.1. 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 relacional y que iba acompañado del lenguaje SEQUEL (acrónimo de *Structured English Query Language*) para manipular y acceder a los datos almacenados en el System R. Posteriormente, la palabra SEQUEL se condensó en SQL (acrónimo de *SQL*).

> **¿Por qué SQL en lugar de SEQUEL?**
>
> Se utiliza el acrónimo *SQL* en lugar de *SEQUEL* porque la palabra *SEQUEL* ya estaba registrado por la compañía inglesa de aviones Hawker-Siddeley.

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

- IBM comercializa diversos productos relacionales con el lenguaje SQL: System / 38 en 1979, SQL / DS en 1981, y DB2 en 1983.
- Relational Software, Inc. (actualmente, *Oracle* Corporation) crea su propia versión de SGBD relacional para la Marina de los EE.UU., la CIA y otros, y el verano de 1979 libera *Oracle* V2 (versión 2) para las computadoras VAX (las grandes competidoras de la época con las computadoras de IBM).

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 implementar una versión común del lenguaje y, así, en 1986 la ANSI (American National Standards Institute) publica el estándar SQL-86, que

en 1987 es ratificado por la ISO (Organización Internacional para la Normalización, o International Organization for Standardization en inglés).

Pronunciación de SQL

El ANSI ha decidido que la pronunciación inglesa del acrónimo *SQL* es / es kju: l /, y la correspondiente catalana sería / esku El /. Hoy en día se encuentran muchos profesionales que, erróneamente, pronuncian *Sequel* .

La tabla 1 .1 presenta las diferentes revisiones del estándar SQL que han aparecido desde 1986. No necesitamos saber qué ha aportado cada revisión, sino que estas revisiones han existido.

Tabla 1.1. Revisiones del estándar SQL

año	revisión	Alias	comentarios
1986	SQL-86	SQL-87 / SQL1	Publicado por el ANSI en 1986, y ratificado por la ISO 1987.
1989	SQL-89		Pequeña revisión.
1992	SQL-92	SQL2	Gran revisión.
1999	SQL: 1999	SQL3	Introduce consultas recursivas, disparadores ...
2003	SQL: 2003		Introduce temas de XML, funciones Windows ...
2006	SQL: 2006		

1.2. Tipo de sentencias SQL

Términos en inglés

En la jerga informática se utilizan los siguientes términos:

- *Fecha definition language* , abreviado DDL
- *Fecha control language* , abreviado DCL
- *Query Language* , abreviado QL
- *Fecha manipulation language* , abreviado DML

Los SGBD relacionales incorporan el lenguaje SQL para ejecutar diferentes 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 agrupar en las siguientes:

1. Sentencias destinadas a la definición de los datos (LDD), que permiten definir los objetos (tablas, campos, valores posibles, reglas de integridad referencial, restricciones ...).

2. Sentencias destinadas al control sobre los datos (LCD), que permiten conceder y retirar permisos sobre los diferentes objetos de la base de datos.
3. Sentencias destinadas a la consulta de los datos (LC), que permiten acceder a los datos en modo consulta.
4. Sentencias destinadas a la manipulación de los datos (LMD), que permiten actualizar la base de datos (altas, bajas y modificaciones).

En algunos SGBD no hay distinción entre LC y LMD, y únicamente se habla de LMD para las consultas y actualizaciones. Del mismo modo, a veces se incluyen las sentencias de control (LCD) junto con las de definición de datos (LDD). No tiene ninguna importancia que se incluyan en un grupo o que sean un grupo propio: es una simple clasificación.

Todos estos lenguajes suelen tener una sintaxis sencilla, similar a las órdenes de consola para un sistema operativo, llamada **sintaxis auto-suficiente** .

SQL alojado

Las sentencias SQL pueden presentar, sin embargo, una segunda sintaxis, sintaxis alojada, consistente en un conjunto de sentencias que son admitidas dentro de un lenguaje de programación llamado *lenguaje anfitrión* .

Así, podemos encontrar LC y LMD que pueden alojarse en lenguajes de tercera generación como C, Cobol, Fortran ..., y en lenguajes de cuarta generación.

Los SGBD suelen incluir un lenguaje de tercera generación que permite alojar sentencias SQL en pequeñas unidades de programación (funciones o procedimientos). Así, el SGBD Oracle incorpora el lenguaje PL / SQL, el SGBD SQLServer incorpora el lenguaje Transact-SQL, el SGBD MySQL 5.x sigue la sintaxis SQL 2003 para la definición de rutinas de la misma manera que el SGBD DB2 de IBM.

1.3. Tipos 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 permitidas. Ciertamente, los diferentes estándares SQL que han ido apareciendo han marcado una cierta línea y los SGBD se acercan, pero tampoco pueden dejar de apoyar a 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 ello tenemos que deducir que, a fin de 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 (nuestra elección ha sido MySQL) teniendo en cuenta que el resto de SGBD también incorpora tipo de datos similares y, en caso de haber de trabajar, siempre tendremos que echar un vistazo a la documentación que cada SGBD facilita.

Cada valor manipulado por un SGBD determinado 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 manera diferente los valores de diferentes tipos de datos.

En el momento de creación de una mesa, hay que especificar un tipo de dato para cada una de las columnas. En la creación de una acción o función almacenada en la base de datos, hay que especificar un tipo de dato para cada argumento. La asignación correcta del tipo de dato es fundamental para que los tipos de datos definen el dominio de valores que cada columna o argumento puede contener. Así, por ejemplo, las columnas de 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:

- Tipos de datos para gestionar información alfanumérica.
- Tipos de datos para gestionar información numérica.
- Tipos 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 descrito. La notación que se utiliza, en cuanto a sintaxis de definición del lenguaje, habitual, consiste en poner entre corchetes ([]) los elementos opcionales, y separar el carácter | los elementos alternativos.

1.3.1. 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

El tipo CHAR [(longitud)]

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 tipo CHARes de 1 carácter, y la longitud máxima permitida es de 255 caracteres.

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

El tipo VARCHAR (longitud)

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 VARCHARalmacenan el valor exacto que indica el usuario sin añadir espacios en blanco. Si se intenta insertar una cadena de longitud más larga, VARCHARdevuelve un error.

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

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

En comparación con el tipo de datos CHAR, VARCHARfunciona tal como se muestra en la tabla 1 .2 .

Tabla 1.2. Comparación entre el tipo CHAR y VARCHAR

valor	CHAR(4)	VARCHAR(4)
'Ab'	'Ab'	'Ab'
'Abcd'	'Abcd'	'Abcd'
'Abcdefgh'	'Abcd'	'Abcd'

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

El tipo BINARY (longitud)

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

El tipo VARBINARY (longitud)

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

Así pues, por ejemplo, una columna definida como VARBINARY (4)a 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** .

El tipo BLOB

El tipo de datos BLOBes un objeto que permite contener una cantidad grande y variable de datos de tipo binario.

De hecho, se puede considerar un dato de tipo BLOBcomo un dato de tipo VARBINARY, pero sin limitación en cuanto al número de bytes. De hecho, los valores de tipo BLOBalmacenan 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

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 manera similar al tipo BLOB, los valores tipo 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 hasta 2 ^ 8 + 2 bytes
- TEXT puede almacenar hasta 2 ^ 16 + 2 bytes
- MEDIUMTEXT puede almacenar hasta 2 ^ 24 + 3 bytes
- LONGTEXT puede almacenar hasta 2 ^ 32 + 4 bytes

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

El tipo ENUMdefine un conjunto de valores de 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 ENUMy se especifica que no admite valores nulos, entonces, el valor por defecto será el primero de la lista de cadenas.

El número máximo de cadenas diferentes que puede soportar el tipo ENUMes 65535.

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

Una columna de 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 diferentes que puede soportar el tipo SET es 64.

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

- ''
- 'One'
- 'Two'
- 'One, two' (el valor 'two, one' no se prevé que el +++++++ los elementos no afecta las listas)

1.3.2. Tipos de datos numéricos

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

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

También soporta:

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

Los tipos de datos INTEGER

El tipo INTEGER(comúnmente abreviado como INT) almacena valores enteros. Hay varios subtipos de enteros en función de los valores admitidos (véase la tabla 1 .3).

Tabla 1.3. Tipo de datos INTEGER

Tipo de entero	Almacenamiento (en bytes)	Valor mínimo (con signo / sin signo)	Valor máximo (con signo / sin signo)
TINYINT	1	-128 / 0	127/255
SMALLINT	2	-32.768 / 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 enteras admiten la especificación del número de dígitos que hay que mostrar de un valor concreto, utilizando la sintaxis:

INT (N), En la que 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 sólo 4 dígitos. Hay que tener en cuenta que esta especificación no condiciona el valor almacenado, sólo fija el valor que hay que 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 más habitual es utilizar el tipo de datos INT.

Tipo FLOAT, REAL y DOUBLE

FLOAT, REALY DOUBLEson los tipos de datos numéricos que almacenan valores numéricos reales (es decir, que admiten decimales).

Los tipos FLOATy REALalmacenan en 4 bytes y los DOUBLEen 8 bytes.

Los tipos FLOAT, REALo DOUBLE PRECISIONadmiten que especifiquen los dígitos de la parte entera (E) y los dígitos de la parte decimal (D, que pueden ser 30 como máximo, y nunca más grandes 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 quiere almacenar el valor **999.00009** , el valor almacenado realmente será **999.0001** , que es el valor más cercano (aproximado) al original.

Tipo de datos DECIMAL y NUMERIC

DECIMALy NUMERICson 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 manera redondeada, es decir, que si hay que almacenar un valor en un espacio que no es adecuado, emitirá un error. Este tipo de datos permiten asegurar que el valor es exactamente lo 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 un 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.

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 debe definirse, siguiendo la siguiente sintaxis:

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

Los valores literales de los bits se dan siguiendo el formato: **b'valor_binari '** . Por ejemplo, un valor binario admisible para un campo de 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.

BITes un sinónimo de TINYINT(1).

Otros tipos numéricos

BOOLo BOOLEANes el tipo de datos que permite almacenar tipos de datos booleanas (que permiten los valores verdadero o falso). BOOLo BOOLEANes sinónimo de TINYINT(1). Almacenar un valor de cero se considera falso. En cambio, un valor distinto de cero se interpreta como cierto.

Modificadores de tipo numéricos

Hay algunas palabras clave que se pueden añadir a la definición de una columna numérica (entera o real) para acondicionar 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_INCREMENTv: Cuando se añade un valor 0 o NULL en aquella columna, el valor que se almacena es el valor más alto incrementado en 1. El primer valor predeterminado es 1.

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 hay que referirse a los datos referentes a un año es importante explicitar los cuatro dígitos. Es decir, que para expresar el año 98 del siglo XX lo mejor es referirse a ella explícitamente así: 1998.

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

- Los valores de los años que van entre 00 a 69 se interpretan como los años: 2.000 a 2.069.
- Los valores de los años que van entre 70-99 interpretan como los años: 1970 a 1999.

El tipo de dato DATE

DATEpermite 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'.

El tipo de dato DATETIME

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

El formato de DATETIMEen 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 tipo DATETIMEvan desde '1000-01-01 12:00:00' hasta '9999-12-31 23:59:59'.

El tipo de dato TIMESTAMP

TIMESTAMPes un tipo de dato similar a DATETIMEy tiene el mismo formato predeterminado: 'AAAA-MM-DD HH: MM: SS'. TIMESTAMPSin embargo, permite almacenar la hora y fecha actuales en un momento determinado.

Si se asigna el valor NULL en una columna TIMESTAMPo no se le asigna ningún explícitamente, el sistema almacena por defecto la fecha y hora actuales. Si se especifica una fecha-hora concretas en la columna, entonces la columna tomará aquel valor, como si se tratara de una columna DATETIME.

El rango de valores que admite TIMESTAMPes de '1970-01-01 12:00:01' a '2038-01-19 03:14:07'.

Una columna TIMESTAMPes 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 mesa hay más de una columna de tipo TIMESTAMP, el funcionamiento de la primera columna TIMESTAMPes la esperable: se almacena la fecha y hora de la operación más reciente, a menos que explícitamente se asigne un valor concreto , y, entonces, prevalece el valor especificado. El resto de columnas TIMESTAMPde una misma mesa no se actualizarán con este valor. En este caso, si no se especifica un valor concreto, el valor almacenado será cero.

Ejemplo de creación de tabla utilizando TIMESTAMP

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

- 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);

Tipo de datos TIME

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

TIMESin embargo, también permite expresar el tiempo transcurrido entre dos momentos (diferencia de tiempo). Por ello, los valores que permite almacenar son de '-838: 59: 59' a '838: 59: 59'. En esta 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', en el que D indica los días . Es posible, también, un formato que admite microsegundos 'HH: MM: SS. Uuuuuu' en que uuuuuu son los microsegundos.

Tipo de datos YEAR

YEARes un dato de tipo BYTEque almacena datos de tipo año. El formato predeterminado 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 fin a 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.

1.3.4. Otros tipos de datos

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

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

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

1.4. consultas simples

Una vez ya conocemos los diferentes tipos de datos que nos podemos encontrar almacenadas 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, de comenzar la gestión de los datos. Evidentemente, para poder gestionar datos, previamente se debe haber definido las tablas que deben contener los datos, y para poder consultar datos hay haberlas introducido antes. El aprendizaje del lenguaje SQL se efectúa, sin embargo, en sentido inverso; es decir, empezaremos conociendo las posibilidades de consulta de datos sobre tablas ya creadas y con datos ya introducidos. Necesitamos, sin embargo, conocer la estructura de las tablas que se deben gestionar y las relaciones existentes entre ellas.

Las tablas que gestionaremos forman parte de dos diseños diferentes, es decir, son tablas de temas disjuntos. Veamos las.

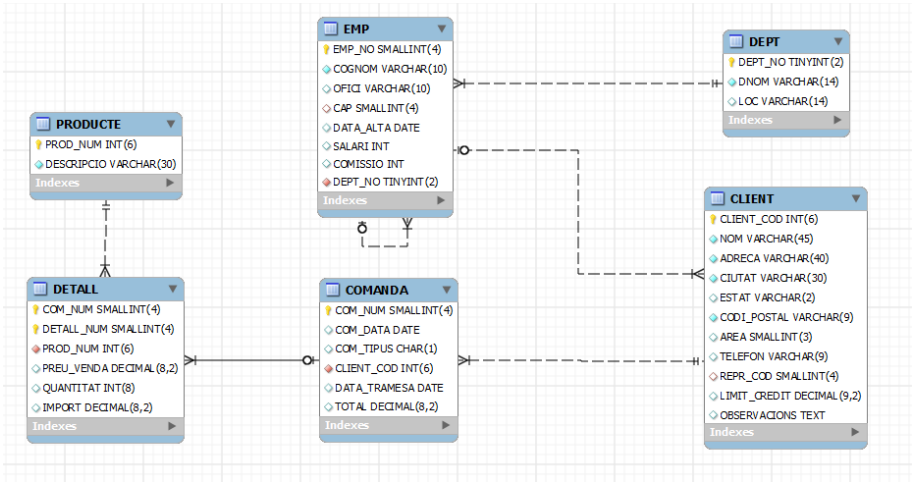
1. Temática empresa

La figura 1 .1 muestra el diseño del esquema Empresa, implementado con la utilidad *Data modeling* del software *MySQL Workbench* , que utiliza una notación fuerza intuitiva:

- Las claves primarias indican con el símbolo de una llave.
- Los atributos que no admiten valores nulos van precedidos del símbolo *.
- Los atributos que admiten valor nulo están marcados con un rombo blanco.
- Las interrelaciones 1: N indican con una línea que termina con tres ramas junto a la entidad interrelacionada en el lado N.
- La opcionalidad se indica con un círculo junto a la entidad opcional y la obligatoriedad con una pequeña línea perpendicular a la interrelación.

> Los datos del diseño EntitatRelació del tema *empresa* , las gestionaremos lo largo de esta unidad.

Figura 1.1. Diseño del esquema Empresa



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

- Tenemos seis entidades diferentes: departamentos (DEP), empleados (EMP), clientes (CLIENTE), productos (PRODUCTO), órdenes (PEDIDO) y detalle de las órdenes (DETALLE).
- Entre las seis entidades establecen relaciones:
  - Entre DEPT y EMP (relación 1: N), ya que un empleado es asignado obligatoriamente a un departamento, y un departamento tiene asignados cero o varios empleados.
  - Entre EMP y EMP (relación reflexiva 1: N), ya que un empleado puede tener por ningún otro empleado de la empresa, y un empleado puede ser jefe de cero o varios empleados.

    - Entre EMP y CLIENTE (relación 1: N), ya 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 varias órdenes a la empresa, y una orden es obligatoriamente de un cliente.
    - Entre PEDIDO y DETALLE (relación fuerte-débil 1: N), ya que el orden está formada por varias líneas, llamadas *detalle del orden* .
    - Entre DETALLE y PRODUCTO (relación N: 1), ya que cada línea de detalle corresponde a un producto.

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

    - Por razones comerciales o de otra índole, en una misma orden hay 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 entregar 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:

DEPT (__Dept_no__ , Dnom , Loc (VN))

EMP (__Emp_No__ , Apellido , Oficio (VN) , Cabeza (VN) , Data_Alta (VN) , Salario (VN) , Comisión (VN)

CLIENTE (__Client_Cod__ , Nombre , Dirección , Ciudad , Estado (VN) , Codi_Postal , Área (VN) , Teléfono (

PRODUCTO (__Prod_Num__ , Descripción)

PEDIDO (__Com_Num__ , Com_Data (VN) , Com_Tipus (VN) , Client_Cod , Data_Tramesa (VN) , Total (VN
DONDE Client_Cod REFERENCIA CLIENTE

DETALLE (__Com_Num , Detall_Num__ , Prod_Num , Preu_Venda (VN) , Cantidad (VN) , Importe (VN)) D

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

>> Tabla DEPT, 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 EMP, que contiene los empleados de la empresa

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
EMP_NO NOT NULL INT (4) Número de empleado de la empresa
APELLIDO NOT NULL VARCHAR (10) Apellido del empleado
OFICIO VARCHAR (10) Oficio del empleado
CAP INT (4) Número del empleado que es el jefe directo (tabla EMP)
DATA_ALTA DATE Fecha Alta
SALARIO INT (10) Salario mensual
COMISION INT (10) Importe de las comisiones
DEPT_NO NOT NULL INT (2) Departamento al que pertenece (tabla DEPT)

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

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
CLIENT_COD 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
CODI_POSTAL 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 EMP)
LIMIT_CREDIT DECIMAL (9,2) Límite de crédito de que dispone el cliente
OBSERVACIONES TEXTO Observaciones

>> Tabla PRODUCTO, que contiene los productos a vender

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
PROD_NUM NOT NULL INT (6) Código de producto
DESCRIPCION NOT NULL VARCHAR (30) Descripción del producto

>> Tabla PEDIDO, que contiene las órdenes de venta

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
COM_NUM NOT NULL INT (4) Número de orden de venta
COM_DATA DATE Fecha de la orden de venta
COM_TIPUS VARCHAR (1) Tipo de orden - Valores válidos: A, B, C
CLIENT_COD NOT NULL INT (6) Código del cliente que efectúa la orden (tabla CLIENTE)
DATA_TRAMESA DATE Fecha de envío del orden
TOTAL DECIMAL (8,2) Importe total del orden

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

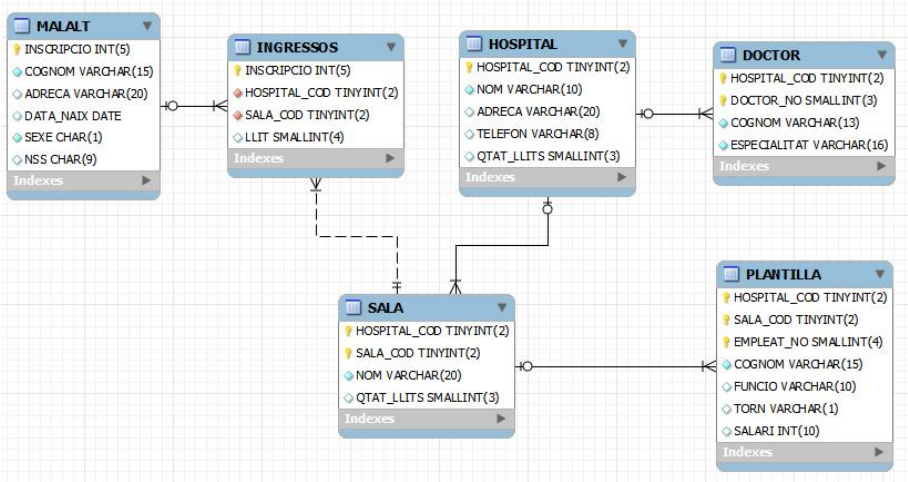
Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
COM_NUM NOT NULL INT (4) Número de orden (tabla PEDIDO)
DETALL_NUM NOT NULL INT (4) Número de línea para cada orden
PROD_NUM NOT NULL INT (6) Código del producto de la línea (tabla PRODUCTO)
PREU_VENDA DECIMAL (8,2) Precio de venta del producto
CANTIDAD INT (8) Cantidad de producto a vender
IMPORTE DECIMAL (8,2) Importe total de la línea

---

2. Temática sanidad

La figura 1 .2 muestra el diseño del tema Sanidad. Este diseño está realizado con la herramienta de análisis y diseño *Fecha Modeling* de *MySQL Workbench*

Figura 1.2. Diseño del esquema Sanidad



Los datos del diseño EntitatRelació del tema *sanidad* las gestionaremos lo largo de este módulo.

Haciendo 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 establecen relaciones:
    - Entre HOSPITAL y SALA (relación fuerte-débil 1: N), ya que las salas se identifican con un código de sala dentro de 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 identifican con un código de doctor dentro de 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), ya 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 y INGRESOS (relación fuerte-débil 1: 1), ya que un enfermo puede estar ingresado o no.
    - Entre Sala INGRESOS (relación 1: N), ya que en una sala puede haber cero o varios enfermos ingresados, y un enfermo ingresado sólo lo puede estar en una única sala.
- Seguro que no es el mejor diseño para una gestión correcta de hospitales, pero nos interesa mantener este diseño para las posibilidades que nos dará de cara al

aprendizaje del lenguaje SQL. Aprovechamos, sin embargo, la ocasión para comentar los puntos oscuros en el diseño:

- Quizás no es muy normal que los empleados de un hospital identifiquen dentro de cada sala. Es decir, en el diseño, la entidad PLANTILLA es débil de la entidad SALA y, tal vez, sería más lógico que fuera débil de la entidad HOSPITAL de manera similar a la entidad DOCTOR.
- Para poder gestionar los pacientes (ENFERMO) que actualmente están ingresados, es necesario establecer una relación entre ENFERMO y SALA, lo 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 el siguiente:

SALA (__Hospital_Cod, Sala_Cod__, Nombre, Qtat_Llits (VN)) donde Sala_Cod REFERENCIA HOSPITAL

ENFERMO (__Inscripció__, Apellido, Dirección (VN), Data_Naix (VN), Sexo, NSS (VN), Hosp_Ingrés (VN), Sala

---

Fijémonos en que la relación (tabla) ENFERMO contiene la pareja de atributos (Hosp_Ingrés, Sala_ingrés) 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 que la mesa ENFERMO normalmente contendrá muchas filas y que, por suerte para los pacientes, muchas de estas tendrán vacíos los campos Hosp_Ingrés y Sala_Ingrés, ya que, del total de pacientes que pasan por un hospital, un conjunto muy pequeño está ingresado en un momento determinado. Esto puede llegar a provocar una pérdida grave de espacio en la base de datos.

En estas situaciones es lícito pensar en una entidad que aglutine los pacientes que están ingresados actualmente (INGRESOS), la cual debe ser débil de la entidad que engloba todos los pacientes (ENFERMO). Esta 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 manera 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 realiza introduce como un valor alfanumérico.

La traducción correspondiente al modelo relacional es la siguiente:

HOSPITAL (__Hospital_Cod__, Nombre, Dirección (VN), Teléfono (VN), Qtat_Llits (VN))

SALA (__Hospital_Cod, Sala_Cod__, Nombre, Qtat_Llits (VN)) donde Hospital_Cod REFERENCIA HOSPITAL

PLANTILLA (__Hospital_Cod, Sala_Cod, Empleat_No__, Apellido, Función (VN), Turno (VN), Salario (VN)) ON

ENFERMO (__Inscripció__, Apellido, Dirección (VN), Data_Naix (VN), Sexo, NSS (VN))

INGRESOS (__Inscripció__, Hospital_Cod, Sala_Cod, Cama (VN)) DONDE Inscripción REFERENCIA ENFERMC

DOCTOR (__Hospital_Cod, Doctor_No__, Apellido, Especialidad) DONDE Hospital_Cod REFERENCIA HOSPIT/

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
TELEFONO VARCHAR (8) Teléfono del hospital
QTAT_LLITS 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 (tabla HOSPITAL)
SALA_COD NOT NULL INT (2) Código de la sala en cada hospital
NOMBRE NOT NULL VARCHAR (20) Nombre de la sala
QTAT_LLITS INT (3) Cantidad de camas de la sala

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

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
HOSPITAL_COD NOT NULL INT (2) Código del hospital (tabla HOSPITAL)
DOCTOR_NO NOT NULL INT (3) Código de doctor dentro de cada hospital
APELLIDO NOT NULL VARCHAR (13) Apellido del doctor
ESPECIALIDAD NOT NULL VARCHAR (16) Especialidad del doctor

>> Tabla PLANTILLA, que contiene 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
EMPLEAT_NO NOT NULL INT (4) Código del empleado (independiente de hospital y sala)
APELLIDO NOT NULL VARCHAR (15) Apellido del empleado
FUNCION VARCHAR (10) Tarea del empleado
TURNO VARCHAR (1) Turno del empleado
Valores posibles: (M) añana - (T) arda - (N) it
SALARIO INT (10) Salario anual del empleado

>> Tabla ENFERMO, que contiene los enfermos

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------
INSCRIPCION NOT NULL INT (5) Identificación del enfermo
APELLIDO NOT NULL VARCHAR (15) Apellido del enfermo
Dirección VARCHAR (20) Dirección del enfermo
DATA_NAIX DATE Fecha de nacimiento del enfermo
SEXO NOT NULL VARCHAR (1) Sexo del enfermo
Valores posibles: (H) ome - (D) onda
NSS CHAR (9) Número de Seguridad Social del enfermo

>> Tabla INGRESOS, que contiene los enfermos ingresados en los hospitales

Nombre Null? tipo Descripción
---------- -------- ---------- ---------- ----------

```
INSCRIPCION NOT NULL INT (5) Código de enfermo
HOSPITAL_COD NOT NULL INT (2) Código de hospital
SALA_COD NOT NULL INT (2) Código de sala de hospital
CAMA INT (4) Número de cama que ocupa dentro de 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.

> Para poder practicar el lenguaje SQL en un SGBD MySQL, véase el Anexo para instalar este software e incorporar las bases de datos de ejemplo descritas (empresa y sanidad).

> Todas las consultas en el lenguaje SQL se hacen 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 como lo indica su nombre, esta sentencia permite **seleccionar** lo que el usuario pide, el cual no debe indicar dónde lo tiene que ir a buscar ni cómo lo hará.

La sentencia SELECT consta de diferentes apartados que se suelen denominar **cláusulas** . Dos de estos apartados son siempre obligatorios y son los primeros que presentaremos. El resto de cláusulas deben utilizarse según los resultados que se quieran obtener.

1.4.1. Cláusulas SELECT y FROM

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

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

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

La cláusula SELECTpermite elegir columnas y / o valores (resultados de las expresiones) derivados de estas.

La cláusula FROMpermite especificar las tablas en las que hay que ir a buscar las columnas o sobre las que se calcularán los valores resultantes de las expresiones.

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

Ejemplo de utilización simple de las cláusulas select y from

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

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

```
SELECT emp_no , apellido , oficio FROM emp;
```

El resultado que se obtiene es el siguiente:

```
EMP_NO APELLIDO OFICIO
------ --------- ----------
7369 SÁNCHEZ EMPLEADO
7499 ARROYO VENDEDOR
7521 SALA VENDEDOR
7566 JIMÉNEZ DIRECTOR
7654 MARTÍN VENDEDOR
7698 NEGRO DIRECTOR
7782 CEREZO DIRECTOR
7788 GIL ANALISTA
7839 REY PRESIDENTE
7844 TOVAR VENDEDOR
7876 ALONSO EMPLEADO
7900 JIMENO EMPLEADO
7902 FERNÁNDEZ ANALISTA
7934 MUÑOZ EMPLEADO

14 rows selected
```

Ejemplo de utilización de expresiones en la cláusula select

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

Como saben que en la tabla EMP 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:

```
SELECT emp_no , apellido , salario * 14  FROM emp;
```

El resultado que se obtiene es el siguiente:

```
EMP_NO APELLIDO SALARIO * 14
------------------- --------- ---------------- ----
7369 SÁNCHEZ 1456000
7499 ARROYO 2912000
7521 SALA 2275000
7566 JIMÉNEZ 5414500
7654 MARTÍN 2275000
7698 NEGRO 5187000
7782 CEREZO 4459000
7788 GIL 5460000
7839 REY 9100000
7844 TOVAR 2730000
7876 ALONSO 2002000
7900 JIMENO 1729000
7902 FERNÁNDEZ 5460000
7934 MUÑOZ 2366000

14 rows selected
```

Fijémonos 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 emplear la siguiente sintaxis:

```
SELECT  < expresión / columna >  [ AS alias ] ,  < expresión / columna >  [ AS alias ] , ...
FROM  < tabla >,  < tabla >, ... ;
```

Tenga en cuenta lo siguiente:

- Si el alias es formado por varias palabras, hay que cerrarlo 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 el MS-Access).

Ejemplo de utilización de alias 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 alias:

```
SELECT emp_no , apellido , salario * 14  AS  "Salario Anual"  FROM emp;
```

Obtendríamos el mismo resultado sin la partícula as:

```
SELECT emp_no , apellido , salario * 14  "Salario Anual"  FROM emp;
```

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

```
EMP_NO APELLIDO Salario Anual
------------------- --------- ---------------- ----
7369 SÁNCHEZ 1456000
7499 ARROYO 2912000
7521 SALA 2275000
7566 JIMÉNEZ 5414500
7654 MARTÍN 2275000
7698 NEGRO 5187000
7782 CEREZO 4459000
```

```
7788 GIL 5460000
7839 REY 9100000
7844 TOVAR 2730000
7876 ALONSO 2002000
7900 JIMENO 1729000
7902 FERNÁNDEZ 5460000
7934 MUÑOZ 2366000

14 rows selected
```

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 emplear un alias) y consiste en utilizar un asterisco en la cláusula select.

Ejemplo de utilización de asterisco en la cláusula select

Se nos pide que 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 que la instrucción SELECT con asterisco nos muestra los datos de todas las columnas de la tabla):

```
SELECT * FROM dept;
```

Y obtenemos el resultado esperado:

```
DEPT_NO DNOM LOC
--------- -------------- -------------
10 CONTABILIDAD SEVILLA
20 INVESTIGACIÓN MADRID
30 VENTAS BARCELONA
40 PRODUCCIÓN BILBAO
```

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 **descriptor** breve de una mesa. En MySQL (y también en Oracle), disponemos del orden **desc** (no es sentencia del lenguaje SQL) para ello. Hay que emplearla acompañando el nombre de la tabla.

Ejemplo de obtención del descriptor de una mesa

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

```
SQL > DESC dept;

Nombre                                    NULL ? tipo
----------------------------------------- -------- - ----------
DEPT_NO                                   NOT NULL INT ( 2 )
DNOM                                      NOT NULL VARCHAR ( 14 )
LOC                                          VARCHAR ( 14 )
```

El orden DESC no es exactamente una orden SQL, sino una orden que facilitan los SGBD para visualizar la estructura o el diccionario de los datos almacenados en una BD concreta. Por lo tanto, como no se trata estrictamente de una orden SQL, admite la no utilización del punto y coma (;) al final de la sentencia. De este modo, los códigos siguientes son equivalentes:

- SQL> desc dept;
- SQL> desc dept

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 al esquema *sanidad* . Lo podemos conseguir?

> La cláusula from puede hacer referencia a tablas de otra base de datos. En esta situación, hay que anotar la tabla como <nom_esquema>. <nom_taula>.

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 gestionados por una instancia de un SGBD corporativo

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

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

Los SGBD corporativos (Oracle, MsSQLServer, MySQL, PostgreSQL ...), al ponerlos en marcha, obligan a tener definido el conjunto de bases de datos al que dan servicio, conjunto que se suele llamar *cluster database* . En una misma máquina, podemos tener diferentes ejecuciones de un mismo SGBD (instancias), cada una de las cuales da servicio a un conjunto diferente de bases de datos (*cluster database*). Así pues, cada instancia de un SGBD permite gestionar un *cluster database* .

En las máquinas en las que hay 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 el que podríamos encontrar diversos servicios de Oracle, MySQL, SQLServer ... identificados por nombres que se deciden en el momento de creación de la instancia (*cluster database*).

Así pues, 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 SQL Server podrían ser diferentes 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 sanitat.hospital;
```

El resultado que se obtiene es el siguiente:

```
HOSPITAL_COD NOMBRE DIRECCIÓN TELÉFONO QTAT_LLITS
------------ ---------- ------------------ ------- - ----------
13 Provincial O Donella 50.964 a 4.264 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 selected
```

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 hay que seleccionar una, hay que utilizar obligatoriamente la sintaxis <nom_taula>. <nom_columna>y, incluso, la sintaxis <nom_esquema>. <nom_taula>. <nom_columna>si se accede a una tabla de otro esquema.

Ejemplo de sentencia " SELECT " con varias mesas y coincidencia en nombres de columnas

Si desde el esquema *empresa* queremos mostrar el producto cartesiano de todas las filas de la tabla DEPTcon todas las filas de la tabla SALAdel esquema *sanidad* (visualización que no tiene ningún sentido, pero que hacemos a modo de ejemplo), mostrando únicamente las columnas que forman las claves primarias respectivas, ejecutaríamos lo siguiente:

```
SELECT dept . dept_no , sanidad . sala . hospital_cod ,
  sanidad . sala . sala_cod
FROM dept , sanidad . sala;
```

El resultado obtenido es formado por cuarenta filas. En mostramos sólo algunas:

```
DEPT_NO HOSPITAL_COD SALA_COD
------------------- ------------------- ------ ----------
10 13 3
10 13 6
10 18 3
...
40 45 1
40 45 2
40 45 4
```

```
40 rows selected
```

En este caso, la sentencia se hubiera podido escribir sin utilizar el prefijo sanidad en las columnas de la tabla SALA en la cláusula select, ya que en la cláusula from no aparece más de una tabla llamada SALA y, por tanto, no hay problemas de ambigüedad. Pudimos escribir lo siguiente:

```
SELECT dept . dept_no , sala . hospital_cod , sala . sala_cod
FROM dept , sanidad . sala;
```

El lenguaje SQL permite definir **alias** para una tabla. Para conseguirlo, hay que escribir el alias 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 alias para nombres de tablas

Si estamos conectados al esquema *empresa* , para obtener el producto cartesiano de todas las filas de la tabla DEPT 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 instrucción siguiente:

```
SELECT de . dept_no , s . hospital_cod , s . sala_cod
FROM dept de , sanidad . sala s;
```

El lenguaje SQL permite utilizar el resultado de una sentencia SELECT como tabla en la cláusula from de otra sentencia SELECT.

Ejemplo de sentencia " SELECT " como tabla en una cláusula from

Así, pues, otra manera de obtener, estando conectados al esquema *empresa* , el producto cartesiano de todas las filas de la tabla DEPT con todas las filas de la tabla SALA del esquema *sanidad* , que muestre únicamente las columnas que forman las claves primarias respectivas, sería la siguiente:

```
SELECT de . dept_no , h . hospital_cod , h . sala_cod
FROM dept de , ( SELECT hospital_cod , sala_cod FROM sanidad . Sala ) h;
```

MySQL (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 usar esta tabla para hacer lo siguiente:

1. Realizar cálculos matemáticos

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

RESULTADO
---------------------
8
1 rows selected
```

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

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

SYSDATE ( )
-------------------------
09 / 02 / 08
1 rows selected
```

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

```
SELECT 4 * 3 - 8 / 2 AS "Resultado" ;

SELECT SYSDATE ( ) ;
```

1.4.2. Cláusula ORDER BY

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

Ejemplo de ordenación de los datos utilizando la cláusula ORDER BY

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

```
SELECT * FROM DEPT ORDER BY loc;
```

Y obtendremos el siguiente resultado:

```
DEPT_NO DNOM LOC
---------- -------------- ---------
30 VENTAS BARCELONA
40 PRODUCCIÓN BILBAO
20 INVESTIGACIÓN MADRID
10 CONTABILIDAD SEVILLA
```

1.4.3. cláusula Where

La cláusula where añade detrás de la cláusula from lo que ampliamos la sintaxis de la sentencia SELECT:

```
SELECT  < expresión / columna >,  < expresión / columna >, ...
FROM  < tabla >,  < tabla >, ...
[ WHERE  < condició_de_cerca > ] ;
```

> 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 fechas

Con el fin de 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: verdadero 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 caracteres especiales siguientes:
  - % 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 '_U%' 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 (llista_valors_separats_per_comes)
```

que permite comparar con una lista de valores.

```
IS [NOT] NULL
```

que permite reconocer si nos encontramos ante un valor null.

```
<Comparador genérico> AÑO (llista_valors)
```

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

```
SELECT * FROM emp WHERE apellido! = Año ( SELECT 'Alonso' FROM dual ) ;
```

```
<Comparador genérico> ALL (llista_valors)
```

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 emp WHERE apellido! = ALL ( SELECT 'Alonso' FROM dual ) ;
```

Fijémonos que:

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

Ejemplo de filtrado simple en la cláusula where

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

La instrucción que permite alcanzar el objetivo es ésta:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , salario * 14 AS "Salario anual" FROM emp W
```

El resultado obtenido es el siguiente:

```
Código Empleado Salario anual
------------------- -------- ---------------- ----
```

```
7499 ARROYO 2912000
7566 JIMENEZ 5414500
7698 NEGRO 5187000
7782 CEREZO 4459000
7788 GIL 5460000
7839 REY 9100000
7902 FERNANDEZ 5460000

7 rows selected
```

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

En el tema *empresa* , se quieren mostrar 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 ésta:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , data_alta AS "Contrato" FROM emp WHERE
```

El resultado obtenido es el siguiente:

```
Código Empleado Contrato
------------------- --------- ---------------- ------
7654            MARTIN     29 / 09 / 81
7782            CEREZO 09 / 06 / 81
7788            GIL 09 / 11 / 81
7839            REY       17 / 11 / 81
7844            TOVAR 08 / 09 / 81
7876            ALONSO     23 / 09 / 81
7900            JIMENO 03 / 12 / 81
7902            FERNANDEZ 03 / 12 / 81
7934            MUÑOZ      23 / 01 / 82

9  rows selected
```

Ejemplo de utilización de operaciones lógicas en la cláusula where

En el tema *empresa* , se quieren mostrar 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" , apellido AS "Empleado" FROM emp WHERE data_alta > = '1981-06-01
```

El resultado obtenido es el siguiente:

```
código Empleado
------------------- ---------
7782 CEREZO
7788 GIL
7839 REY
7902 FERNANDEZ

4 rows selected
```

Ejemplo 1 de utilización del operador like

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

La instrucción solicitada puede ser esta:

```
SELECT apellido AS "Empleado" FROM emp WHERE apellido LIKE 'S%' ;
```

Esta instrucción muestra 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, a fin de asegurar la solución, en el enunciado se puede utilizar la función incorporada upper (), que devuelve una cadena en mayúsculas:

```
SELECT apellido AS "Empleado" FROM emp WHERE UPPER ( apellido ) LIKE 'S%' ;
```

Ejemplo 2 de utilización del operador like

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

La instrucción solicitada puede ser esta:

```
SELECT apellido AS "Empleado" FROM emp WHERE UPPER ( apellido ) LIKE '% S%' ;
```

Ejemplo 3 de utilización del operador like

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

La instrucción solicitada puede ser esta:

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

Ejemplo de utilización del operador between

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

La instrucción solicitada puede ser esta:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , salario AS "Salario" FROM emp WHERE sala
```

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

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , salario AS "Salario" FROM emp WHERE sala
```

Ejemplo de utilización de los operadores in o = año

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

La instrucción solicitada puede ser esta:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , dept_no AS "Departamento" FROM emp WH
```

Sin embargo, podemos utilizar el operador in:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , dept_no AS "Departamento" FROM emp WH
```

Ejemplo de utilización del operador '' not in ''

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

La instrucción solicitada puede ser esta:

```
SELECT emp_no AS "Código" , apellido AS "Empleado" , dept_no AS "Departamento" FROM emp WH
```