Tema 1

Almacenamiento de la información

Tabla de contenidos

1 Introducción

La informática se define como *"La disciplina que se ocupa del proceso automático de información"*. Por lo tanto la información es la razón de ser de la tecnología informática y del ordenador, que en el fondo es sólo una máquina que procesa información.
Con la información se realizan dos tareas principales:

- Procesamiento. Consiste en combinar información para obtener información resumida o derivada.
- Almacenamiento. Consiste en guardar la información para poder utilizarla posteriormente.

En este tema se va a tratar la segunda parte, el almacenamiento de la información, especialmente a largo plazo.

2 Almacenamiento de la información

La información se almacena en dispositivos denominados *memorias*. Una memoria es simplemente un dispositivo que almacena información y que permite recuperarla cuando se le solicita.

Existen muchas tecnologías de construcción de las memorias, con características distintas, ventajas e inconvenientes, lo que hace que se convierta en una labor fundamental de diseño de un equipo de proceso de datos el elegir la memoria o memorias adecuadas para cada parte del equipo.

Las características comunes de todos los dispositivos de almacenamiento son:

- Unidad de almacenamiento. Es la cantidad de información más pequeña que se puede leer o escribir en el dispositivo. Puede oscilar desde 1 byte hasta miles de bytes, dependiendo del dispositivo y la tecnología utilizada. Si la memoria fuera un archivador con cajones donde se almacena la información, la unidad de almacenamiento sería la capacidad de un cajón.
- Tipo de acceso. Indica la forma en que se puede acceder a una unidad de almacenamiento determinada dentro del dispositivo. Existen dos tipos de acceso:
    - Aleatorio. Se puede acceder directamente a una unidad de almacenamiento si sabemos cual es (su dirección).
    - Secuencial. Las unidades de almacenamiento sólo se pueden acceder de una en una (en secuencia). Para acceder a una unidad determinada hay que ir examinándolas una a una hasta encontrar la deseada.

Independientemente de la tecnología, la memoria de un equipo se divide en dos niveles de almacenamiento: primario y secundario.

2.1 Almacenamiento primario

El almacenamiento primario consta de los dispositivos de memoria que son directamente accesibles al procesador del ordenador. Está formado por la memoria RAM, memorias ROM o FLASH y la memoria caché (que son memorias RAM más pequeñas y rápidas que la RAM común). **Toda la información que deba ser accedida por el procesador debe estar almacenada en el almacenamiento primario.** Por lo tanto, si un dato que debe acceder el procesador no

está en el almacenamiento primario hay que transferirlo o copiarlo desde el almacenamiento en el que esté situado hacia el almacenamiento principal (usualmente a la RAM).

Debido a su relativamente pequeño tamaño, la unidad de almacenamiento en la memoria primaria es de una palabra de entre 8 bits (1 byte) y 64 bits (8 bytes), según la arquitectura del procesador.

Los dispositivos de almacenamiento primario siempre utilizan el tipo de acceso aleatorio.

2.2 Almacenamiento secundario

Son el resto de dispositivos de almacenamiento que no son directamente accesibles por el procesador. Usualmente su capacidad de almacenamiento es varios órdenes de magnitud mayor que la del almacenamiento primario y su velocidad de acceso varios órdenes de magnitud menor que la de éste. Los dispositivos de almacenamiento secundario suelen ser:

- Discos magnéticos (discos duros)
- Discos ópticos (CD / DVD / Bluray)
- Cintas magnéticas
- Discos de estado sólido (SSD), que utilizan tecnología de memoria FLASH.

En el caso del almacenamiento secundario y debido a su gran capacidad, la unidad mínima de almacenamiento no es una palabra ya que se necesitarían direcciones muy grandes. En su lugar se utilizan una unidad llamada *bloque* o *página* (para almacenamiento FLASH). Un bloque es un conjunto de bytes que puede oscilar entre los 128 bytes y los 4Kbytes.

El tipo de acceso puede ser aleatorio (discos en general) o secuencial (cintas).

3 Sistemas de ficheros

A continuación se describirá de forma somera los conceptos relacionados con los ficheros. No hay que confundir los ficheros que se van a estudiar en este tema con los archivos desde el punto de vista del sistema operativo del ordenador. El nombre es similar pero el concepto es distinto.

3.1 Registros

La información se almacena en forma de *registros*.

Un registro es una colección de valores o datos que están relacionados entre si. Cada uno de los valores o datos se llama *campo*. Por ejemplo, un registro `persona` contendría campos como `nombre`, `apellidos`, `dni`, `fecha de nacimiento`, `teléfono, edad`, etc.

Cada campo tiene tres propiedades fundamentales:

- **Nombre**. Sirve para identificar de forma inequívoca al campo dentro de un mismo registro, por lo que no puede estar repetido dentro de un registro.

- **Tipo**. Determina el conjunto de los posibles valores que puede tener el campo (número entero, número con decimales, texto, fecha, etc.).
- **Tamaño**. Indica el espacio de almacenamiento que ocupa el valor o dato. Puede ser fijo (el campo siempre ocupa el mismo espacio sea cual sea el valor que contenga el campo) o variable (el campo puede ocupar más o menos espacio según el valor que tenga, aunque se suele poner un límite máximo a la longitud del contenido).

La lista con los campos que tiene un registro, indicando las propiedades de cada uno, determina el formato o la *estructura* del registro.

Ejemplo: En nuestro ejemplo (persona) podríamos definir el registro como sigue:

- **Registro**: persona
  - **Campo**
    - **Nombre**: nombre
    - **Tipo**: Texto
    - **Tamaño**: Fijo 30 caracteres.
  - **Campo**
    - **Nombre**: apellidos
    - **Tipo**: Texto
    - **Tamaño**: Fijo 50 caracteres.
  - **Campo**
    - **Nombre**: dni
    - **Tipo**: Texto
    - **Tamaño**: Fijo 9 caracteres.
  - **Campo**
    - **Nombre**: fecha-de-nacimiento
    - **Tipo**: Fecha
    - **Tamaño**: Fijo 8 caracteres
  - **Campo**
    - **Nombre**: telefono
    - **Tipo**: Texto
    - **Tamaño**: Fijo 9 caracteres
  - **Campo**
    - **Nombre**: edad
    - **Tipo**: Entero
    - **Tamaño**: Fijo 2 cifras

3.2 Ficheros

Un *fichero informático* es un conjunto de registros, almacenado en algún sistema de almacenamiento, usualmente secundario.

Cada registro se almacena en uno o más bloques del dispositivo de almacenamiento. De cómo se realice esta asignación dependerá el rendimiento del sistema, como se analizará más adelante.

Los ficheros son la unidad básica de información que utilizan los programas. Dicho de otra forma, todos los datos acaban siendo almacenados en ficheros.

Las operaciones básicas que se pueden realizar sobre un fichero son:

- **Consulta**. Consiste en acceder a la información que contiene un registro.
- **Inserción**. Consiste en añadir un nuevo registro al fichero.
- **Modificación**. Consiste en modificar el valor de uno o más campos de un registro ya existente en el fichero.
- **Borrado**. Consiste en eliminar un registro del fichero.

El resto de operaciones sobre los ficheros se pueden realizar siempre utilizando estas operaciones básicas.

Los ficheros se clasifican en ficheros de registros de longitud fija o variable.

3.2.1 Ficheros de registros de longitud fija

En este tipo de ficheros todos los registros son del mismo tipo y todos los campos tienen longitud fija. Por lo tanto ocupan el mismo espacio de almacenamiento.

Su principal ventaja es que son más fáciles de manipular puesto que para saber en qué posición del disco está un fichero sólo hay que multiplicar el número del registro por el tamaño.

Como desventaja principal tienen que son poco flexibles y pueden llegar a desperdiciar espacio.

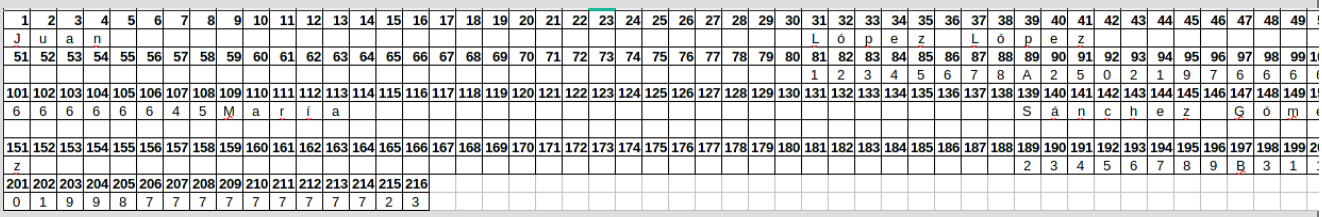
Ejemplo:

En apartados anteriores definimos el registro de longitud fija (porque todos los campos tienen longitud fija) persona. Para representar los datos de las personas:

- Persona 1
  - **nombre**: Juan
  - **apellidos**: López López
  - **dni**: 12345678A
  - **fecha-de-nacimiento**:25/02/1976
  - **telefono**: 666666666
  - **edad**: 45
- Persona 2
  - **nombre**: Maria
  - **apellidos**: Sanchez Gómez
  - **dni**: 23456789B

- **fecha-de-nacimiento**: 31/10/1998
- **telefono**: 777777777
- **edad**: 23

En un fichero de registros de longitud fija la información se almacenaría



Como se puede ver, cada registro ocupa 30+50+9+8+9+2=108 bytes y por lo tanto 2 registros ocupan 2 * 108 = 216 bytes. Se puede comprobar a gran cantidad de espacio desperdiciado en los campos `nombre` y `apellidos`. El primer registro comienza en el byte número 1, el segundo en el 109 (1 + 108), el tercero en el 217 (1 + 2 *108), el cuarto en el 325 (1 + 2 *108), etc.

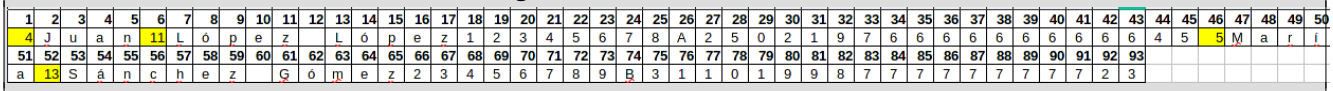
3.2.2 Ficheros de registros de longitud variable

En este tipo de fichero los registros pueden ser de distintos tipos o bien ser todos del mismo tipo pero tener campos de longitud variable. Por lo tanto el espacio de almacenamiento que ocupa cada registro no es fijo sino que debe almacenarse como parte del registro.

Su principal desventaja es que son más difíciles de manipular puesto que hay que realizar cálculos complejos o utilizar estructuras auxiliares (ver los índices más adelante) para poder manipularlos de forma eficiente.

Tienen a su favor que aprovechan el espacio de forma más eficiente.

Ejemplo:
En apartados anteriores definimos el registro de longitud fija (porque todos los campos Supongamos ahora que los campos `nombre` y `apellidos` tienen longitud variable pero un tamaño máximo de 30 y 50, respectivamente. Para almacenar la longitud se colocará como primer valor de los campos `nombre` y `apellidos` la longitud que ocupan, a fin de saber donde terminan.
Ahora, la información se almacenaría de la siguiente manera:



Los bytes con el fondo amarillo son los que almacenan las longitudes de los campos `nombre` y `apellidos`.
Como se puede ver, ahora los registros no ocupan siempre lo mismo. El primer registro ocupa 45 bytes mientras el segundo ocupa 48. Se ahorra mucho espacio (93 bytes frente a 216 que ocupaban en la versión con longitud fija, un ahorro del 57%) pero ahora la búsqueda de los registros es más difícil porque hay que ir leyendo los registros uno a uno y sumando las longitudes para poder determinar donde empieza o acaba cada uno. Se puede mejorar un poco el rendimiento si se incluye la longitud de cada registro completo al inicio pero esto añade algo más de espacio y de complejidad (si hay que cambiar el contenido de un campo de longitud variable lo más seguro es que haya que

actualizar la longitud del registro. Además se complican las inserciones, modificaciones y borrados, como se verá más adelante

3.2.3 Organización del espacio del almacenamiento de los ficheros

Como se comentó anteriormente, una decisión importante que hay que tomar a la hora de trabajar con ficheros es decidir cómo se colocan los datos de cada registro en los bloques del dispositivo de almacenamiento.

Existen dos formas básicas de organización del espacio de los ficheros: Secuencial y relativa.

3.2.3.1 Organización secuencial

En la organización secuencial se van almacenando los registros en bloques consecutivos, en el mismo orden en que se van añadiendo al fichero. Un nuevo registro siempre se añade al último bloque del fichero, si tiene espacio suficiente y si no se escribe en el siguiente.

Ejemplo:
A un nuevo fichero secuencial se le añade el registro de clave 10. El fichero quedaría:

10

A continuación se añaden los registros con claves 27, 19 y 21. El fichero quedaría ahora:

10
27
19
21

La inserción de nuevos registros es muy eficiente puesto que únicamente hay que elegir el último bloque disponible.

La búsqueda es poco eficiente ya que si se desea localizar un registro determinado hay que ir consultando los registros uno a uno desde el inicio del fichero hasta que se localice el registro buscado. Como media habrá que recorrer la mitad de los registros para localizar el deseado y en el caso de que una búsqueda sea fallida (el registro que se busca no está en el fichero) hay que recorrer **todo** el fichero.

Ejemplo:
Si se desea localizar el registro con clave 19 habrá que ir al inicio del archivo e ir leyendo registros hasta que se localice el 19, lo que implicará leer (y descartar) los registros 10 y 27.
Si se busca el registro con clave 53 (que no se encuentra en el fichero) habrá que leer los cuatro registros antes de decidir que efectivamente, el registro no está almacenado.

El borrado de datos también es ineficiente puesto que al ir borrando registros van quedando "huecos" dentro del fichero. Este problema se puede paliar modificando el sistema de inserción para que "reutilice" huecos al insertar registros, siempre y cuando éstos ocupen una longitud menor que la del hueco. Sin embargo este sistema complica el sistema de inserción, ralentizándolo. En algunos sistemas se realiza periódicamente una operación llamada "compactación" por lo que se crea un nuevo fichero a partir de uno antiguo, copiando los registros no borrados y reemplazando luego el fichero original. Esta operación toma tiempo, sin embargo y en muchos casos requiere una parada del sistema mientras se realiza, lo que la hace inapropiada para sistemas que deben funcionar de manera continua.

Ejemplo:
Si se elimina el registro con clave 19, el fichero quedaría:

10
27
21

Y si ahora se añade un nuevo registro, el de clave 33, pasaría a:

10
27
21
33

Y tras la compactación:

10
27
21
33

La modificación puede ser simple, si los registros son de longitud fija, o más compleja, si los registros son de longitud variable. En este último caso, si el tamaño del registro modificado es mayor que el que tenía originalmente, no cabrá en el hueco en el que estaba almacenado. La solución es eliminar el registro antiguo y almacenar el registro con los cambios al final del fichero, como si se realizara una inserción.

Ejemplo:
Si los registros son de longitud fija y se modifica el registro de clave 21, el nuevo fichero quedaría:

10
27
21
33

El fichero no parece haber sufrido ningún cambio y estructuralmente así ha sido. El único cambio es en el *contenido* del registro 21.
Sin embargo, si se diera la mala suerte de que los registros son de longitud variable y el nuevo contenido del registro 21 tuviera un tamaño superior al del hueco que lo alberga, habría que realizar la modificación mediante borrado e inserción:

10
27
33
21

Como se puede ver, la organización secuencial es simple de implementar pero no es muy eficiente. Aún así puede ser la adecuada para ficheros que se suelen procesar siempre de forma completa y en los que la principal operación de modificación es la inserción y sufre pocas modificaciones y borrados. En este caso puede ser una organización suficientemente eficiente. Por desgracia, los datos en sistemas reales no siempre se adaptan a estas restricciones, por lo que el sistema secuencial no será el más adecuado para ellas. No obstante, y a fin de mejorar su rendimiento, han surgido variaciones de la organización secuencial que palían los inconvenientes arriba comentados. Las más empleadas son:

- **Organización secuencial indexada.** En esta organización se utilizan dos ficheros para guardar la información que antes se guardaba en uno sólo. Uno de ellos almacena los datos exactamente de la misma forma que en la organización secuencial. El otro fichero, denominado índice, contiene la posición de cada registro dentro del fichero secuencial, junto con el valor del campo clave o principal de ese registro, de forma que se puede acceder a él de forma directa sin tener que pasar por todos los anteriores, lo que era el principal problema de la organización secuencial pura. Los sistemas más potentes permiten tener más de un índice por fichero de forma que se pueden organizar búsquedas por distintos campos.
- **Organización secuencial encadenada.** En esta organización, cada registro lleva un campo oculto adicional que indica la posición del siguiente registro. De esta forma, aunque los registros se vayan insertando siempre al final del fichero, los campos de posición permiten establecer otro orden a la hora de recorrer el fichero, lo cual puede ser importante si el acceso debe hacerse de forma secuencial pero en un orden distinto al de la inserción.

3.2.3.2 Organización ordenada o relativa

En este tipo de organización, los registros se almacenan en orden según el valor de uno de sus campos, llamado *campo de ordenación*. Usualmente este campo también tiene un valor único para cada registro. En este caso se denomina *campo clave*.

Ejemplo:
A un nuevo fichero relativo se le añade el registro de clave 10. El fichero quedaría:

10

A continuación se añaden los registros con claves 27, 19 y 21. El fichero quedaría ahora:

10
19
21
27

La inserción de registros es poco eficiente porque hay que mantener la ordenación. En muchos casos esto exige, o bien mover registros hacia el final del archivo para hacer hueco, o bien utilizar ficheros de maniobra.

Las búsquedas, sin embargo, son muy eficientes si se realizan por el campo de ordenación ya que se puede realizar búsqueda binaria cuyo tiempo de búsqueda es logarítmico respecto al número total de registros. Si la búsqueda se hace por otros campos tiene la misma eficiencia que la organización secuencial simple.

Respecto a los borrados, su eficiencia es similar a la de los ficheros secuenciales.

La modificación no presenta problemas salvo que:

- Los registros sean de longitud variable y la longitud del nuevo contenido sea mayor que el del contenido actual. Esto exigirá mover el resto de registros hasta el final del fichero para hacer espacio, lo que puede ser muy costoso.
- Se modifique el contenido del campo de ordenación. Esto puede ocasionar que el registro deba ser movido, lo cual también es costoso.

3.2.4 Índices

Los índices son estructuras de datos que relacionan valores de un campo (normalmente el campo clave) de un registro con la posición que ocupa ese registro dentro del fichero de datos, que es el fichero en el que se almacena realmente la información.

Son similares a los índices analíticos que vienen al final de los libros de texto o científicos en los que aparece una lista de la palabras o términos que aparecen en el libro junto con la primera página en la que aparece dicho término o con una lista de todas las páginas en las que aparece dicho término, según el tipo de índice que se desee.

Hay varios tipos de índices, que se discuten a continuación:

3.2.4.1 Índices Primarios

Un índice primario es un fichero ordenado que contiene registros, de longitud fija con dos campos: Uno es el campo clave del fichero de datos y el otro es un número que indica el bloque del fichero de datos donde se encuentra el registro correspondiente.

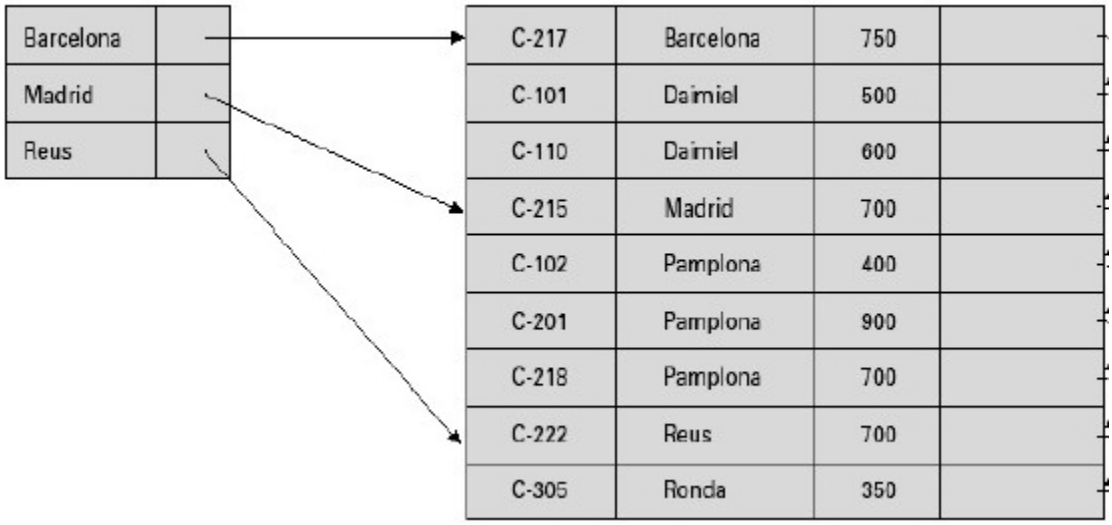
Son mucho más pequeños que los ficheros de datos puesto que sólo contiene un campo de datos y el campo con el número de bloque.

numEmpleado	nombre	apellido	cargo	sexo	fechNac	salario	numOficina
SL21	Jhon	White	Gerente	M	01-Oct-45	300000	B005
SG37	Peter	Denver	Asistente	M	10-Nov-60	120000	B003
SG14	David	Ford	Supervisor	M	09-Sep-58	180000	B003
SA9	Mary	Lee	Asistente	F	17-Sep-59	90000	B007
SG5	Susan	Sarandon	Gerente	F	21-Mar-60	240000	B003
SL41	Julie	Roberts	Asistente	F	13-Jun-63	90000	B005

3.2.4.2 Índices de agrupamiento

Cuando los registros de un fichero están ordenados por un campo no clave (que no tiene un valor único para cada registro), este campo se denomina campo de agrupamiento ya que el fichero contiene grupos de registros con el mismo valor del campo.

Los índices de agrupamiento son índices que se crean sobre estos campos. En este caso cada entrada del índice contiene un valor del campo de agrupamiento y el número de registro *del primer registro del grupo*. De esta forma se puede acceder rápidamente al primer registro del grupo y a partir de ahí recorrer el fichero de forma secuencial.



3.2.4.3 Índices secundarios

Los índices secundarios son iguales a los primarios o de agrupamiento pero cuando el campo sobre el que se indexa no es el campo por el que está ordenado el fichero.

4 Sistemas de bases de datos

Las primeras aplicaciones que manipulaban datos utilizaban ficheros como los descritos en el apartado anterior.

Cada equipo trabajaba con sus propios programas y ficheros y se encargaba de su mantenimiento y gestión.

Con el paso del tiempo y el aumento de la cantidad de la información y de la complejidad de la misma comenzaron a aparecer problemas:

- **Separación y aislamiento de los datos.** Cuando los datos están separados en distintos ficheros, es responsabilidad de la aplicación el coordinar el acceso a los distintos ficheros.
- **Duplicación de datos.** Pueden existir ficheros distintos con copias de los mismos datos, lo que provoca problemas de consistencia de datos (si las copias no son idénticas) y de desperdicio de espacio (ya que hay varias copias de los mismos datos). Por ejemplo, puede ocurrir que el departamento de Contabilidad de una empresa tenga un fichero de empleados con unos datos y el departamento de Personal otro fichero de empleados con algunos datos iguales y otros distintos.
- **Dependencia de los datos de las aplicaciones.** Ya que la definición y manipulación de los ficheros se realiza completamente por las aplicaciones, cualquier cambio en cualquiera de las dos exige la modificación de la aplicación..
- **Incompatibilidad de ficheros.** Dado que la estructura de los ficheros está definida en la aplicación que los maneja, otra aplicación no puede acceder a los ficheros.

- **Consultas fijas.** Dado que la aplicación es la que conoce la estructura de los ficheros, cualquier consulta que se quiera realizar sobre ellos deberá estar codificada en la aplicación. Dicho de otra manera, no se puede realizar ninguna consulta que no esté codificada en la aplicación.
- **Control de concurrencia.** El acceso de varias aplicaciones al mismo fichero a la vez es complicado ya que se pueden provocar inconsistencias al modificar una aplicación un fichero mientras otra lo lee.
- **Catálogo.** Conforme el número de aplicaciones y ficheros crece en una organización se vuelve complicado el saber donde está almacenado un dato. La información está ahi pero no se sabe exactamente en qué fichero, ya que cada departamento tiene los suyos y los gestiona de forma separada a los demás departamentos.

Debido a estos problemas surgieron las bases de datos como un modelo de organización de los datos.

Una base de datos es un conjunto de datos almacenados entre los que existen relaciones lógicas y ha sido diseñada para satisfacer los requerimientos de información de una empresa u organización.

Una base de datos es un conjunto de datos *organizados* en estructuras que se definen una única vez y que se utilizan simultáneamente por varios equipos y usuarios.

En lugar de almacenarse en ficheros desconectados entre si y con información redundante, los datos de una base de datos están centralizados. La base de datos no pertenece a un equipo sino a *toda la organización*.

Además, la base de datos no sólo contiene la información sino que también almacena una descripción de dicha información. A esta descripción se la conoce como *metadatos* y se almacena en una estructura denominada *diccionario de datos o catálogo*.

4.1 Arquitectura de un sistema de bases de datos

En 1975, el comité ANSI propuso un estándar para la creación de sistemas de bases de datos basado en una arquitectura con tres capas o niveles. El objetivo de esta arquitectura es el de separar los distintos niveles de abstracción desde los que se puede ver el esquema de una base de datos. Dicho de otra forma, son formas distintas de ver una misma base de datos, desde la vista más concreta hasta la de más alto nivel. Los tres niveles o capas son:

- **Nivel interno.** En este nivel se describe la estructura física de la base de datos. En este nivel se describen con detalle el almacenamiento de la base de datos, así como los métodos de acceso. Se habla de ficheros, discos, directorios, índices, etc.
- **Nivel medio.** En este nivel se describe la estructura de la base de datos completa para toda la organización mediante un esquema conceptual. Se ocultan los detalles de cómo se almacenan los datos y se centra en describir estos datos. En este nivel se habla de entidades, atributos, relaciones, restricciones, etc.
- **Nivel externo.** En este nivel se describen las vistas de usuario. Cada vista de usuario describe la parte de la base de datos que interesa a un usuario o grupo y oculta el resto. De esta forma un usuario sólo tiene acceso a la parte de los datos que le atañe. En este nivel se utiliza un esquema conceptual o lógico para describir las vistas. A este nivel es al que acceden los programas de aplicación.

Hay que recordar que los tres niveles contienen distintas descripciones *de los mismos datos*, pero en distintos niveles de abstracción. Los únicos datos que existen están en el nivel físico almacenados en un dispositivo de almacenamiento.

Esta arquitectura intenta obtener la *independencia de los datos*, que es la capacidad de un sistema de permitir cambios en un nivel sin que éste afecte a los niveles superiores. Se pueden definir dos tipos de independencia de datos:

- **Independencia lógica.** Es la capacidad de modificar el esquema conceptual (nivel medio) sin tener que alterar las vistas y por lo tanto, las aplicaciones. Se puede modificar el esquema para ampliar la base de datos o para reducirla. Si se reduce, las aplicaciones que no se refieran a los datos eliminados no deberían verse afectadas.
- **Independencia física.** Es la capacidad de modificar el esquema interno sin tener que alterar el resto de esquemas. Por ejemplo, puede ser necesario reorganizar los ficheros para mejorar el rendimiento o añadir / quitar índices. Es más fácil de conseguir que la independencia lógica, que normalmente no se consigue.

4.2 Modelos de datos

Una base de datos contendrá información (datos) correspondiente a "cosas" que existen en el mundo real (personas, objetos, situaciones, relaciones, etc.) pero, obviamente, no contendrá **toda** la información posible sobre estas "cosas". Por ejemplo, si en nuestra base de datos guardamos información sobre cada cliente de la empresa, sólo se guardará la información sobre los mismos que sea pertinente o útil para nuestra organización, descartándose el resto de información por irrelevante (nombre y apellidos, dirección, DNI podrían ser datos necesarios pero color de pelo o de ojos, número de hijos o estado civil parece que no lo sería).

La descripción simplificada de una parte del mundo real se denomina *modelo* y es un concepto ampliamente utilizado en muchos campos del conocimiento. Un modelo consiste en una descripción simplificada de una parte de la realidad a fin de poder ser manipulado más fácilmente. Por ejemplo, si se quiere saber cómo se comportará un diseño de avión ante las corrientes de aire no se construye el avión completo (con motores, asientos, baños, etc.) y se prueba en vuelo sino que se crea una maqueta a escala con la forma externa más parecida posible al avión real pero sin sus "tripas" y se prueba con ventiladores en lugar de lanzarlo al aire. Esta maqueta es un modelo del avión real en el que se han descartado un montón de cosas (motores, depósitos de combustible, sistemas de navegación, etc.) y se ha centrado en la forma externa. Obviamente no es un avión real pero es perfectamente válido para estudiar su comportamiento aerodinámico.

Un modelo de datos funciona de forma similar. Es una representación de la información de una parte de la realidad que contiene sólo la información que se considera relevante para el propósito para el que se construye el modelo. La información irrelevante no aparece en el mismo.

Un modelo de datos nos permite manipular la información de forma que se compruebe si es correcta, completa, consistente, etc. sin necesidad de complicaciones innecesarias por información que en realidad no nos importa.

Una definición más formal sería: *Un modelo de datos es una colección de herramientas conceptuales (mentales) que describen los datos y las relaciones que existen entre los mismos, así como sus restricciones.* Por herramientas conceptuales o mentales entendemos una serie de conceptos o ideas simples que nos permiten analizar o representar una realidad compleja por composición. "Describen los datos y relaciones" se refiere a que describen los datos y las

relaciones entre ellos, lo cual también es una información importante. Por ejemplo, un nombre o DNI sueltos dan poca información pero juntos nos indican que una persona con el nombre dado tiene un DNI con el número que se proporciona: ambos datos pertenecen a la misma entidad (cliente, persona, empleado, etc.). Esa información es una relación. Las restricciones sobre los datos son reglas que se aplican a los mismos y limitan o definen los mismos. Por ejemplo, un DNI debe constar de ocho números y una letra. Cualquier otra combinación de letras y números no es un DNI válido. Además, mediante un algoritmo ya definido, la letra está relacionada con los números de forma que para una combinación dada de 8 números sólo le corresponde una letra determinada y ninguna otra.

4.3 Tipos de modelos

Dado que la arquitectura de la base de datos es de tres niveles (externo, medio e interno), hay que modelar los datos en cada nivel. Por lo tanto, existen tres tipos de modelos:

- Modelos conceptuales.
- Modelos lógicos.
- Modelos físicos.

4.3.1 Modelos conceptuales

Se utilizan para describir los datos de una forma parecida a como los manipulamos las personas, aunque formalizando la descripción a fin de que sea precisa y sin ambigüedades.

Son modelos muy flexibles y permiten expresar datos, relaciones y restricciones explícitamente.

Existen muchos modelos pero el más extendido y utilizado por su sencillez y eficiencia es el *Modelo Entidad-Relación* o MER.

El MER representa la realidad a través de *entidades*, que son "cosas" que existen de forma independiente y que se distinguen de "cosas" similares por sus características. Por ejemplo, si estamos creando un modelo de la información que se gestiona en el departamento de nóminas de una empresa tendríamos, con casi total seguridad, una entidad denominada **empleado** que contendría las características relavantes para nosotros que tiene cada una de las personas que trabaja para la empresa tales como DNI, nombre y apellidos, dirección, salario, número de hijos, etc.

A estas características de las entidades se les denomina *atributos*. Un atributo de una entidad es un dato que tienen todas las ocurrencias de dicha entidad.

A cada una de las distintas ocurrencias de una entidad se le denomina *instancia* de esa entidad. Por ejemplo, Juan Antonio Perez, con DNI 88888888A, que vive en la Avenida de los Tejos, 16, cobra 1500 euros al mes y no tiene hijos sería una instancia de la entidad empleado y Manuel Sanchez Carvajal, con DNI 99999999Z, dirección en Calle Ajoporro, 23, sueldo 1456 euros mensuales y con un hijo sería otra. Son dos empleados distintos pero tienen características comunes, aunque el valor concreto de las mismas sean distintos. Hay que hacer hincapié aquí en que lo importante es que los dos empleados tienen las mismas características (por ejemplo, direccion) aunque el valor concreto de cada una será posiblemente distinto para cada uno (aunque podría ser igual, por ejemplo, dos empleados podrían ser padre e hijo y viven en el mismo domicilio).

Asimismo el modelo también describe las relaciones entre las entidades. Estas relaciones describen vínculos entre instancias de dos o más entidades. Por ejemplo, la entidad **empleado** y la entidad **sucursal** podrían estar vinculadas con la relación **trabaja en** que informa de en cual oficina trabaja cada empleado, o, dicho de otra forma, qué empleados trabajan en una oficina determinada.

Más adelante estudiaremos este modelo con más detalle.

4.3.2 Modelos lógicos

Los modelos lógicos utilizan una descripción más formal para describir los datos de forma que se puedan realizar operaciones sobre los mismos de forma no ambigua o mecánica. Dicho de otra forma, son modelos más cercanos a la forma de trabajar de la máquina que los modelos conceptuales, que están más cercanos a la forma de trabajar de las personas. Los modelos lógicos han sufrido una evolución más patente a lo largo del tiempo conforme la tecnología evolucionaba y las bases de datos se hacían más grandes y complejas.

4.3.2.1 Modelo Jerárquico

En el modelo jerárquico las únicas relaciones entre entidades que se permiten son de padres a hijos. Una instancia de una entidad (llamada entidad padre) puede estar relacionada con una o más instancias de otra (entidade hija). Es fácil de programar pero tiene el inconveniente de que es muy rígida y hay muchas relaciones que no cumplen esta restricción. Estas relaciones deben representarse utilizando "trucos" tales como duplicar la información, ocasionando problemas de consistencia.

4.3.2.2 Modelo en red

Más avanzado que el jerárquico permite que una entidad tenga más de un padre, eliminando la restricción que tenía el jerárquico y mejorando la consistencia. Aún así es difícil de administrar.

4.3.2.3 Modelo relacional

En este modelo los datos y las relaciones se representan utilizando el concepto de tabla, lo cual hace que la manipulación sea uniforme.

4.3.2.4 Modelo orientado a objetos

En estos modelos se intenta almacenar el comportamiento de las entidades además de su información. Además permiten establecer relaciones entre los datos que no se pueden hacer fácilmente con los otros modelos, tales como herencia. A pesar de ser más avanzadas que las anteriores, su uso no se ha extendido mucho por la falta de un estándar, por lo que el mercado está muy fragmentado y cada sistema es distinto completamente a los demás.

4.3.2.5 Modelo declarativo

En estos modelos los datos se almacenan como datos y reglas. Permiten consultas muy complejas. Son utilizados principalmente en campos como los sistemas expertos e Inteligencia Artificial.

4.3.3 Modelos físicos

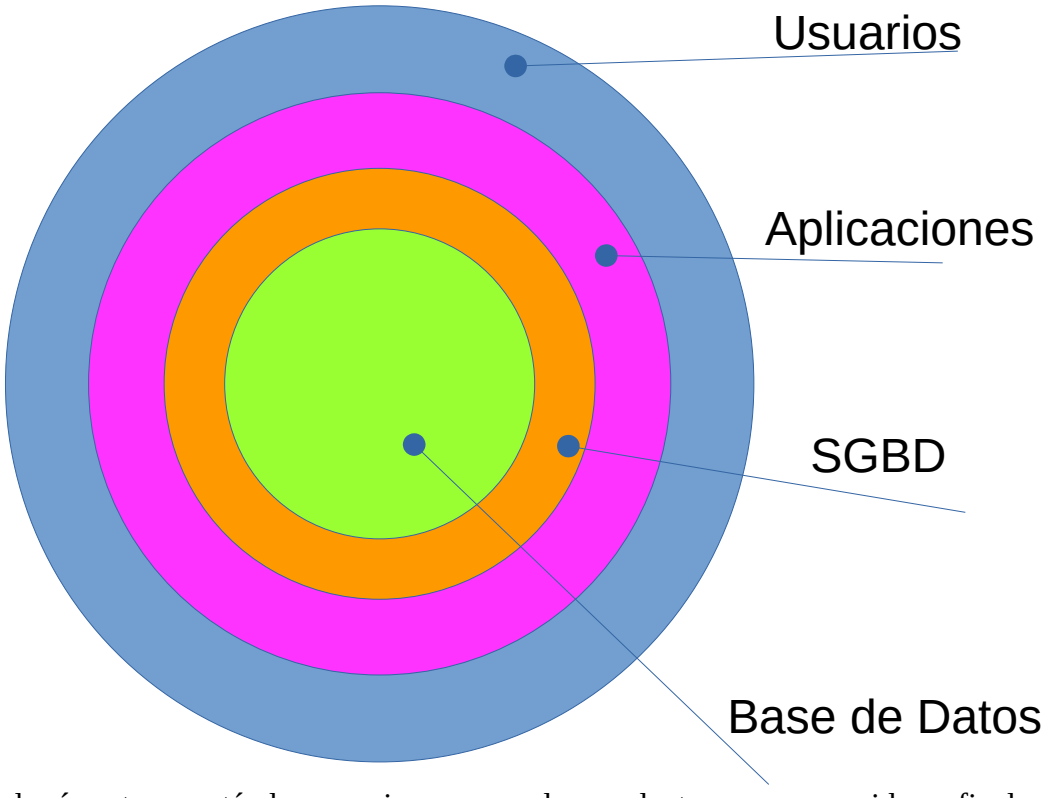
Los modelos físicos de datos describen las estructuras de almacenamiento concretas en el sistema de gestión de bases de datos, indicando qué se almacena, donde y cómo.

El modelo físico depende fuertemente del sistema de gestión de bases de datos empleado, por lo que no se puede dar una descripción general del mismo, debiendo referirse a la documentación del sistema de gestión de bases de datos concreto elegido.

5 Sistemas Gestores de Bases de Datos

Un Sistema de Gestión de Bases de Datos (SGBD) es una aplicación o conjunto de aplicaciones que permite a los usuarios definir, crear y mantener bases de datos así como proporcionar acceso a las mismas. Es un sistema intermediario entre el usuario y las bases de datos.

El funcionamiento de un sistema de información con SGBDs sería el siguiente:



En el nivel más externo están los usuarios, que son los productores y consumidores finales de la información del sistema. Hay que tener en cuenta, sin embargo, que los usuarios no tienen porque ser *personas*. Pueden ser otros sistemas externos que interactúan con el nuestro consumiendo y produciendo información (por ejemplo una cámara que reconoce matrículas en la entrada o salida de un parking)

Los usuarios interactúan con aplicaciones que proporcionan una vista coherente de los datos y facilitan el manejo de los mismos, proporcionando un interfaz más amigable (formularios, informes, ventanas de edición, etc.)
Las aplicaciones se comunican con el SGBD para solicitar datos o realizar altas o modificaciones sobre los mismos.
Los SGBDs son los encargados de mantener las bases de datos físicas en las que se almacena la información y desde donde se recupera cuando sea necesario.

5.1 Objetivos de un SGBD

Los objetivos principales de un SGBD son los siguientes:

- **Permitir la independencia lógica y física de los datos.** Las aplicaciones (o usuarios avanzados) utilizan el modelo lógico al comunicarse con el SGBD. Este se encarga de establecer las correspondencias necesarias entre éste y el modelo físico que se utiliza en las bases de datos (índices, organización de ficheros, etc.)
- **Garantizar la consistencia de los datos.** El sistema debe garantizar que las reglas de consistencia de datos que se le han proporcionado se cumplen en todo momento sin necesidad de intervención del usuario (o a pesar de ésta, en caso de errores humanos).
- **Ofrecer seguridad de acceso a los datos.** El sistema debe garantizar que sólo los usuarios autorizados pueden acceder a los datos y operar con ellos.
- **Gestión de transacciones.** A veces se deben realizar grupos de operaciones de forma atómica, esto es, o bien se realizan todas correctamente o bien no se debe realizar ninguna. De esta forma se evita que se queden operaciones a medio hacer. Por ejemplo, si en una base de datos se va a eliminar un propietario y se van a traspasar sus propiedades a otro, esto exige varias operaciones: una de traspaso por cada propiedad, seguida de una de eliminación. Lo que no se desea es que si ocurre un error (una caida del sistema, por ejemplo) durante uno de los traspasos la base de datos se quede con varias propiedades ya traspasadas, otras sin traspasar y el propietario sin eliminar. Lo ideal es que se mantuviera el estado original de antes de comenzar (ninguna propiedad traspasada) de forma que se pudiera volver a inntar con garantías. A estos grupos de operaciones que se deben de realizar de forma atómica se les denomina *transacciones* y el sistema debe ser capaz de gestionarlas de forma adecuada.
- **Gestión de concurrencia.** Ocurre concurrencia cuando hay acceso simultáneo por parte de distintos usuarios a los mismos datos. En estas condiciones se pueden producir problemas de inconsistencia de los datos que el sistema debe resolver. Esta es una de las razones principales que llevaron a la aparición de los SGBD en un principio.

5.2 Funciones de un SGBD

Para poder cumplir los objetivos anteriormente reseñados, un SGBD debe realizar las siguientes funciones:

- **Gestionar el diccionario de datos.** Un diccionario de datos es una base de datos que mantiene el SGBD automáticamente y que contiene información sobre los datos que se almacenan en la misma. Esto es, es una base de datos sobre los datos. A esta información también se la conoce como *metadatos*. Aunque la

información exacta almacenada en el diccionario depende del SGBD, de forma general, éste contiene lo siguiente:

  - Nombre, tipo y tamaño de cada dato.
  - Relaciones entre los datos.
  - Reglas de integridad sobre los datos.
  - Usuarios con autorización para realizar operaciones sobre los datos, indicando qué usuario puede realizar qué operación sobre qué dato.
  - Estadísticas varias de uso.

- **Garantizar la integridad transaccional.** Se debe garantizar que todas las actualizaciones correspondientes a una transaccion se llevan a cabo o bien que ninguna lo hace. Existen varias técnicas para resolver esto, siendo la más empleada el uso de *diarios* (logs).
- **Gestionar el acceso concurrente a los datos.** Cuando se permite el acceso simultáneo de varios usuarios a los mismos datos se pueden producir problemas de consistencia de los datos si un usuario está escribiendo unos datos que otro usuario está leyendo en ese mismo instante. El sistema debe garantizar que los datos que accede cada usuario son consistentes. Las técnicas más empleadas para asegurar esto es el uso de *bloqueos* (locks).
- **Recuperar datos.** Cuando se produce un fallo (humano o de los sistemas) es importante que la base de datos se recupere a un estado ***consistente*** lo más cerca posible del momento del fallo. Es inevitable que ocurra cierta pérdida de datos pero hay que minimizarla todo lo posible.
- **Proporcionar interfaces de uso.** El SGBD debe ser accesible desde el exterior (si no, ¿para que sirve?). Para ello debe proporcionar canales o accesos por los que conectar con él (red, acceso local, memoria compartida, etc.)
- **Gestionar las restricciones sobre los datos.** Las restricciones sobre los datos son reglas que estos deben cumplir en su estado consistente. El sistema debe garantizar que esto es asi, evitando, por ejemplo, la modificación de un dato si dicha modificación viola una o más reglas de integridad.
- **Proporcionar herramientas de administración.** El SGBD debería proporcionar herramientas para facilitar la administración y uso del mismo. A veces estas herramientas no se proporcionan directamente por el fabricante sino por terceras personas.

5.3 Componentes de un SGBD

Aunque cada sistema es distinto, de forma general todos los SGBDs incluyen los siguientes componentes:

- **Lenguajes de datos.** Se utilizan para dar instrucciones al SGBD. Normalmente se distinguen tres tipos de lenguajes según la funcionalidad que ofrecen:

    - **Lenguaje de Definición de Datos (Data Definition Language, DDL).** Este lenguaje se utiliza para manipular las definiciones de los objetos de la base de datos así como de su estructura, relaciones y restricciones.
    - **Lenguaje de Control de Datos (Data Control Language, DCL).** Este lenguaje se utiliza para manipular la seguridad de los datos (usuarios, grupos, privilegios, etc.)
    - **Lenguaje de Manipulación de Datos (Data Manipulation Language, DML).** Este lenguaje se utiliza para manipular el *contenido* de la base de datos. Permite consultar los datos así como crear, modificar y eliminar datos.

- **Diccionario de datos.** Usualmente se implementa como otra base de datos cualquiera del sistema aunque se manipule de forma distinta.
- **Objetos.** Objetos que se mantienen en bases de datos. El número y tipo depende del SGBD.
- **Herramientas.** Ofrecen un medio de administrar y gestionar el SGBD.
- **Optimizador de consultas.** Traduce el DML a las operaciones básicas a realizar sobre el modelo físico (búsquedas, consulta de índices, etc.). Además procura realizar la traducción de forma que la consulta sea lo más eficiente posible sin que el usuario deba preocuparse por ello.
- **Gestor de transacciones.** Se ocupa de gestionar las transacciones y el acceso concurrente para asegurar la consistencia de la base de datos.
- **Planificador.** En algunos SGBDs se ocupa de lanzar tareas que se deben iniciar de forma automática, ya sea en respuesta a determinadas condiciones de funcionamiento o a una planificación temporal.
- **Gestión de replicación.** Algunos SGBDs proporcionan mecanismos para realizar copias offline de los datos, ya sea a otros SGBDs de reserva o a sistemas de copias de seguridad (backups).

5.4 Usuarios de los SGBDs

Generalmente se distinguen cuatro grupos de usuarios que utilizan los SGBDs. Estos son:

- **Administradores.** Se ocupan del mantenimiento del sistema completo (instalación, puesta en marcha, gestión de copias, gestión de almacenamiento, etc.) Son los usuarios con más "poder" del sistema.
- **Diseñadores.** Realizan el diseño de la base de datos y se encargan de crear los modelos correspondientes y a implementarlos en el SGBD, a veces en colaboración con los administradores.
- **Programadores.** Realizan programas que interactúan con los datos almacenados en el SGBD a través de éste. Suelen tener acceso completo a los datos aunque no se les suele permitir el modificar la estructura de los mismos.
- **Usuarios finales.** Son clientes de las bases de datos que las usan sin necesitar tener conocimiento alguno sobre su funcionamiento, ubicación, etc.

6 SGBDs propietarios y libres

Aunque es un movimiento que existe desde antes, con la popularización de Internet ha proliferado el software libre como alternativa al software comercial cerrado que existía antes.

El mundo de los SGBDs no ha sido refractario a estos cambios que han ocasionado que actualmente se pueda disponer de una amplia paleta de SGBDs tanto propietarios como libres. Este es un factor como otro cualquiera que hay que tomar en cuenta a la hora de seleccionar el SGBD a utilizar para una organización o departamento. Entre los factores a tener en cuenta están:

- **Precio.** Usualmente el software libre es gratuito mientras el propietario no lo es.
- **Funcionalidad.** El software propietario suele ofrecer mejores funcionalidades que el gratuito.
- **Facilidad de uso.** El software propietario suele ofrecer mejores o más sencillas herramientas de administración, uso, etc.
- **Soporte.** El software comercial suele ofrecer mejor soporte aunque bastantes de software libre también ofrecen servicio de soporte (normalmente no gratuito).
- **Comunidad.** Los proyectos de software libre suelen ofrecer una buena comunidad de usuarios dispuestos a ayudar ante los problemas.
- **Control.** Con el software libre el control sobre el SGBD es total ya que nos permite seguir adelante aunque el fabricante del SGBD cese el desarrollo y fabricación del mismo o el soporte.

7 Resumen

En este tema se han descrito los problemas del almacenamiento de datos de forma permanente y a gran escala. Desde los sistemas de almacenamiento hasta los sistemas de ficheros, enumerando las ventajas e inconvenientes de cada implementación.

A continuación se han descrito las bases de datos, como solución a la disparidad de sistemas de almacenamiento y a los problemas de inconsistencias y redundancias en la información perteneciente a una organización.

Por último se han descrito los SGBD como el software que hace posible el manejo de estas bases de datos y que proporciona servicios adicionales como transacciones o acceso concurrente a los datos.

Tema 2

Modelo Conceptual

Tabla de Contenidos

1 Introducción

El modelo conceptual de datos nos da un modelo de los datos y relaciones que existen en el mundo real, dentro de un campo de aplicación concreto, que es el que estamos analizando.
Es, por lo tanto, una *abstracción* de la realidad que contiene la información más relevante de la misma para el campo de aplicación que es está estudiando.
Es el primer paso, y el más importante, a la hora de crear una base de datos útil, funcional y robusta para una organización, ya que de él depende que se obtenga una *foto* realista que sea una descripción fiel de la realidad, por un lado, y lo bastante simple para poder implementarla en un sistema real de bases de datos, por otro.

Para crear el modelo conceptual emplearemos el modelo Entidad/Relación (E/R) de Peter Chen. El primer paso será ver los diferentes elementos que tiene este modelo para representar la información.

2 Modelo E/R

El modelo entidad-relación se compone de:

- Entidades
- Relaciones
- Atributos

2.1 Entidades

El primer paso para elaborar nuestro modelo será descubrir las entidades.
Una entidad es cualquier objeto concreto o abstracto del cual podremos almacenar información. Elementos concretos pueden ser un coche, un socio de una biblioteca, un libro, un cliente, una mesa, etc. Otros elementos abstractos que pueden ser entidades son por ejemplo una idea, un sueño, un proyecto que aún no se ha llevado a cabo, etc. También hay otros que aunque pueden ser más tangibles algunas veces no son tan evidentes, por ejemplo una inversión en bolsa, un divorcio, una sentencia judicial, una declaración de un testigo, etc.
Todo dependerá del contexto en el que estemos trabajando. Para cada elemento que encontremos deberemos preguntarnos si nos interesa guardar información sobre él y en caso afirmativo lo incluiremos como una posible entidad, puede ocurrir que al final no sea necesaria y la descartemos, pero aconsejamos que inicialmente todo aquello que nos parezca oportuno sea incluido, ya que posteriormente tendremos tiempo de filtrarlo y eliminarlo, si corresponde.
El modelo E/R distingue entre entidades ***fuertes*** (también llamadas propias o regulares) y entidades ***débiles***, las primeras son las que tienen existencia por sí mismas y no dependen de nadie. Estas entidades se representan mediante un rectángulo con el nombre en medio. El nombre debe ser un sustantivo. Las entidades débiles dependen de otra entidad fuerte para poder existir y no pueden existir por sí mismas. Su representación gráfica será un rectángulo de línea doble con el nombre en medio.
En el ejemplo tenemos como entidad fuerte un pedido y como entidad débil cada una de las líneas del pedido, pues una lína de pedido no puede existir por sí sola, depende de la existencia previa de un pedido al que pertenezca.

2.1.1 Ocurrencias o Instancias

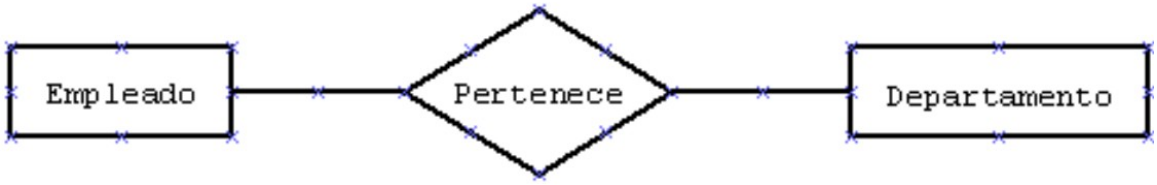
Una *ocurrencia*, tambien llamada *instancia* es un elemento concreto de una entidad. Por ejemplo, para una hipotética entidad entidad `Pedido` (que contiene un número, empresa y fecha), el pedido número 230 realizado por Sealta, S.L. el día 04/03/2013 es una ocurrencia y otra es el pedido número 231 realizado por Javier Martín el día 04/03/2013. Cada pedido es una ocurrencia de la entidad `Pedido`.
Si en lugar de hablar de la entidad `Pedido` habláramos de una entidad `Clientes` (con nombre, dirección y CIF), la empresa Sealta, S .L. situada en Albacete en el polígono Industrial La Florida y con

CIF: B2345934859, será una ocurrencia y Javier Martín que es autónomo y tiene su sede fiscal en Toledo en la Avenida de la Gran Maestranza y con el NIF: 123456789P, será otra ocurrencia.

Ejercicio: Indica al menos cinco entidades en cada uno de los siguientes contextos:
- Un instituto.
- Una liga de fútbol.
- Una agencia de viajes.
- Un supermercado.
- Una empresa de alquiler de coches.

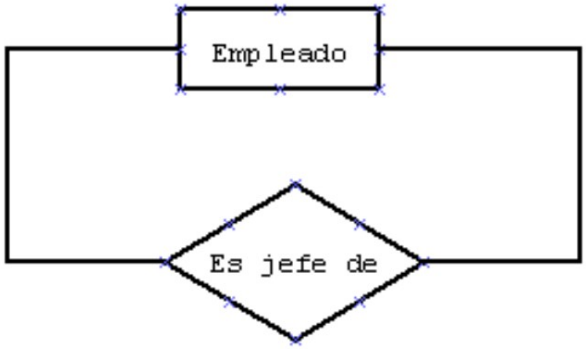
2.2 Relaciones

También llamadas interrelaciones, una relación representa una asociación o correspondencia entre entidades. Es el elemento que nos permitirá relacionar las ocurrencias de las diferentes entidades. Por ejemplo qué pedidos pertenecen a un cliente, qué coche ha sido multado por qué agente, etc. La representación gráfica de una relación es un rombo y en su interior se escribe el nombre de la relación que debe ser un verbo o una acción verbal.
Veamos un ejemplo que aparece en la gran mayoría de las empresas en las que existen varios departamentos y cada uno de los empleados pertenece a un determinado departamento. Tendríamos dos entidades, por un lado `Empleado` y por otro lado `Departamento`. La asociación que utilizamos en este caso es `Pertenece`, los empleados pertenecen a departamentos.

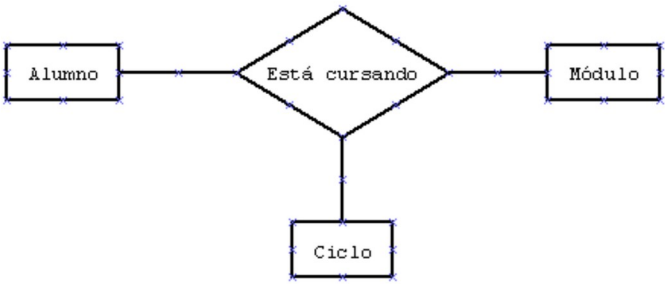


Se denomina ***Grado*** de una relación al número de entidades que participan en esa relación. En el ejemplo anterior hay dos entidades que están asociadas por la relación, en este caso estaríamos hablando de una relación de grado 2. Lo más habitual es encontrar relaciones de grado 2 (también llamadas relaciones binarias), sin embargo también podemos encontrar otros grados como:

- Relaciones de grado 1 o Reflexivas, en las que una relación crea una correspondencia entre unas ocurrencias de una entidad con otras ocurrencias de la misma entidad. Por ejemplo, la relación `Es jefe de` con la entidad `Empleado`. Algunos empleados son jefes de otros empleados.



- Relaciones de grado 3 o Ternarias. En estas relaciones intervienen tres entidades.

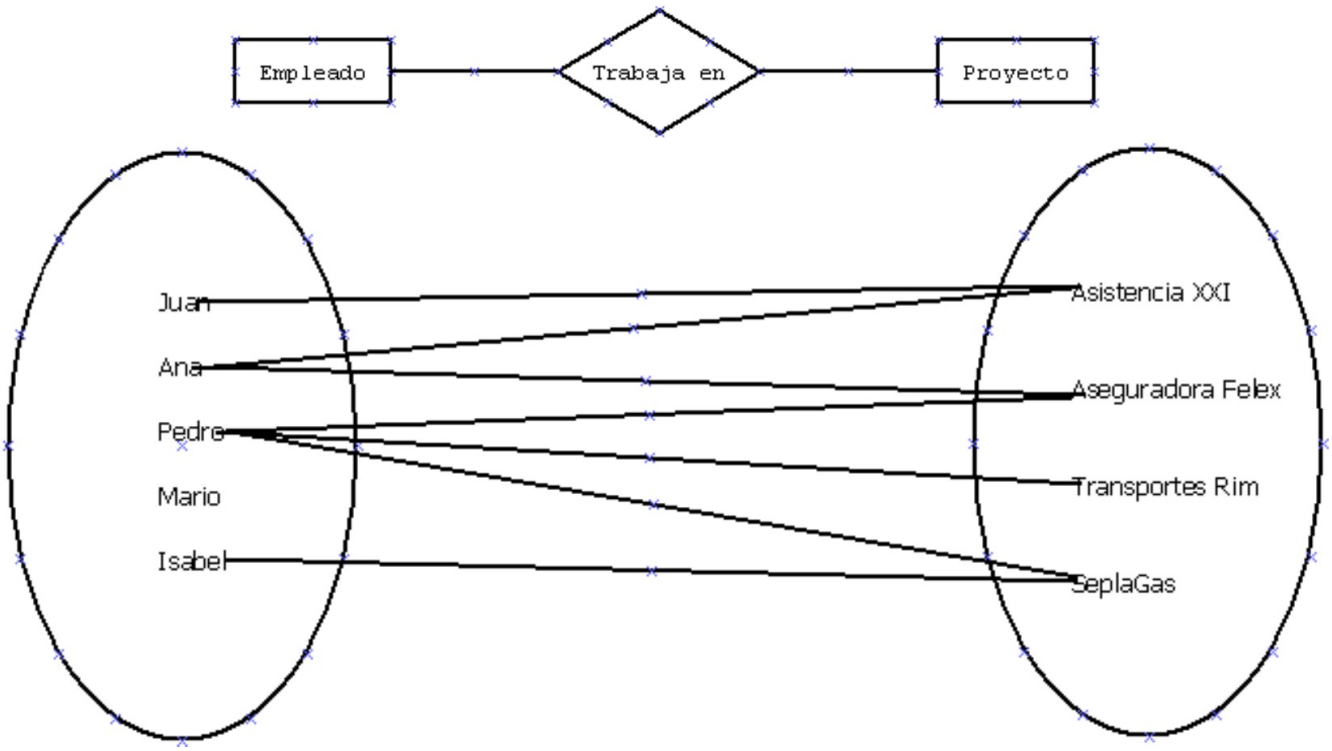


- Aunque es muy poco frecuente, podemos encontrarnos relación que asocien cuatro entidades, en este caso sería una relación de grado 4 y así sucesivamente hasta grado n.

Ejercicio: En el ejercicio anterior elegimos al menos cinco entidades en cada uno de los contextos que indicamos más abajo. Ahora busca las relaciones que pueden existir entre las diferentes entidades que has encontrado e indica el grado correspondiente para cada relación encontrada. Si no has encontrado la entidades, aquí te sugerimos algunas posibles (por supuesto no son las únicas, seguro que tu has encontrado más, algunas iguales o similares y otras distintas.)
- Un instituto.(Posibles entidades: profesores, estudios, cursos, asignaturas o módulos, alumnos, exámenes, etc.)
- Una liga de fútbol. (Posibles entidades: entrenadores, equipos, partidos, jugadores, goles, árbitros, tarjetas, etc.)
- Una agencia de viajes. (Posibles entidades: clientes, tipos de viajes, destinos, medios de transporte, hoteles, empresas de traslado, seguros, etc.)
- Un supermercado. (Posibles entidades: productos, empleados, departamentos, horarios de trabajo, tickets de compra, envíos a domicilio, etc.)
- Una empresa de alquiler de coches. (Posibles entidades: gamas de vehículos, coches, revisiones, seguros, clientes, precios, temporadas, etc.)

2.2.1 Participación

Es el mínimo y máximo número de veces que una ocurrencia de una entidad puede aparecer relacionada con otras ocurrencias de la otra entidad dentro de una relación determinada.
Supongamos que tenemos la entidad `Empleado` y la entidad `Proyecto` asociadas por la relación `Trabaja en`. La participación nos mostrará el número mínimo y máximo de proyectos en los que un empleado puede trabajar y por otro lado indicará, para un proyecto dado, cual es el número mínimo y máximo de empleados que pueden trabajar en él.
Los valores que se emplean son 0, 1 y n. Los valores 0 y 1 se emplean para indicar el mínimo y el 1 y la n para indicar el máximo. Siempre se presentarán los dos valores elegidos entre paréntesis y separados por una coma (mínimo, máximo). Los pares posibles son (0,1), (0,n), (1,1) y (1,n).
Supongamos que preguntamos en la empresa cuales son los empleados y nos dan 5 nombres (Juan, Ana, Pedro, Mario e Isabel) y por otro lado preguntamos cuantos proyectos tienen y nos dan 4 nombres de proyectos (Asistencia XXI, Aseguradora Felex, Transportes Rim y SeplaGas).
Para ver la participación de forma sencilla preguntaremos al cliente en que proyecto trabaja cada uno de los empleados y dibujamos los dos conjuntos dibujando las líneas que muestran la correspondencia de la relación `Trabaja en`.



Como podemos observar Juan trabaja en un proyecto, Ana en dos, Pedro en tres, Mario en ninguno e Isabel en uno. Por otro lado en Asistencia XXI trabajan dos empleados, en aseguradora Felez otros dos empleados, en Transportes Rim un empeado y en SeplaGas dos empleados.
Una vez analizado el resultado debemos realizar las siguientes preguntas (las responderemos nosotros con los conocimientos adquiridos de nuestras conversaciones con el cliente, pero en caso de duda hay que preguntar al cliente, ver cómo ha funcionado la empresa durante los años anteriores, etc.):

- ¿Un empleado tiene que estar siempre trabajando en un proyecto o puede estar sin trabajar en ninguno? En este caso podemos observar como existe un empleado (Mario) que no trabaja en ningún proyecto. Siempre que realicemos estas preguntas debemos fijarnos en la escala temporal, no solo tenemos que preguntarnos por el momento actual, también por el pasado y por el futuro. Por ejemplo, ¿al crear un nuevo empleado tendrá o no tendrá ya un proyecto en el que trabajar? ¿Cuando finalice un proyecto los empleados de ese proyecto ya estarán trabajando en otro o podrá haber un tiempo mientras son reasignados a otro proyecto?, etc. De esta forma obtendremos la participación mínima, un 0 si el empleado puede estar en algún momento sin estar trabajando en un proyecto y un 1 si un empleado siempre va a estar trabajando en un proyecto como mínimo.
- Para obtener la participación máxima debemos preguntarnos ¿un empleado puede trabajar como máximo en un proyecto o puede trabajar en más de un proyecto? En nuestro diagrama está claro que Ana y Pedro están trabajando en más de un proyecto. Para obtener la participación máxima seleccionaremos un 1 si un empleado puede trabajar como máximo en un proyecto o n si puede trabajar en más de un proyecto.
- Después de estas preguntas obtendremos la participación de los empleados, en este caso será (0,n).
- Ahora realizaremos unas preguntas similares para los proyectos. ¿En un proyecto siempre tiene que haber al menos un empleado que trabaja en él? En nuestro caso, consideraremos que

después de hablar con los clientes, nos han dicho que siempre antes de crear un proyecto se ha asignado un responsable que va a trabajar en él. Esto nos indica que todo proyecto siempre tendrá al menos un empleado que trabaja en él. La participación mínima en este caso será un 1.

- ¿En un proyecto solamente puede estar trabajando un empleado o puede haber proyectos en los que trabajen varios empleados? Como podemos apreciar en nuestros conjuntos hay proyectos en los que están trabajando varios empleados. La participación máxima en este caso será n.
- Con la respuestas de estas preguntas obtendremos la participación de los proyectos, en este caso será (1,n).

Colocaremos la participación junto a nuestro diagrama E/R. La participación de los empleados respecto a los proyectos se colocará junto a la entidad `Proyectos`, y la participación de los proyectos respecto a los empleados se colocará junto a la entidad `Empleados`.

Aunque parece que lo estamos colocando al revés, no es así. Se coloca de esta forma para que su lectura sea sencilla, de esta forma podemos leer de izquierda a derecha que un `Empleado` `Trabaja en` 0 ó n `Proyectos`. Y de derecha a izquierda podemos leer que en un `Proyecto` `Trabajan` de 1 a n `Empleados`.

2.2.2 Cardinalidad

Indica el número de relaciones en las que una entidad puede aparecer. Se anota en términos de:

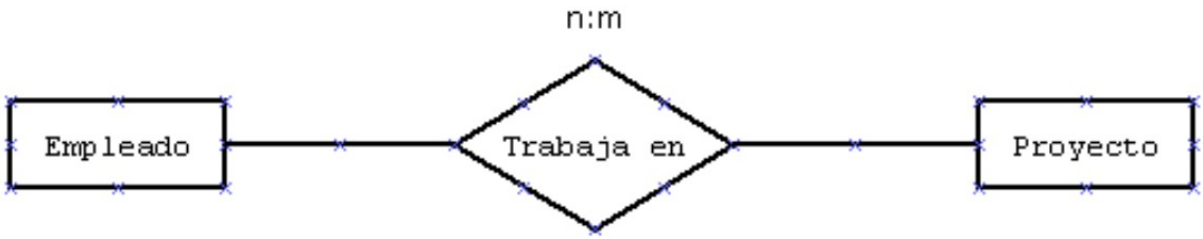
- **Cardinalidad mínima.** Indica el número mínimo de asociaciones en las que aparecerá cada ejemplar de la entidad (el valor que se anota es de cero o uno)
- **Cardinalidad máxima.** Indica el número máximo de relaciones en las que puede aparecer cada ejemplar de la entidad (puede ser uno o muchos)

La Cardinalidad será obtenida a partir de la participación. Partiremos del ejemplo del punto anterior:

Para obtener la cardinalidad tomaremos el valor máximo de cada uno de los pares de las participaciones obtenidas.

- Para la primera vemos cual es el valor máximo de (1, n), en este caso será n.
- Para la segunda vemos cual es el valor máximo de (0, n), en este caso será n.
- El valor de la cardinalidad serán los valores obtenidos separados por el símbolo dos puntos, n:n, pero si dejamos los dos valores con n puede dar lugar a error suponiendo que la cardinalidad debe ser la misma en toda correspondencia, por ello en estos casos una de las n se cambia por m y la cardinalidad sería n:m.

El resultado sería:

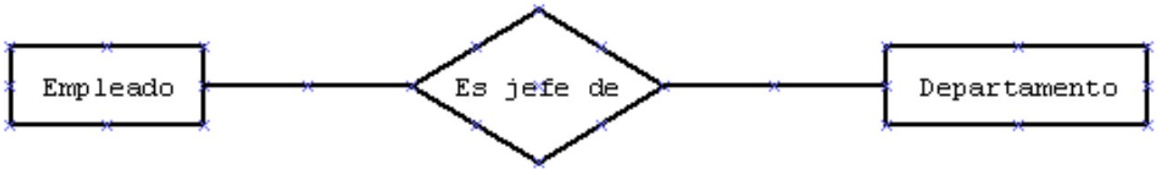


2.2.3 Tipos de cardinalidad

Podemos encontrar varios tipos de cardinalidad

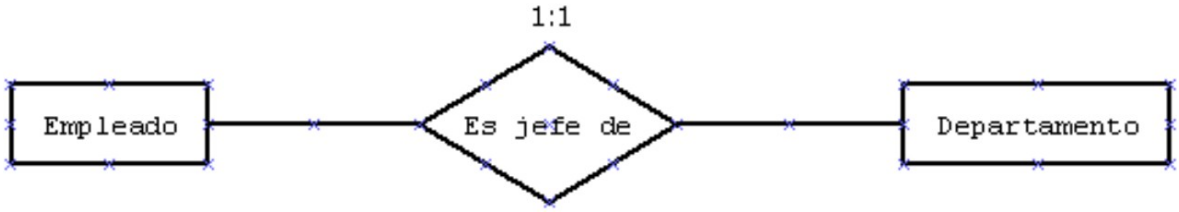
2.2.3.1 Primer tipo. Cardinalidad 1:1.

Partiendo de las entidades ya vistas anteriormente `Empleado` y `Departamento`, vamos a analizar la relación `Es jefe de`.



Primero buscamos la participación. Un empleado puede ser jefe de un departamento o no serlo. Pero vamos a suponer que no puede ocurrir que un empleado que sea jefe de más de un departamento. La participación será (0,1).
Por otro lado un departamento siempre tendrá asignado un jefe de departamento y en un departamento no puede haber más de un jefe. La participación en este caso será (1,1).

Por último obtenemos la cardinalidad con el valor máximo de cada participación. El máximo de (1,1) es 1 y el máximo de (0,1) es 1. Luego la cardinalidad que obtenemos será 1:1

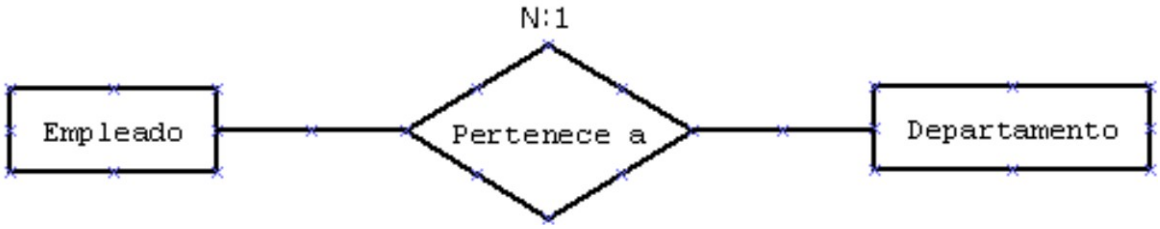


2.2.3.2 Segundo tipo. Cardinalidad 1:N.

Seguimos con las entidades `Empleado` y `Departamento` y con la relación `Pertenece a`.

Buscamos la participación. Un empleado debe pertenecer a un departamento y solo a un departamento. La participación será (1,1). Por otro lado, un departamento siempre tendrá al menos un empleado aunque lo más normal que en un departamento trabajen varios empleados. La participación en este caso será (1,n).

Por último obtenemos la cardinalidad con el valor máximo de cada participación. El máximo de (1,1) es 1 y el máximo de (1,n) es n. Luego la cardinalidad que obtenemos será n:1 (que es lo mismo que 1:n, sólo depende desde el lado de la relación que se mire).

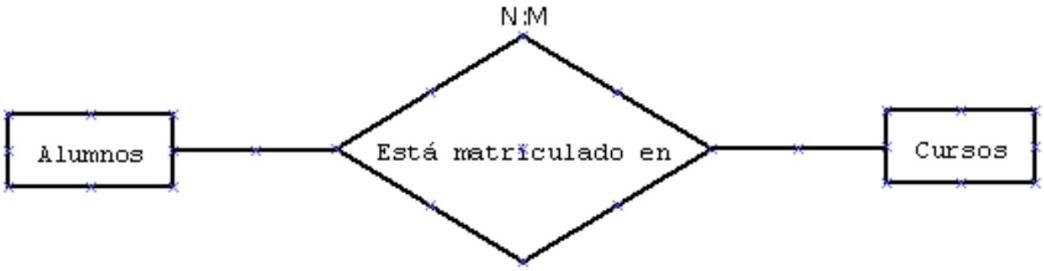


2.2.3.3 Tercer tipo. Cardinalidad N:M.

En este caso vamos a utilizar las entidades `Alumno` y `Curso` y la relación `Está matriculado en`.

Calculamos la participación. Un alumno debe estar matriculado en un curso o en varios. Luego la participación será (1,n). Por otro lado, en un curso puede no haber alumnos matriculados (por ejemplo es un curso nuevo) o puede haber varios alumnos que lo estén realizando. La participación será (0,n).

Por último obtenemos la cardinalidad con el valor máximo de cada participación. El máximo de (0,n) es n y el máximo de (1,n) es n. Luego la cardinalidad que obtenemos sería n:n y lo representaremos como N:M.



2.2.4 Cardinalidad en relaciones de grado distinto de 2

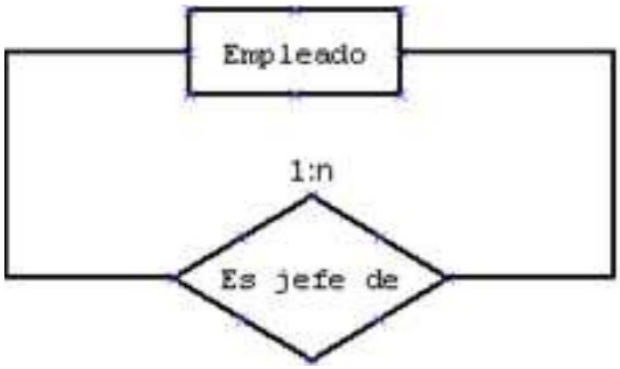
Como ya hemos indicado anteriormente, el grado más común en las relaciones es el 2 pero existen ocasiones en las que son necesarios el uso de relaciones de grado distinto. En esta sección revisaremos ese tipo de relaciones y la aplicación de la cardinalidad.

2.2.4.1 Relaciones de grado 1.

El procedimiento es el mismo que ya se ha descrito anteriormente, la única diferencia es que en ambos extremos de la relación la entidad es la misma. Disponemos de la entidad `Empleado` y la relación con ella misma `Es jefe de`.

Buscamos la participación de esta relación. Un empleado es jefe de 0 empleados (cuando no es jefe) o de n empleados (cuando es jefe) Luego la participación será (0,n).
Por otro lado, un empleado siempre tendrá un jefe (1), excepto el jefe máximo que no tendrá nadie de quien dependa (0). La participación será (0,1).

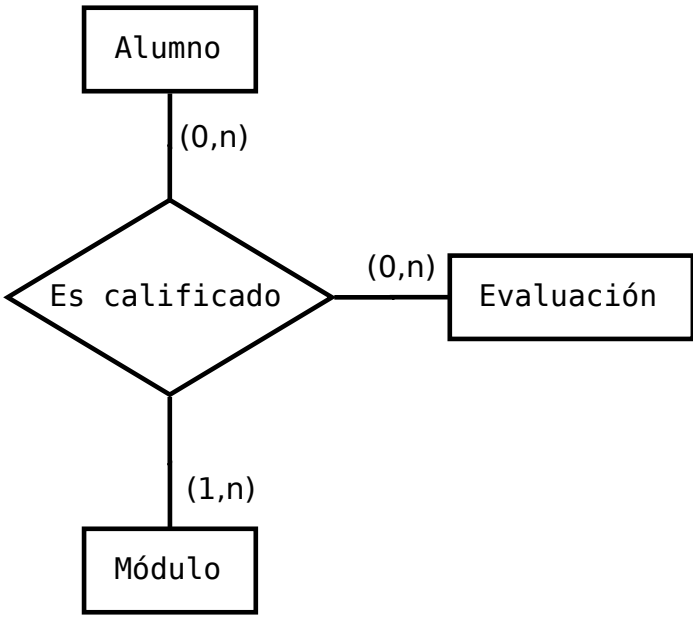
Obtenemos la cardinalidad a partir de los valores máximos de sus pares, en este caso será 1 y n, quedando 1:n.



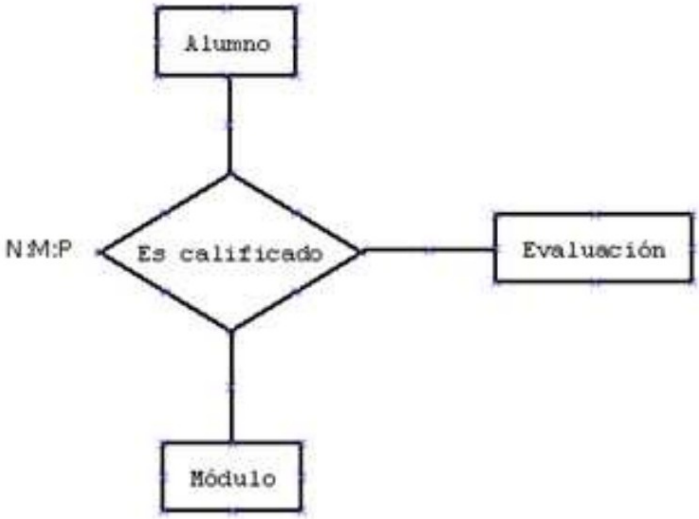
2.2.4.2 Relaciones de grado 3.

La participación en una relación de grado tres no es tan fácil de averiguar como en las de grado 2 o grado 1. Para ello debemos tomar una pareja determinada de dos entidades y ver la participación que tiene la tercera con esa pareja determinada. Después tomamos otra pareja y por último la que queda. Veamos un ejemplo, supongamos que tenemos una relación de grado tres llamada `Es calificado` que relaciona a `Alumno`, con `Módulo` y con `Evaluación`. Las preguntas son:

- Para un alumno y un módulo determinados, ¿en cuantas evaluaciones se puede calificar? Puede ser 0 (si aún no se ha calificado) y un máximo de 3 ó 4, dependiendo de los estudios. Para simplificar vamos a dejarlo en indeterminado (n). Por lo tanto será (0,n)
- Para una evaluación y un alumno determinados, ¿en cuantos módulos se califica?. Está claro que, por lo menos se calificará en un módulo (si no, qué está estudiando) y como máximo será n (los módulos que cursa. Por lo tanto será (1,n)
- Y por ultimo, para una evaluación y un módulo determinados, ¿cuantos alumnos se califican? Aquí hay alguna duda porque no sabemos si hay asignaturas sin alumnos. Si las hubiera, el mínimo sería cero. Si no las hubiera, entonces sería 1. Vamos a tomar como hipótesis que se permiten asignaturas por alumnos. En ese caso el mínimo sería cero. ¿Y el máximo? n. Por lo tanto seria (0,n)



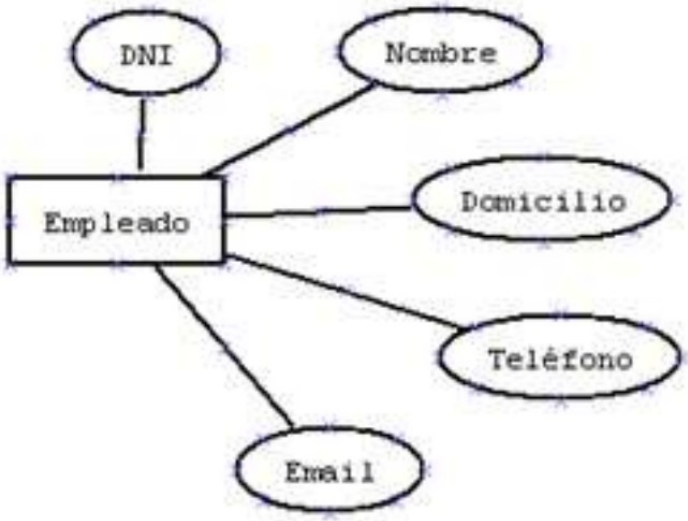
Para obtener la cardinalidad obtendremos el máximo de cada uno de los pares: máximo de (0,n) es n, máximo de (1,n) es n y por último máximo de (0,n) también es n. Quedaría N:N:N, al igual que en las de grado 2, para indicar que no tiene que ser el mismo número de ocurrencias las que se correspondan cambiaremos las letras quedando N:M:P.



Por último, cuando trabajamos con participaciones en relaciones de grado 2, la participación de una entidad se coloca al otro lado de la relación, sin embargo cuando el grado es 3 o superior eso no se puede hacer, pues la participación es respecto de las otras dos entidades combinadas, por ello se deja junto a la misma entidad.

2.3 Atributos

Los atributos son las propiedades o características que deseamos guardar de *una entidad o de una relación*. Los atributos se representan como elipses conectadas al elemento al que pertenecen.
Por ejemplo, algunos de los atributos que puede tener la entidad `Empleado` pueden ser su `DNI`, su `Nombre`, `Domicilio`, `Teléfono` y `Email`. Su representación será:



Existen diferentes clasificaciones de atributos, según la característica que se tenga en cuanta de ellos:

- Atributos simples y compuestos. Los atributos simples contienen un valor simple (número, fecha, etc), mientras que los compuestos contienen otros valores (por ejemplo. dirección contendría calle, número, escalera, etc.)
- Atributos únicos o múltiples: Un atributo único puede contener un sólo valor mientras que un mÍltiple puede contener más de uno (por ejemplo, un atributo teléfono).
- Atributos obligatorios u opcionales. Un atributo obligatorio debe tener un valor siempre, mientras un opcional puede tenerlo o no.

2.3.1 Atributos clave

Un atributo clave son aquellos atributos o conjuntos de atributos (cuando está formado por varios atributos se llama clave compuesta) que son únicos para cada una de las ocurrencias de una entidad. Es decir un atributo clave no puede tener un valor que se repita en varias ocurrencias y tampoco puede ser nulo.
Puede ocurrir que en una entidad encontremos varios campos que sean claves bien de forma aislada o bien en conjunto con otros campos. Por ejemplo, para un empleado podríamos tener un número de registro que la empresa da a los empleados al contratarlos, sería diferente para cada empleado y puede ser una clave. También al contratar a un empleado se le da de alta en la seguridad social y el empleado recibe un número de la seguridad social que es único para cada empleado, sería otra posible clave. Otro atributo evidente que también puede ser clave es el DNI que será diferente para cada empleado.
Todas las claves encontradas para una entidad se denominan claves candidatas y de todas ellas debemos seleccionar una (generalmente será la que más se utilice en el contexto en el que estamos trabajando) que será la que utilizaremos como clave principal.
La clave principal se distinguirá del resto de los atributos porque el nombre del atributo o atributos que forman la clave principal aparecerá subrayado.
Por lo tanto podemos concluir que la clave principal es aquel atributo o conjunto de atributos que permite identificar de forma unívoca cada una de las ocurrencias de una entidad.

2.3.2 Dominio de un atributo

Un dominio es la naturaleza del dato de un atributo de una entidad o relación. Por ejemplo una cadena de caracteres, un valor entero, un valor real, una fecha, un valor lógico (verdadero o falso), una imagen, etc.
Para todos los atributos de una entidad o relación debemos elegir su dominio. Por ejemplo para los atributos de la entidad `Empleado` del ejemplo anterior sería:

Atributo	Dominio
DNI	Cadena de caracteres de longitud 10
Nombre	Cadena de caracteres de longitud 50
Domicilio	
Dirección	Cadena de caracteres de longitud 50
Localidad	Cadena de caracteres de longitud 50
Código Postal	Número entero
Provincia	Cadena de caracteres de longitud 30
Teléfono	Número entero
Email	Cadena de caracteres de longitud 70

3 Guía para la creación del modelo E/R

A continuación vamos a presentar una guía a seguir para crear nuestro modelo conceptual de datos utilizando el modelo E/R.

1. Estudiar el problema hasta que seamos capaces de reproducir la situación de memoria. Es muy importante leer y leer la documentación de la que dispongamos (reuniones con el cliente, documentos de la empresa, informes que genera o necesita generar la empresa, etc.) hasta comprender completamente el funcionamiento que vamos a modelar, plantearnos todas las dudas que se nos ocurran e intentar buscar una solución justificada para cada una para después comprobarlo con nuestro cliente (o con el profesor si se trata de un ejercicio de clase).
2. Buscar las posibles entidades que tendrá nuestro modelo. Puede ser que al principio no salgan todas o que pongamos entidades que después no serán necesarias. No hay problema, puesto que la creación del modelo conceptual de datos deberá ser revisada varias veces y en cada una de esas revisiones iremos depurando nuestro modelo hasta conseguir el modelo adecuado para la información que vamos a implantar en la empresa. Una forma fácil de buscar las entidades candidatas es localizar los *nombres o sustantivos* en la definición del problema.
3. Buscar las relaciones existentes entre ellas que puedan ser interesantes para la información que deseamos almacenar en nuestra base de datos. Al igual que en el punto anterior, puede ser que pongamos alguna de sobra o bien que nos falte alguna. En las sucesivas revisiones que haremos podremos ir mejorando nuestro modelo. Una vez localizadas las relaciones buscaremos sus participaciones y su cardinalidad. Al igual que con las entidades, una forma sencilla de localizar relaciones es buscar *los verbos o acciones* que aparecen en la especificación.
4. Localizar los atributos de las entidades y de las relaciones que hayamos establecido en los puntos anteriores. Además para cada entidad deberemos buscar las claves candidatas (puede ser un atributo o un conjunto de atributos) y de ellas elegir la clave principal de cada entidad (recuerda que si hay varias claves candidatas debemos elegir como principal aquella que tenga más sentido según el contexto con el que estemos trabajando). También debemos detectar los atributos compuestos y los elementos que los forman. Los atributos también suelen aparecer como *nombres o sustantivos* en la especificación. Una vez que tenemos todos los atributos especificaremos el dominio de cada uno de ellos.

Cuando hemos terminado el último paso tenemos el primer boceto de nuestro modelo de datos. Ahora es un proceso de refinamiento. Debemos tomar nuestro modelo y toda la información del problema que estamos representando y verificar que toda la información relevante está reflejada en nuestro modelo, si es así habremos terminado con este paso, si no lo es revisaremos los pasos uno a uno hasta conseguir nuestro objetivo. Hay que revisar asimismo la *navegabilidad* del modelo, esto es, si dos entidades que están directamente o indirectamente relacionadas son alcanzables siguiendo relaciones de nuestro modelo. Si es asi, el modelo será correcto.

4 Modelo E/R ampliado

El modelo E/R ampliado introduce un nuevo concepto: La generalización/especialización, también conocido como la relación "Es un" (Is a).

Una **especialización** consiste en crear nuevas entidades (llamadas subentidades o subtipos) a partir de una entidad ya existente (llamada superentidad o supertipo). **TODAS** las subentidades tendrán todos los atributos que tiene la superentidad y además definirán atributos propios de cada una de ellas.

Ejemplo:

Supongamos que al analizar los datos de una organización encontramos una entidad llamada empleado, que representa una de las personas que trabajan en la organización. Este empleado tendrá unos atributos, como, por ejemplo, nombre, dni y número de la seguridad social. Al continuar el análisis nos cuentan que en dicha organización, los proyectos los realizan empleados pero sólo los empleados que tienen perfil técnico. Un empleado técnico es a todos los efectos un empleado normal, con los datos indicados antes pero, además, tiene una cualificación (por ejemplo, peón electricista u oficial electricista). Este, obviamente, sería un atributo de empleado, pero *no todos los empleados lo tienen* ya que hay otros, como por ejemplo los comerciales, que no tiene perfil técnico pero si tienen un porcentaje de comisión que se llevan por cada venta, dato que no se aplica a los técnicos.

En este caso vemos que tenemos tres entidades. Por un lado empleado, que tiene la información que se aplica a *cualquier* empleado de la empresa y por otro técnico y comercial, que además de ser empleados, tienen alguna información adicional asociada a su puesto de trabajo concreto.

Éste sería un ejemplo de especialización. Las entidades técnico y comercial **especializan** a la entidad empleado, pues tienen la misma información pero además cada una de ellas añade información que les es propia. La entidad empleado sería la superentidad y las entidades comercial y técnico las subentidades.

El proceso opuesto se denomina **generalización**. En este caso, observamos que varias entidades tienen atributos comunes y se extraen estos atributos en una superentidad, dejando en las entidades ya existentes sólo los atributos que les son propios.

En ambos casos, la relación es la misma. Sólo cambia la forma en que se llega a ella: dividiendo una entidad en superentidad/subentidades que la especializan o partiendo de entidades ya existentes, se convierten en subentidades y los atributos comunes se introduce en una nueva superentidad que los generaliza.

Es importante tener en cuenta que esto no se debe utilizar como un método para combinar atributos, extrayéndolos a una superclase. Se debe cumplir la premisa de que se puede afirmar que una subentidad *también es* la superentidad, es decir, es las dos cosas.

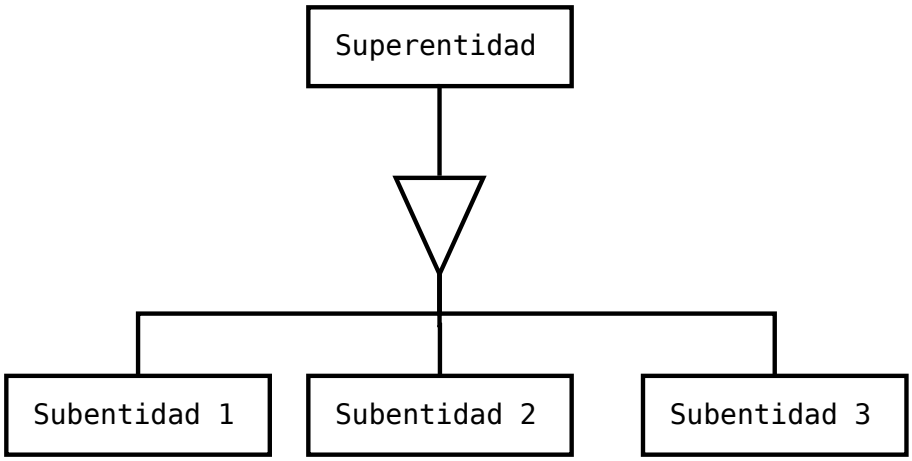
Ejemplo:

Supongamos el ejemplo que hemos dado antes de empleado, técnico y comercial. Además de tener atributos comunes, se puede afirmar que un comercial *es un* empleado. Lo mismo se puede afirmar de un técnico. En ese caso la relación de especialización/generalización se aplica correctamente.

Supongamos ahora que en nuestro análisis aparencen dos entidades: pájaro, con atributos especie, color y velocidad, y automóvil, con atributos marca, modelo, color y velociadad.

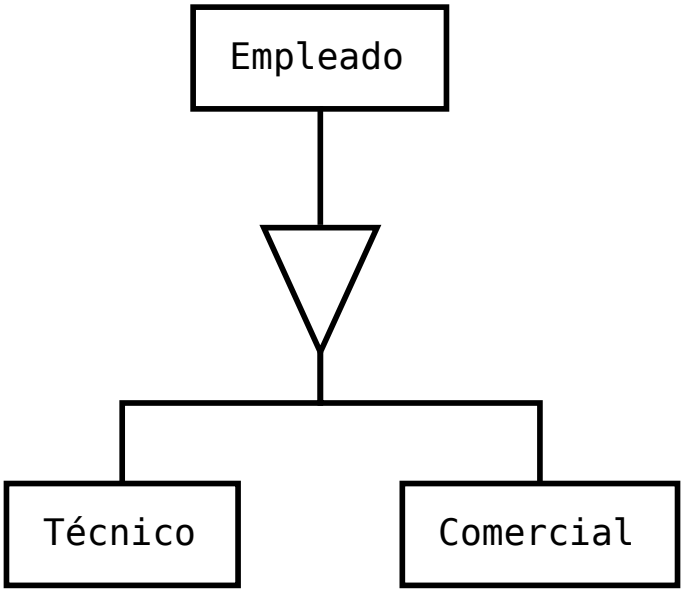
Se podría pensar que ya que los dos comparten atributos color y velocidad sería una buena idea el sacar esos atributos a una superentidad, a fin de aprovechar espacio. El problema es que no se puede encontrar, excepto con un gran esfuerzo de la imaginación, una superentidad que generalice a pájaro y automóvil ("cosas con color y velocidad", por ejemplo). En ese caso no hay que intentar generalizar y dejar las entidades como están.

La representación en el modelo entidad/relación de este nuevo tipo de relación se hace representándola mediante un triángulo (equilátero o isosceles) con la punta hacia abajo y un lado hacia la superentidad, utilizando líneas, como en el resto de relaciones, para conectar las entidades relacionadas.



Ejemplo:

El ejemplo que hemos estado siguiendo en este capítulo (empleado, técnico, comercial), se representaría como:



Las relaciones de generalización/especialización se pueden "afinar" un poco más, añadiendo información sobre como participan las entidades en la relación. Desde este punto de vista se pueden ver dos formas de participación (inclusiva / exclusiva, total / parcial). Ambas son compatibles entre si, esto es, una relación de espacialización puede ser inclusiva total, otra exclusiva parcial, otra exclusiva total y por último otra puede ser inclusiva parcial.

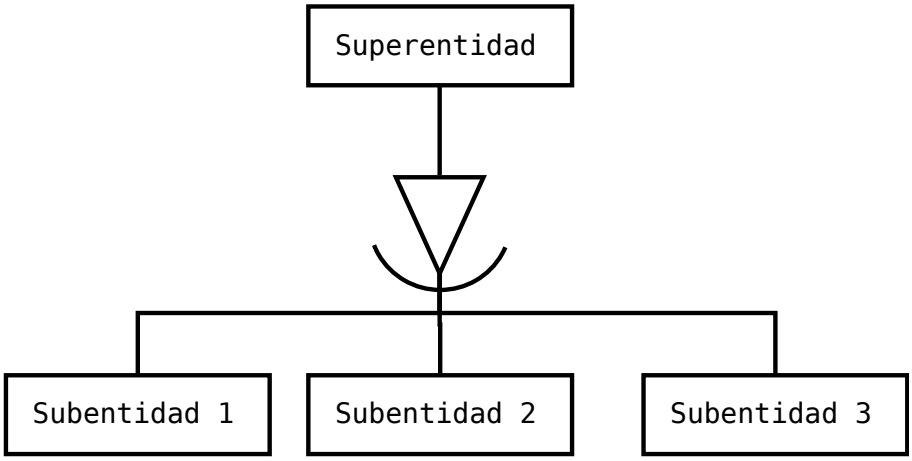
Vamos a describir estos tipos de participación.

4.1 Especialización inclusiva o exclusiva

En el tipo de especialización exclusiva, cada una de las ocurrencias de la superentidad sólo puede pertenecer a uno de los subtipos. Por ejemplo, un empleado debe ser o bien técnico o bien comercial, pero no puede ser ambos.

En una relación inclusiva, sin embargo, se permite que ocurrencias de la superentidad puedan ser tambien de varias subentidades al mismo tiempo. En el caso del ejemplo se permitiría que un empleado fuera técnico, comercial o ambos.

Cuando la especialización es exclusiva se indica en el diagrama con un semicírculo debajo de la punta del triángulo.



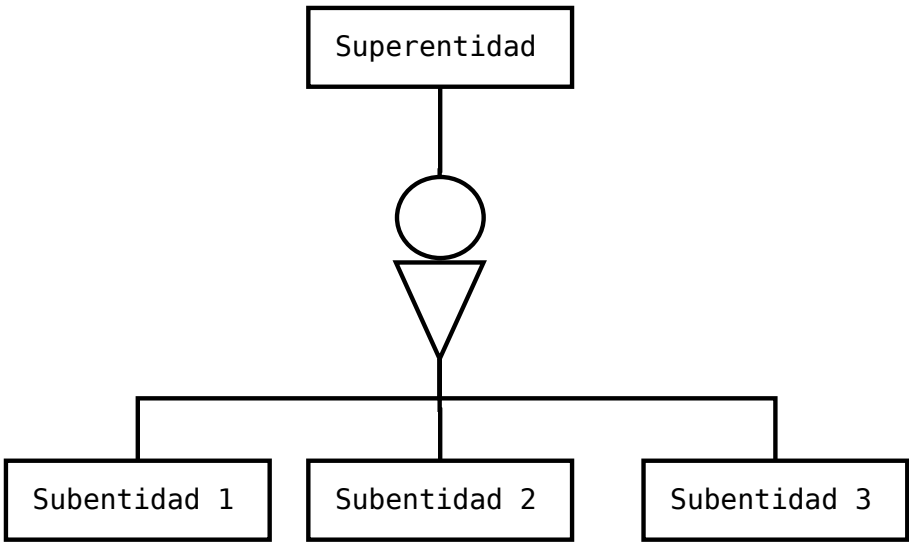
En el caso de que la especialización sea inclusive, no se incluye el semicírculo, es decir, el símbolo se dibuja con la forma original.

4.2 Especialización parcial o total

Se dice que una especialización es **total** cuando toda ocurrencia de la superentidad debe estar **obligatoriamente** relacionado con una ocurrecia de alguna de las subentidades, es decir, que no puede haber ocurrencias de la superentidad que no sean también ocurrencia de alguna de las subentidades. Por poner un ejemplo, un empleado debe ser un comercial o técnico. No se permiten empleados que no son uno de los dos.

En cambio, cuando esta obligación no existe, se dice que la relación es **parcial**. En este caso se permitiría la existencia de empleados que no sean también técnicos o empleados.

En un diagrama, la especialización total se indica con un círculo en la línea que conecta la superentidad con la base del triángulo.

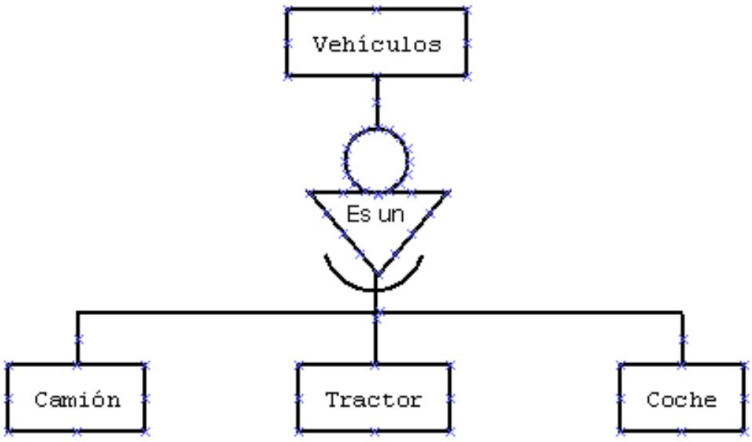


En el caso de que la especialización sea parcial, el círculo no se dibuja.

4.3 Conclusión

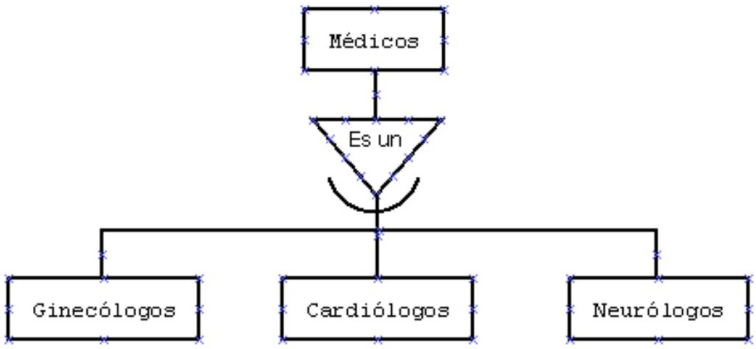
La especialización añade información extra sobre relaciones que no se puede representar por otros medios o, por lo menos, no de una forma clara y explícita.

Un ejemplo. el diagrama:



Indica que todo Vehiculo debe ser o bien un Camión, o bien un Tractor o bien un Coche pero sóno uno de ellos y además está obligado a serlo, esto es, no pueden existir Vehiculos que no sean también de una de las tres subentidades (la relación es exclusiva y total.

En cambio, el siguiente diagrama:



Nos muestra una relación en la cual un Médico puede ser Ginecóloco, Cardiólogo o Neurólogo o bien sólo Médico (la especialización es parcial) pero si está especializado sólo puede estarlo en una de las tres subentidades, es decir, no puede haber un médico que sea, a la vez, Cardiólogo y Ginecólogo, por ejemplo (la relación es exclusiva)

5 Resumen

En esta sección hemos detallado la utilidad el modelo entidad-relación desarrollado por Chen asi como sus principales elementos y la forma de obtener modelos que podamos utilizar posteriormente para definir bases de datos completas y robustas.

Tema 2
Ejemplo completo de modelo entidad-relación

Supongamos que Javier Pintor escribe libros de informática por afición. A Javier le gustaría poder almacenar la información de los libros que escribe y que publica por su cuenta cuando dispone de algo de dinero. Los libros se identifican mediante un código ISBN, de 20 caracteres de longitud y se quieren guardar los datos típicos de un ejemplar.

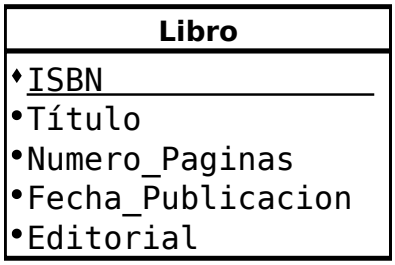
Lo primero es buscar las entidades. Podríamos pensar en Autor y en Libro. Sin embargo ¿cuántos autores vamos a guardar? Solamente tendría una ocurrencia, los datos de Javier. ¿Merece la pena guardarlo en este caso? Mi opinión es que no, pues es él mismo quien lo va a manejar y él ya se conoce a sí mismo, no necesita guardar información sobre él. ¿Se necesita la entidad Libro?, por supuesto que sí, es donde va a guardar la información de los libros que escribe.
El segundo paso es buscar las relaciones, en este caso solo tenemos una entidad y no parece que existan relaciones reflexivas para ella, luego en este caso no tenemos relaciones y por lo tanto no tenemos que buscar ni participación ni cardinalidad.
Buscamos los atributos de la entidad. Como el enunciado no especifica nada en especial, exceptuando el ISBN, colocaremos los más habituales para esta entidad, no obstante esta información habría que cotejarla con Javier por si quiere añadir algo más o eliminar alguno que no le interese. Algunos de estos atributos pueden ser:

- ISBN
- Título
- Número_Páginas
- Fecha_Publicación
- Editorial.

Por último especificamos el dominio para cada uno de ellos indicando su tamaño de forma aproximada (posteriormente habría que verificarlo con ejemplos reales para no quedarnos cortos en en número máximo de caracteres de cada campo) (todos los campos son obligatorios y sólo pueden tener un valor)

- ISBN: Cadena de caracteres de tamaño 20.
- Título: Cadena de caracteres de tamaño 50.
- Número_Páginas: Un número entero.
- Fecha_Publicación: Será un dato de tipo fecha.
- Editorial: Una cadena de caracteres de tamaño 30.

Ahora, a buscar claves. De todos los atributos ¿cuál piensas que puede identificar de forma única a cada uno de los libros? Para un solo autor como es el caso que nos ocupa podríamos pensar que el Título puede ser un buen candidato pues no parece lógico que publique dos libros con el mismo título, sin embargo ¿qué ocurre si revisa el libro tres años después y hace una nueva reedición? No podría poner el mismo título, tendría que cambiarlo, pues la clave principal no se pude repetir. Por ello consideramos que el ISBN es el campo clave principal, ya que es un conjunto de caracteres que te asignan cuando publicas un libro y es único para cada libro. Si el libro fuese reeditado obtendría un nuevo ISBN al publicarlo, con lo cual no habría problema.

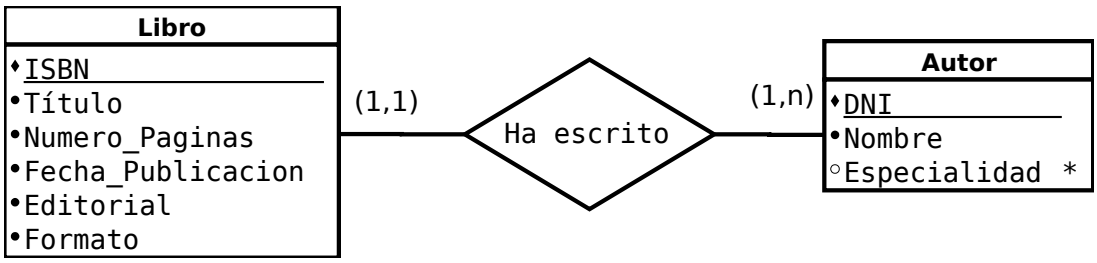


Javier Pintor ha estado comentando con sus compañeros del centro de secundaria donde imparte clases que se ha creado una base de datos con los libros que ha escrito. Otros profesores le comentan que también han escrito libros para las asignaturas que imparten y que los han publicado por su cuenta, unas veces en papel y otras en formato digital. Por ello Javier ha decidido modificar su base de datos para que aparezcan también los libros de sus compañeros y así poder publicar en su servidor Web los autores y los libros de cada uno.

Se ha modificado el contexto de nuestro problema y con ello también se verá modificado nuestro modelo E/R. Realizamos el mismo proceso que en el ejemplo anterior.
Primero buscamos las entidades. Ahora sí que nos interesará disponer de la entidad `Autor` ya que hay varios autores que se van a almacenar en esa entidad (va a tener varias ocurrencias) y por otro lado también necesitamos la entidad `Libro` para guardar la información de los libros publicados.
Buscamos las relaciones. Habrá una relación entre la entidad `Autor` y la entidad `Libro` y la podemos llamar *`Ha escrito`*. Buscamos la participación de esta relación, un `Autor` *`Ha escrito`* al menos uno (sino no sería autor) o varios `Libro`s, luego sería (1,n). Por otro lado un `Libro` siempre *`Ha sido escrito`* por 1 `Autor` y como mucho por 1 solo, pues el enunciado no plantea que un libro pueda haber sido escrito por varios compañeros a la vez (en caso de duda habría que preguntar al cliente o confirmarlo viendo las ocurrencias físicas de los `Libro`s), luego sería (1,1). Por lo tanto, la *cardinalidad* (el máximo de cada participación) sería 1:N.
Buscamos los atributos de cada entidad.
Algunos de los atributos de la entidad `Autor` pueden ser (con su dominio):

- DNI: Cadena de caracteres de tamaño 10.
- Nombre: Cadena de caracteres de tamaño 50.
- Especialidad: Cadena de caracteres de tamaño 50.

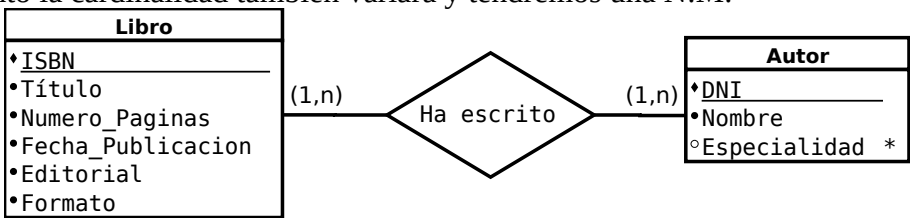
Todos los campos son obligatorios, excepto Especialidad. Éste puede tener, además, más de un valor.
Como clave de la entidad `Autor` utilizaremos el campo DNI que es único para cada una de las ocurrencias de la entidad.
Los atributos de la entidad `Libro` serán los mismos que teníamos antes pero hay que añadir uno nuevo para indicar el Formato del mismo (que seria una cadena de caracteres de tamaño 20, obligatorio y de un sólo valor)
La clave principal de la entidad `Libro` seguiría siendo ISBN.



Javier y sus compañeros se han dado cuenta de que con los libros que tienen publicados aún hay varias asignaturas para las que no tienen un libro de texto elaborado por alguno de ellos. Han pensado en crear esos libros para tener cubierta toda la gama de asignaturas que ellos imparten. Sin embargo, con la experiencia que tienen saben que el tiempo medio que tarda un autor en escribir un libro de texto es de un par de años, por ello han decidido dividirse el trabajo y en cada uno de los libros que faltan van a trabajar varios autores para terminarlos en el menor tiempo posible. ¿El modelo que tenemos del ejemplo anterior sirve para este nuevo contexto? ¿Si hubiese que cambiar algo que sería?

A la primera pregunta tenemos que responder que no sirve el mismo modelo del ejemplo anterior, pues una de las condiciones que presentaba ese modelo es que cada `Libro` era escrito por un solo `Autor`, sin embargo ahora se plantea la posibilidad de que un mismo `Libro` pueda estar escrito por varios `Autor`es, luego la participación y la cardinalidad van a cambiar.

A la segunda pregunta (ya parcialmente contestada en la anterior) hay que decir que cambia la participación entre la correspondencia `Libro`s que ***`Han sido escritos`*** por `Autor`, en este caso un `Libro` habrás sido escrito como mínimo por un `Autor`, pero como máximo por varios, pasaría a ser (1,n). Por tanto la cardinalidad también variará y tendremos una N:M.



Ampliaciones al ejercicio

1. Varios profesores de la zona han visto la página web del centro de Javier y las publicaciones incluidas y les ha gustado mucho la iniciativa. Algunos de estos profesores también tienen sus publicaciones y algunas de ellas se solapan con las existentes, es decir están elaboradas para las mismas asignaturas. Estos profesores han pedido permiso para incluir sus publicaciones en la misma base de datos y así poder ofrecer varias alternativas para la misma asignatura. A Javier y a los demás autores les ha parecido buena idea y se van a incluir también las obras de todos los profesores que lo soliciten. Además como el número de libros comienza a ser elevado y puede haber varios textos para una misma asignatura se desea reflejar de alguna forma que textos pueden servir para cada una de las asignaturas que se imparten en el centro.
2. Después de haber elaborado el modelo anterior, al revisarlo nos hemos dado cuenta de que hay libros como el de Inglés que puede servir para la asignatura de Inglés de Primero de DAM, la de primero de DAW y la de primero de ASIR. ¿El modelo propuesto en la imagen anterior contempla esta posibilidad? ¿Que habría que cambiar?
3. Todos los profesores que participan en las publicaciones recogidas en la base de datos se han sorprendido de la buena calidad de los materiales elaborados y han decidido que sería una buena idea enviar información a todos los centros de la comunidad autónoma por si desean emplearlos para sus clases. Por ello Javier va a modificar el modelo elaborado hasta ahora para reflejar en él aquellos centros que han seleccionado alguno de los libros de la base de datos para impartir alguna de sus asignaturas en un determinado año académico. Modifica el modelo de Javier para reflejar este nuevo contexto.

Tema 3
Modelo Lógico Relacional

Tabla de Contenidos

1.- Introducción

Utilizando el modelo entidad-relación, se ha obtenido una imagen conceptual de la información que se desea manejar con un sistema de información a partir de la realidad. Se ha obtenido, por tanto, un modelo conceptual, que, como ya se ha comentado anteriormente, es totalmente independiente del Sistema de Gestión de Bases de Datos (SGBD) que se quiera utilizar.

En este tema se dará el siguiente paso y se convertirá un modelo conceptual ya existente en un modelo lógico, ya adaptado a un tipo concreto de SGBD. En este caso el tipo de SGBD que se va a utilizar es el SGBD relacional, dado que es el tipo más común actualmente. Una vez obtenido el modelo lógico, se podrá obtener de él el modelo físico de un SGBD concreto que ya se podrá utilizar para su explotación.

2.- Estructura del Modelo Relacional

En esta sección se describirá el modelo relacional de bases de datos de forma genérica, esto es, en su forma ideal, independiente de las implementaciones existentes (Oracle, SQL Server, MySQL, DB2, etc.)

2.1.- Definición de base de datos relacional

Una base de datos relacional es una base de datos compuesta por un conjunto de relaciones, de ahí su nombre.

A diferencia del modelo entidad-relación ya considerado en temas anteriores una relación, en un sistema de bases de datos relacional, es un conjunto de datos con la misma estructura. Se podría

considerar equivalente, en términos apróximados, a lo que es una entidad en el modelo entidad-relación.

2.2.- Terminología Relacional

El modelo relacional se ocupa de la estructura, de la integridad y de la manipulación de los datos, cada una de estas partes tiene sus propios términos especiales. A continuación se exponen los términos relacionados con la estructura de datos.

- Una ***relación*** es un conjunto de ***tuplas*** con la misma estructura. Sería algo similar a una *tabla* con información.
- Una ***tupla*** es un conjunto de pares ***atributo-valor***. Se podría asemejar a una *fila* de una tabla (***relación***). El número de ***tuplas*** de una relación se denomina ***cardinalidad*** de esa ***relación*** y al número de ***atributos*** se le llama ***grado***.
- Un ***atributo*** es un dato simple con un nombre, que debe ser único dentro de una misma ***relación***, aunque pueden existir ***atributos*** con el mismo nombre en ***relaciones*** distintas. El valor de un ***atributo*** es el dato que hay actualmente almacenado en dicho ***atributo***. Se podría considerar como una *celda* o *casilla* de una fila de la tabla.
- Un ***dominio*** es un conjunto de valores. Todo ***atributo*** tiene asociado un ***dominio*** que indica cuales son los valores válidos que puede contener dicho ***atributo***. Se podría semejar a un *tipo de datos* en un lenguaje de programación.
- Por lo tanto, de manera formal, se define una ***relación*** R sobre un conjunto de ***dominios*** D_1, D_2, ...D_n como una estructura con dos partes:
  - ***Cabecera***. La ***cabecera*** de una ***relación*** es un conjunto de pares ***atributo-dominio***
    $\{(A_1,D_1), (A_2, D_2), ..., (A_n,D_n)\}$
    La ***cabecera*** indica **qué** ***atributos*** tiene la ***relación*** (como se llama cada uno) y el ***dominio*** del que cada ***atributo*** puede obtener sus valores. La ***cabecera*** de una ***relación*** es *estática*, esto es, no varía con el tiempo.
  - ***Cuerpo***. El ***cuerpo*** de una ***relación*** es un conjunto de ***tuplas*** con pares ***atributo-valor*** de la forma:
    $\{(A_1,V_1), (A_2, V_2), ..., (A_n, V_n)\}$
    El ***cuerpo*** contiene la información almacenada en la ***relación*** (los información almacenada en la tabla). El número de ***tuplas*** es variable a lo largo del tiempo, según se añada o elimine información de la ***relación***.

Ejemplo:

PROVEEDOR

CODIGO	NOMBRE	CIUDAD
1	Salazar	Londres
2	Jaimes	París
3	Bernal	París
4	Corona	Londres
5	Aldana	Atenas

Esta tabla es la relación PROVEEDOR.

Las tuplas son cada una de estas filas

1,Salazar,Londres

2,Jaimes,París

3,Bernal,París

4,Corona,Londres

5,Aldana,Atenas

La cardinalidad es 5 y el grado es 3.

El dominio NATURALES es el conjunto de los números naturales, el dominio CIUDADES es el conjunto de nombres de ciudades de los proveedores y el dominio TEXTO50 es el conjunto de todas las posibles combinaciones de letras de hasta 50 caracteres de longitud.

La cabecera sería

{(CODIGO,NATURALES),(NOMBRE,TEXTO50),(CIUDAD,CIUDADES)}

Y el cuerpo

{(CODIGO,1),(NOMBRE,Salazar),(CIUDAD,Londres)}

{(CODIGO,2),(NOMBRE,Jaimes),(CIUDAD,París)}

{(CODIGO,3),(NOMBRE,Bernal),(CIUDAD,París)}

{(CODIGO,4),(NOMBRE,Corona),(CIUDAD,Londres)}

{(CODIGO,5),(NOMBRE,Aldana),(CIUDAD,Atenas)}

Se podrían establecer de forma informal las siguientes equivalencias:

Término relacional formal	Equivalentes informales
Relación	Tabla
Tupla	Fila o registro
Cardinalidad	Número de filas
Atributo	Columna o campo
Grado	Número de columnas
Dominio	Conjunto de valores permitidos.

2.3.- Condiciones de una relación

Según estableció el creador del sistema relacional, Codd, **TODAS** las relaciones deben cumplir las siguientes condiciones:

1) Puede contener un solo tipo de registro, cada uno tiene un número fijo de campos con su correspondiente nombre. La base de datos estará formada por muchas tablas, de modo que cada tipo de registro será una tabla diferente.
2) Dentro de una relación no pueden existir tuplas repetidas (que tienen exactamente los mismos datos). Esta propiedad se desprende del hecho de que el cuerpo de una relación es un conjunto matemático y en un conjunto no puede repetirse un elemento.
3) Dentro de una relación no pueden existir atributos repetidos. Esta propiedad se desprende del hecho de que la cabecera de una relación se define también como un conjunto matemático.
4) El orden de las tuplas en una tabla no está determinado. También es un conjunto.
5) El orden de los atributos en una tabla no está determinado. Por la misma razón.

2.4.- Reglas de integridad relacional

Cualquier base de datos es un reflejo de la realidad. Ahora bien, algunas configuraciones de valores sencillamente no tienen sentido, en cuanto a que no reflejan ningún posible estado del mundo real. Por ejemplo, que en un pedido aparezca una cantidad negativa. Por lo tanto, es necesario ampliar la definición de la base de datos, a fin de incluir ciertas reglas de integridad, cuya función es informar al SGBD de ciertas restricciones que se dan en el mundo real.

Por ejemplo la lista de las reglas para una base de datos de proveedores, piezas y pedidos podría ser:

- Las cantidades son múltiplos de 100
- Las cantidades no pueden ser negativas
- Los números de los proveedores deben tener la forma Snnn
- Los números de las piezas deben tener la forma Pnnn

La mayor parte de las reglas de integridad son específicas, en cuanto a que se aplican a bases de datos concretas. Los ejemplos anteriores son específicos en ese sentido.

El modelo relacional, en cambio incluye dos reglas generales de integridad, que son generales en el sentido de que se aplican no solo a una base de datos especifica como la de proveedores, piezas y pedidos, sino a todas las bases de datos que ya existen o que puedan llegar a existir. Estas dos reglas generales de integridad hacen referencia, respectivamente, a las claves primarias y a las claves ajenas.

Por último, señalar que todas las reglas de integridad, tanto las especificas con respecto una base de datos, como las dos reglas generales del modelo relacional, se aplican a las relaciones base.

Una *Relación Base* corresponde a una relación autónoma, con nombre, que se almacena en la base de datos. Existen otro tipo de relaciones, cuya naturaleza es más efímera, como por ejemplo el resultado de una consulta. Ya se hablará de este asunto posteriormente con más detalle.

Las reglas generales de integridad son:

- Regla de Integridad de la Entidad (claves primarias)
- Regla de Integridad Referencial. (claves ajenas)

Para poder definir correctamente las reglas, primero debemos definir el concepto de clave primaria y ajena (o foránea)

2.4.1.- Clave primaria

Se dice que un *K* es una *Clave Candidata* o clave posible de una *relación* *R*, si cada tupla de *R* es identificada de forma unívoca por el valor de *K*. Una relación puede tener varias claves candidatas.

Una *Clave Primaria* es cualquier elección de una *clave candidata*; al resto de claves candidatas le llamamos *Claves Alternativas*. En la práctica se elige como clave primaria la que se considera de más importancia.

Las claves primarias tienen mucha importancia, porque constituyen el mecanismo de direccionamiento a nivel de tuplas básico en un sistema relacional.

2.4.2.- Clave Ajena

En términos informales se puede definir una Clave Ajena en una relación R2 como un atributo de dicha relación cuyos valores deben concordar con los de las claves primarias de alguna relación R1 (donde R1 y R2 no tienen que ser necesariamente distintos).

La relación que contiene la clave ajena se conoce como Relación Referencial, y la relación que contiene a la clave primaria correspondiente se denomina relación referida o Relación Objetivo. Podemos representar esta situación con un Diagrama Referencial.

Ejemplo:

Supongamos que tenemos las siguientes relaciones:

PROVEEDOR

S	NOMBRE	CIUDAD
1	Salazar	Londres
2	Jaimes	París
3	Bernal	París
4	Corona	Londres
5	Aldana	Atenas

PIEZA

P	NOMBREP	PRECIO
1	TORNILLO	10
2	TUERCA	20
3	ARANDELA	15
4	MARTILLO	17

PEDIDO

S	P	CANTIDAD
1	1	200
1	2	300
2	4	500
3	1	700
3	2	1200

S y P de la relación PEDIDO son claves ajenas.

S (en PEDIDO) representa una referencia a la tupla donde se encuentra el valor correspondiente de la clave primaria (tupla objetivo). PEDIDO es la relación referencial y PROVEEDOR es la relación objetivo.

P representa una referencia a la tupla donde se encuentra el valor correspondiente de la clave primaria (tupla objetivo). PEDIDO es la relación referencial y PIEZA es la relación objetivo.

El Diagrama Referencial es:

Hay que tener en cuenta las siguientes consideraciones:

a) Una clave ajena dada y la clave primaria correspondiente deben definirse sobre el mismo dominio.

b) La clave ajena no necesita ser un componente de la clave primaria de la relación que la contiene (son independientes).

c) Las Relaciones R1 y R2 en la definición de clave ajena no son necesariamente distintas. Es decir, una relación podría incluir una clave ajena cuyos valores deben concordar con los valores de la clave primaria de esa misma relación. (relación reflexiva o auto referencial)

Ejemplo:

En la siguiente relación

EMP (NUMEMP,...,SALARIO,...,NUMEMP_GER,...)

Aquí, el atributo NUMEMP_GER representa el número de empleado del *gerente* del empleado identificado mediante NUMEMP. NUMEMP es la clave primaria, y NUMEMP_GER es una clave ajena que hace referencia a ella. Este tipo de relaciones se denominan auto referenciales.

d) Las claves ajenas, a diferencia de las claves primarias, pueden aceptar nulos en ocasiones (cuando el pertenecer a una relación no es obligatorio).

2.4.3.- Regla de Integridad de las Entidades

Ningún componente de clave primaria de una relación base puede aceptar nulos.

2.4.4.- Regla de Integridad Referencial

La base de datos no debe contener valores de clave ajena sin concordancia, esto es, todo valor de clave ajena se debe corresponder con un valor de clave primaria existente.

2.4.5.- Reglas adicionales para claves ajenas

Cualquier estado de la base de datos que no satisfaga la regla de integridad referencial será incorrecto por definición, pero ¿cómo pueden evitarse tales estados incorrectos?.

Una posibilidad es que el sistema rechazara cualquier operación que en caso de ejecutarse, produciría un estado ilegal. Pero en muchos casos una alternativa preferible sería que el sistema aceptara la operación, pero realizara (en caso necesario) ciertas operaciones de compensación con objeto de garantizar el estado legal como resultado final.

En consecuencia, para cualquier base de datos, el diseñador de esta deberá poder especificar cuales operaciones han de rechazarse y cuales han de aceptarse, y en el caso de estas últimas, cuáles operaciones de compensación (si acaso) deberá realizar el sistema.

Para cada clave ajena es necesario responder las siguientes tres preguntas:

1. ¿Puede aceptar nulos?.

La respuesta dependerá de la política vigente en la porción del mundo real que se esté modelando. En el caso de un modelo conceptual existente de tipo entidad-relación, dependería de si la cardinalidad mínima en la relación vale cero o no. Si vale cero se podrán aceptar nulos. Si vale 1 no se podrán aceptar nulos.

2. ¿Qué deberá suceder si hay un intento de eliminar el objetivo de una referencia de clave ajena?

Entonces existen tres posibilidades:

a) **RESTRINGIDA. (RESTRICTED).** La eliminación sólo se puede realizar si no hay ninguna clave ajena con el mismo valor que la clave primaria de la tupla que se va a eliminar. Dicho de otra forma, no se puede eliminar una tupla si hay claves ajenas que "apuntan" a ella.

b) **SE PROPAGA (CASCADES).** La operación de eliminación se propaga en cascada, eliminando también las tuplas con claves ajenas que tienen el mismo valor que la clave primaria de la tupla que se elimina. Dicho de otra forma, si se elimina una tupla, se eliminan todas las tuplas que "apuntan" a ella.

c) **ANULA (NULLIFIES).** Los valores de las claves ajenas que tengan el mismo valor que la clave primaria de la tupla que se elimina se cambian al valor NULL (nulo). Este caso no es aplicable si la clave ajena no acepta nulos.

3. ¿Qué deberá suceder si hay un intento de modificar el objetivo de una referencia de clave ajena?

Entonces existen tres posibilidades (al igual que en la eliminación):

a) **RESTRINGIDA. (RESTRICTED).** No se permite la modificación si hay claves ajenas que coinciden con la que se va a modificar.

b) **SE PROPAGA (CASCADES).** Se modifica tanto la clave primaria que se está cambiando expresamente como la clave ajena que tiene el mismo valor.

c) **ANULA (NULLIFIES).** Se modifica el valor de la clave primaria y el de la clave ajena se cambia a NULL. Este caso no es aplicable si la clave ajena no acepta nulos.

Concretando, para cada clave ajena del diseño, el diseñador de la base de datos deberá especificar:

1. El atributo o combinación de atributos que constituyen esa clave ajena.
2. La relación objetivo a la cual hace referencia esa clave ajena.
3. La respuesta a las tres cuestiones anteriores. Es decir, la respuesta a las reglas para claves ajena:
   - Regla de Nulos
   - Regla de Eliminación
   - Regla de Modificación.

3.- Representación del modelo relacional

El modelo relacional se representa habitualmente de dos formas: textual o gráfica.

3.1.- Representación textual del modelo relacional

En la representación textual se da una lista con las relaciones y sus atributos.

Las claves primarias se subrayam y las claves ajenas se unen con flechas a las claves primarias con las que enlazan.

El ejemplo anterior quedaría

PROVEEDOR(S,NOMBRE,CIUDAD)

PIEZA(P,NOMBREP,PRECIO)

PEDIDO(S,P,CANTIDAD)

Además habría una flecha desde el atributo S de PEDIDO hacia el atributo S de PROVEEDOR y desde el atributo P, también de PEDIDO hacia el atributo P de PIEZA.

3.2.- Representación gráfica del modelo relacional

La representación gráfica utiliza cajas para representar las relaciones. En ellas aparece el nombre de la relación en la parte superior y una línea lo separa del resto de la caja, que contiene una lista con los atributos de la relación. En esta lista las claves primarias aparecerán subrayadas.

Las relaciones entre claves primarias y claves ajenas se muestran utilizando flechas que van desde la clave ajena hasta la clave primaria a la que apuntan.

El mismo ejemplo anterior representado gráficamente sería:

Tema 4
Transformación del modelo conceptual al modelo lógico

Tabla de Contenidos

1.- Introducción

Utilizando el modelo entidad-relación, se ha obtenido, a partir de la realidad, una imagen conceptual de la información que va a manipular un sistema de información. Se ha obtenido, por tanto, un modelo conceptual que, como ya se ha comentado anteriormente, es totalmente independiente del Sistema de Gestión de Bases de Datos (SGBD) que se quiera utilizar.

También se han descrito ya la estructura y operaciones que se pueden realizar con el modelo relacional. Queda, por tanto, el aprender como convertir un modelo entidad-relación ya diseñado en el modelo relacional equivalente. En esta unidad se describe el proceso a seguir para realizar dicha conversión.

2.- Conversión del modelo conceptual Entidad-Relación al modelo lógico Relacional

La conversión del modelo conceptual al modelo lógico se realiza en una serie de pasos a través de los cuales se van convirtiendo los distintos elementos de un modelo a sus equivalentes en el otro. Hay que hacer notar, sin embargo, que hay elementos que no se pueden transformar directamente y que hay que documentar en otra forma, por ejemplo con un documento anexo.

2.1.- Conversión de entidades y atributos

A la hora de convertir las entidades hay que distinguir entre el caso de que la entidad a convertir sea una entidad débil o no.

2.1.1.- Conversión de entidades no débiles

Por cada entidad **no débil** *E* en el Modelo Entidad Relación (MER) se definirá una nueva relación *R* en el Modelo Relacional (MR), a ser posible con el mismo nombre de E.

En *R* se incluirán todos los atributos **simples** de *E* (no los atributos multivaluados o compuestos).

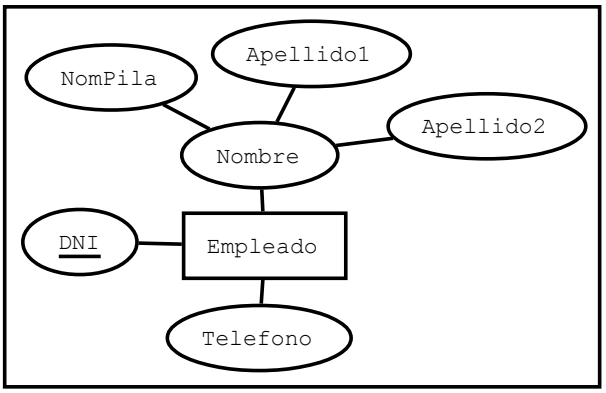
También se incluirán en *R* todos los atributos simples que componen los atributos compuestos de *E* como atributos simples de *R*.

En los atributos *opcionales* se deberá permitir la aparición de valores nulos.

La clave primaria de *R* estará formada por los atributos clave de *E*.

Ejemplo:

Supongamos la entidad



Dará lugar a la siguiente relación:

`Empleado(DNI,NomPila,Apellido1,Apellido2,Telefono)`

2.1.2.- Conversión de entidades débiles

Por cada entidad **débil** D en el MER, vinculada con otra entidad fuerte *E*, se definirá una nueva relación *R* en el MR, a ser posible con el mismo nombre de *D*.

En *R* se incluirán todos los atributos simples de *D* (no los multivaluados ni compuestos).

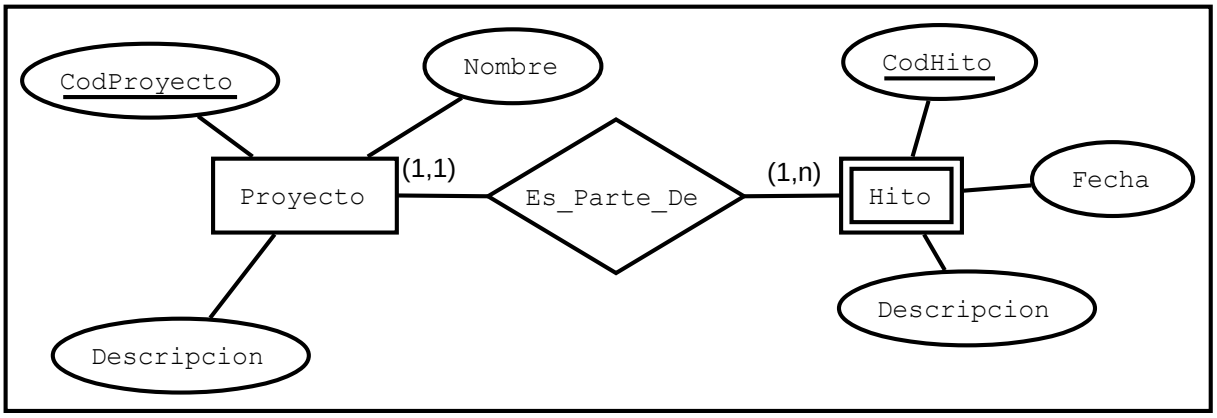
También se incluirán en *R* todos los atributos simples que componen los atributos compuestos de *D* como atributos simples de *R*.

También se añadirán a *R* los atributos que forman la clave primaria de *E*.

La clave primaria de *R* estará formada por **todos** los atributos de *R*..

Ejemplo:

El modelo



Se transformará en

`Proyecto(CodProyecto,Nombre,Descripcion)`

`Hito(CodProyecto,CodHito,Fecha,Descripcion)`

2.2.- Conversión de relaciones

Para la conversión de las relaciones hay que distinguir entre varios casos, dependiendo de varios factores:

- El grado de la relación. Se distingue entre relaciones binarias, incluidas las reflexivas, y las ternarias, cuaternarias, etc.
- La cardinalidad de la relación, en el caso de relaciones binarias. Se distingue entre relaciones 1:1, 1:N (ó N:1) y N:M.
- La presencia de atributos en la relación.

2.2.1.- Relaciones binarias sin atributos

Para el caso de relaciones binarias sin atributos distinguimos según la cardinalidad de la relación.

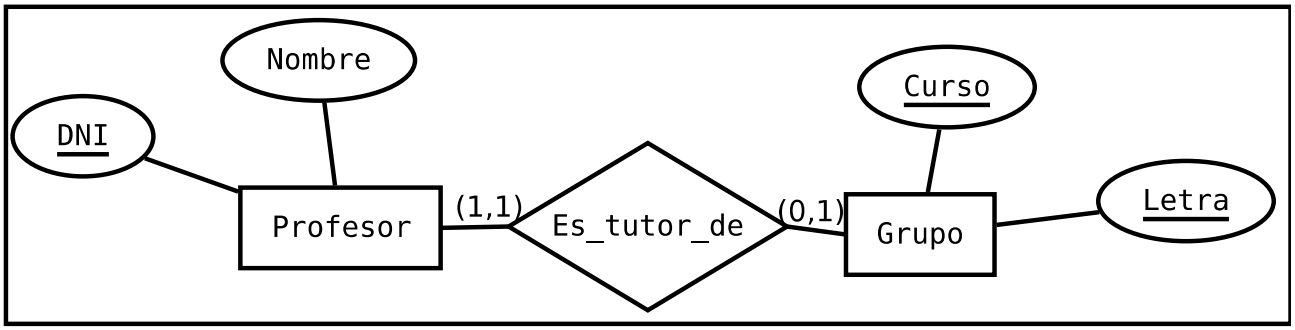
2.2.1.1 Relaciones 1:1

Por cada relación binaria sin atributos con cardinalidad 1:1 entre entidades **no débiles** se añade la clave primaria de una de ellas en la otra como clave ajena.

Si una de las dos entidades tiene una participación de (0,1) y la clave ajena se ha colocado en la relación correspondiente a la otra entidad, entonces habrá que permitir que acepte valores nulos a fin de satisfacer la participación requerida.

También hay que establecer una restricción para que el valor de la clave ajena sea único.

Ejemplo:



Se transformaría en

`Profesor(DNI,Nombre)`

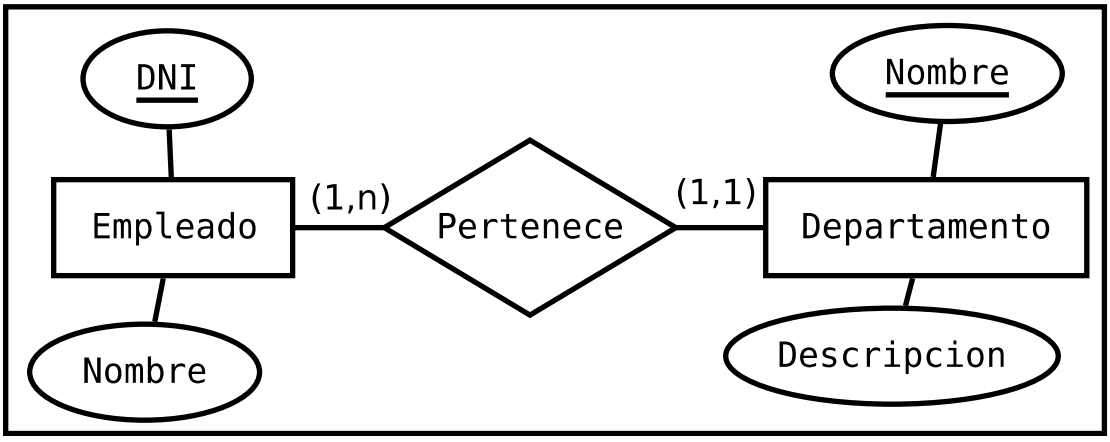
`Grupo(Curso,Letra,DNITutor)`

`DNITutor` de la relación `Grupo` debe ser único, esto es, no debe permitir valores duplicados.

2.2.1.2 Relaciones 1:N

Por cada relación binaria sin atributos con cardinalidad 1:N entre entidades **no débiles** se añade la clave primaria de la relación correspondiente al lado 1 de la relación como clave ajena en la relación correspondiente al lado N de la relación. Si en el lado 1, la cardinalidad es (0,1), la clave ajena deberá poder valer NULL.

Ejemplo:



Se transformaría en:

`Empleado(DNI,Nombre,NombreDpto)`

`Departamento(Nombre,Descripcion)`

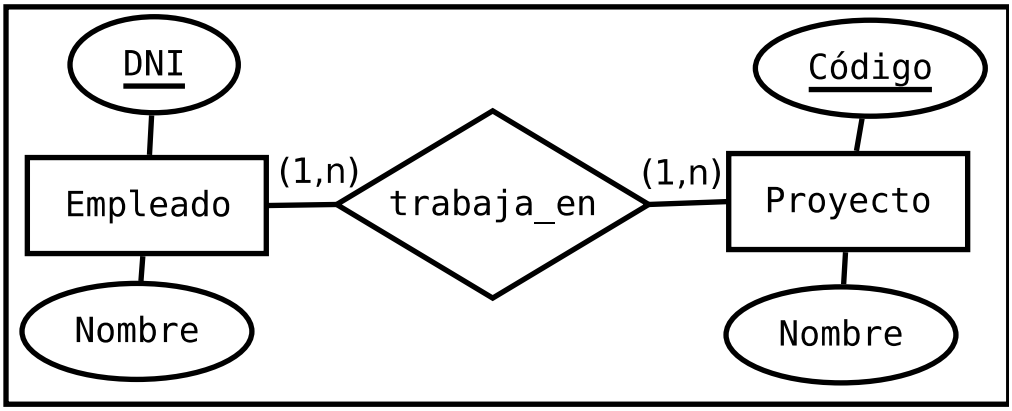
2.2.1.3 Relaciones N:M

Por cada relación binaria sin atributos con cardinalidad N:M entre entidades **no débiles** *E* y *F* se crea una nueva relación *R*, a ser posible llamada como la relación del MER.

R tendrá como atributos todos los que forman la clave primaria de *E* y todos los que forman la clave primaria de *F*.

La clave primaria de *R* estará formada por los atributos que forman la clave primaria de *E* más los que forman la clave primaria de *F*.

Ejemplo:



Se transformaría en:

`Empleado(DNI,Nombre)`

`Proyecto(Código,Nombre)`

`Trabaja_en(DNI,Codigo)`

2.2.2.- Relaciones binarias con atributos

Se tratan como una relación n-aria con atributos. Ver más adelante.

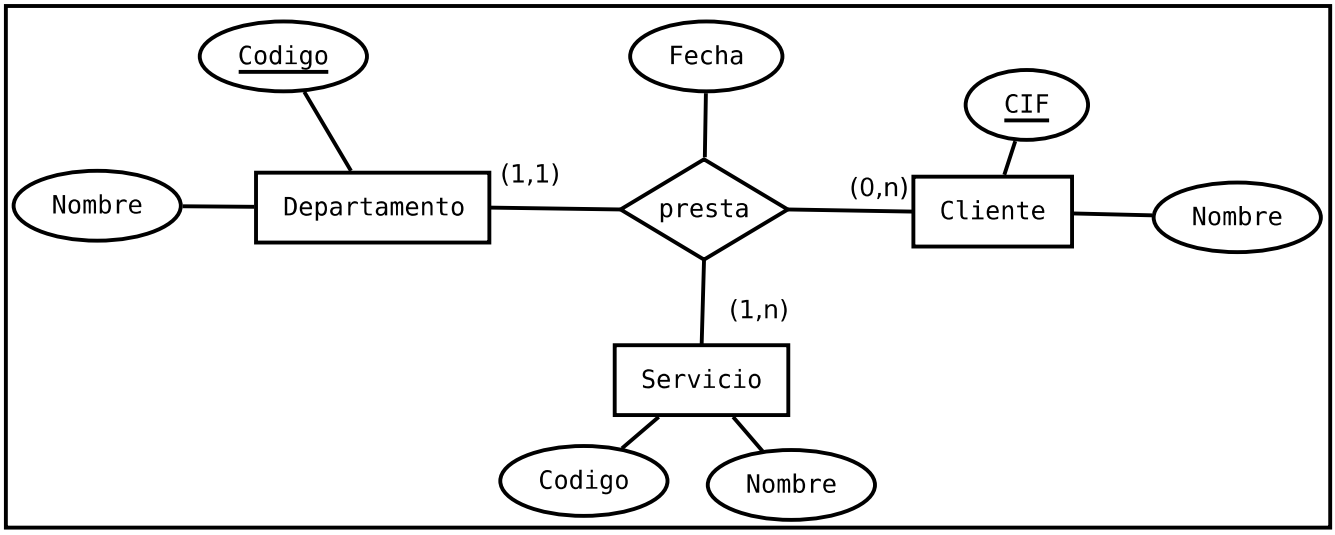
2.2.3.- Relaciones n-arias

Por cada relación n-aria *RN* con o sin atributos entre las entidades E_1,E_2,...,E_n en el MER, se crea una nueva relación *R* en el MR.

R contendrá como atributos los que forman las claves de E_1,E_2,...,E_n. Además contendrá como atributos los de *RN*.

La clave primaria de *RN* estará formada por los atributos que forman la clave primaria de E_1, más los que forman la clave de E_2,etc.

Ejemplo:



Se transformaria en:

`Departamento(Codigo,Nombre)`

`Cliente(CIF,Nombre)`

`Servicio(Codigo,Nombre)`

`Presta(CodDpto,CIF,CodSvc,fecha)`

Hay que hacer notar que esta forma de construir una relación se puede utilizar para representar todo tipo de relaciones binarias y n-arias, con y sin atributos, aunque generalmente se considera más eficiente la forma especial descrita en los apartados anteriores.

2.3.- Elementos no convertibles

Hay elementos del modelo entidad-relación que no tienen expresión en el modelo relacional, entre ellos los atributos compuestos, los multivalor y las relaciones de generalización/especialización.

2.3.1.- Atributos compuestos

Como ya hemos descrito en un apartado anterior, los atributos compuestos se transforman en una serie de atributos simples que se añaden en lugar del atributo compuesto.

2.3.2.- Atributos multivalor

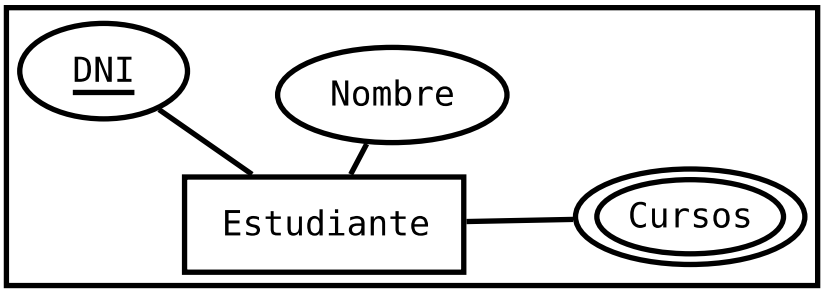
Los atributos multivalor son aquellos que en lugar de un valor pueden tener una lista más o menos larga de valores del mismo tipo. Como el modelo relacional no soporta este tipo de atributos, se debe encontrar una forma de representarlos.

Por cada atributo multivalor *M*, perteneciente a una entidad *E* en el MER se crea una relación *R* en el MR.

Los atributos de *R* serán los que formen la clave primaria de *E* más el correspondiente al atributo multivaluado.

La clave primaria de *R* estará formada por la clave de *E* mas el atributo multivaluado.

Ejemplo:



Se representaría como:

`Estudiante(DNI,Nombre)`

`Cursos(DNI,Curso)`

2.3.3.- Jerarquías de generalización/especialización

Las jerarquías de generalización/especialización (relaciones IS A) no tienen conversión directa en el modelo relacional, por lo que hay que "simularlas".

Hay varias estrategias para simular la generalización/especialización utilizando un modelo relacional. En este documento sólo se va a tratar una de ellas que es lo bastante general como para soportar todos los casos.

En un MER puede existir una entidad general o padre *E* y múltiples entidades especializadas o hijas H_1, H_2, ..., H_n.

Se definirá una relación *R* para la entidad *E* y otras n (R_1, R_2, ..., R_n) para las entidades H_1, H_2, ..., H_n.

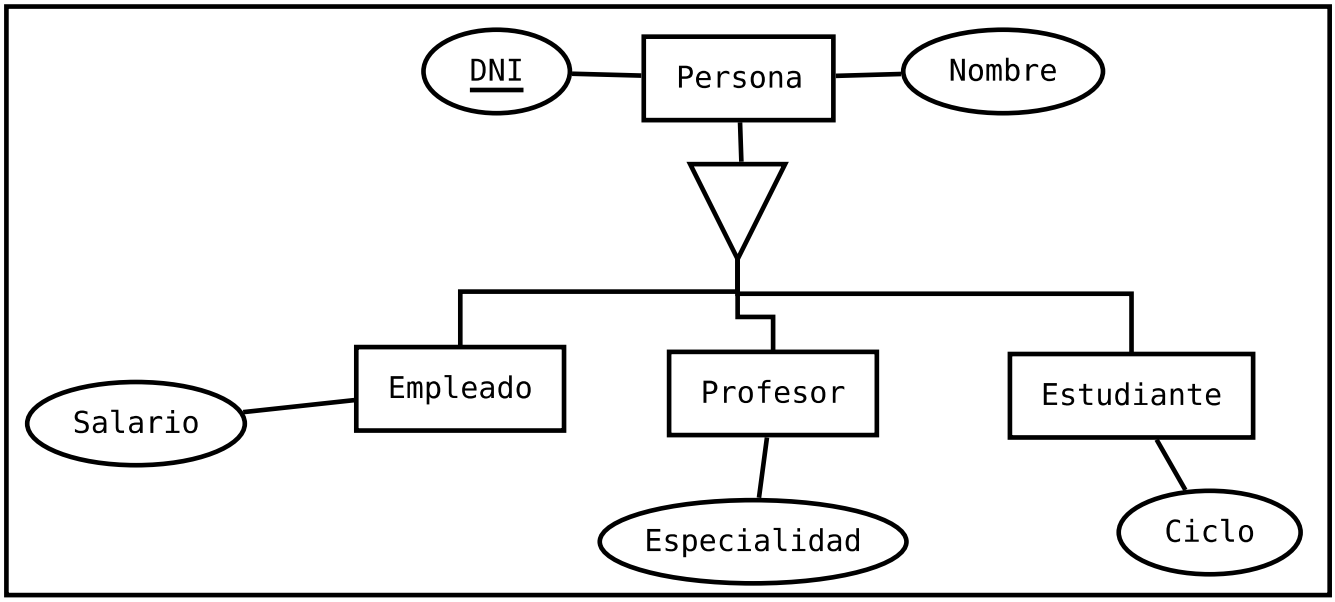
Los atributos de *R* serán los de *E*, los de R_1, los de H_1, los de R_2 los de H_2, etc. Además R_1, R_2, ..., R_n tendrán como atributos los que forman la clave primaria de *E*.

La clave primaria de *R*, R_1, R_2, ..., R_n será la clave primaria de *E*.

Este tipo de conversión soporta especialización exclusiva o inclusiva. Si es exclusiva, para cada entidad especializada habrá dos tuplas, una en la relación general y otra en la correspondiente a su especialización. Si es inclusiva, la tupla en la relación general seguirá existiendo pero podrá haber una en más de una relación perteneciente a las entidades especializadas.

También se soporta la especialización parcial o total. Si es parcial puede haber tuplas en la relación general que no se correspondan con ninguna tupla en ninguna de las especializaciones. Si es total, sin embargo se deberá obligar a que haya en alguna especialización una tupla que coincida con cualquier tupla de la relación general.

Ejemplo:



Puede representarse en un modelo relacional como:

`Persona(DNI,Nombre)`

`Empleado(DNI,Salario)`

`Profesor(DNI,Especialidad)`

`Estudiante(DNI,Ciclo)`

Como la relación es inclusiva, podrá haber tuplas con el mismo `DNI` en `Empleado`, `Profesor` y `Estudiante`. Si fuera exclusiva (con el arco debajo) sólo podría haber una tupla con un `DNI` dado o bien en `Empleado`, o bien en `Profesor` o bien en `Estudiante`, pero no en más de una.

Como la relación es parcial, puede haber tuplas con un `DNI` dado en `Persona` y que no haya ninguna tupla con ese `DNI` ni en `Empleado` ni en `Profesor` ni en `Estudiante`. Esto significa que hay una persona que no es ninguna de estas tres cosas. Si la relación fuera total (con el círculo por encima), si hay una tupla con un `DNI` dado en `Persona` debería haber una con el mismo `DNI` o en `Empleado` o en `Profesor` o en `Estudiante` (o en más de una, dado que es inclusiva).

3.- Resumen

En este documento se ha tratado la forma de convertir un modelo entidad relación cualquiera a un modelo relacional equivalente.

Se han documentado una serie de pasos para convertir entidades, atributos y relaciones, con especial atención a los distintos tipos de relaciones.

Tema 5
Normalización del Modelo Lógico Relacional

Tabla de Contenidos

1.- Introducción

El modelo relacional describe los datos a través de relaciones que están enlazadas entre si por valores idénticos de campos clave: claves primarias y claves ajenas.

Sin embargo, una base de datos expresada en modelo relacional no está libre de inconsistencias de datos. Existe un proceso sistemático de eliminación de inconsistencias en bases de datos relacionales denominado *normalización*.

2.- Normalización

Supongamos que se tiene una base de datos relacional sencilla que consta de una única relación, que se muestra a continuación:

R

CodProveedor	**Ciudad**	**País**	**CodPieza**	**Cantidad**
1	Londres	Inglaterra	1	200
1	Londres	Inglaterra	2	1300
1	Londres	Inglaterra	4	345
2	Madrid	España	1	24
2	Madrid	España	6	988
3	Paris	Francia	2	200
4	Londres	Inglaterra	3	34

Se puede comprobar fácilmente que esta relación no está demasiado bien diseñada y que puede dar problemas de inconsistencia de datos por la abundancia de redundancias que aparecen. Por ejemplo, se repite muchas veces la información de que el proveedor 1 es de Londres.
Esta redundancia puede provocar graves problemas en la manipulación de los datos. Específicamente:

- Problemas de actualización. Si, por ejemplo, el proveedor 1 cambia de ciudad habría que actualizar TODAS las tuplas en las que aparece dicho proveedor. Si el operador se equivoca y no modifica alguna de ellas, se podrían producir dos respuestas a la pregunta ¿En qué ciudad reside el proveedor?
- Problemas de inserción. Como los datos de un pedido y de un proveedor van entrelazados, no se podría añadir un proveedor hasta que no haga un pedido.
- Problemas de eliminación. Si eliminamos un proveedor que sólo tenga un pedido, también se elimina el pedido.

Para evitar este tipo de problemas se realiza un proceso denominado *normalización*. La normalización se realiza asegurando que la base de datos está en lo que se llama una *forma normal*. Una forma normal es una serie de condiciones que deben cumplir **todas** las relaciones de la base de datos. Estas condiciones aseguran que se reducen las redundancias al mínimo posible.
Antes de definir las formas normales se deben definir dos conceptos fundamentales que se utilizan en las condiciones de estas formas normales: Dependencia funcional simple y dependencia funcional plena.

2.1.- Dependencia funcional simple

La dependencia funcional simple, o dependencia funcional a secas, entre dos atributos se define de la siguiente forma:

Dados dos atributos X e Y de una relación R, se dice que Y depende funcionalmente de X si, y sólo si, en cualquier instante cada valor de X tiene asociado, en R, un único valor de Y. Se suele denotar como X->Y

Dicho de una forma informal se dice que un atributo Y depende de otro X si el valor de Y está determinado por el valor de X pero no al revés.

2.2.- Dependencia funcional plena

La dependencia funcional plena se define de la siguiente forma:

Dado un conjunto de atributos X (X_1, X_2, ..., X_n) y otro atributo Y, se dice que Y tiene dependencia funcional plena de X si Y depende de X y no se puede encontrar ningún subconjunto de X del que dependa Y.

Dicho informalmente, si el valor de un atributo depende de una clave compuesta, debe depender de **toda** la clave y no de parte de ella.

2.3.- Primera Forma Normal

Una relación está en primera forma normal (1FN) si se cumple la condición de que todos los dominios no contienen valores que sean conjuntos sino que son valores escalares.

Dicho en otras palabras, una relación está en 1FN si no hay ningún atributo cuyo valor sea un conjunto de valores.

Es una forma normal que, por definición, toda tabla que cumpla las condiciones del modelo relacional debe cumplir, por lo que usualmente todas las bases de datos deberían estar en esta forma normal.

En el ejemplo anterior, la relación ya está en 1FN porque no hay atributos cuyos valores sean conjuntos.

Si se encontraran relaciones con atributos multivalor, habría que crear una nueva tabla por cada uno de los atributos multivalor junto con la clave primaria de la tabla en la que estaba el atributo. La clave primaria de la nueva tabla serán ambos atributos.

2.4.- Segunda Forma Normal

Una relación está en Segunda Forma Normal (2FN) si, además de estar en 1FN, cualquier atributo que no forme parte de la clave primaria tiene dependencia funcional plena con la clave primaria.

Si la clave primaria de la relación no es compuesta, la relación está automáticamente en 2FN.

Dicho informalmente, esta forma normal intenta hacer que todos los atributos de una relación dependan de **toda** la clave principal. Si sólo dependen de parte, esto quiere decir que hay un problema.

Para solucionar uel problema que presenta una relación que no cumple la 2FN se toman los atributos que sólo dependen de parte de la clave y se mueven a otra nueva relación que sólo contendrá los atributos parcialmente dependientes y la parte de la clave principal de la que dependen. Esta parte será la clave principal de la nueva relación. Los campos de la clave principal permanecen en la relación original también.

En el ejemplo, la clave es (CodProveedor, CodPieza).

Si se examinan los otros se puede comprobar lo siguiente:

Ciudad depende sólo de CodProveedor, no de CodPieza. Si se examinan los datos se verá que hay valores distintos de Ciudad para el mismo valor de CodPieza (por ejemplo, para CodPieza con valor 1 hay dos valores distintos de Ciudad: Londres y Madrid).

Pais está en el mismo caso que Ciudad.

Cantidad *si* depende de la clave completa.

Por lo tanto, dividimos la relación R en dos: R1 y R2

`R1(CodProveedor,CodPieza,Cantidad)`

`R2(CodProveedor,Ciudad,Pais)`

Si se vuelven a analizar ambas relaciones se comprobará que están ambas en 2FN, por lo que la base de datos está en 2FN.

2.5.- Tercera Forma Normal

Una relación está en Tercera Forma Normal (3FN) si está en 2FN y además ninguno de los atributos que no forman parte de la clave primaria tiene dependencias transitivas respecto de la clave de la relación.

Informalmente quiere decir que hay atributos que dependen de atributos que no forman parte de la clave primaria.

Para solucionar el problema se mueve el atributo dependiente y el intermedio a otra nueva relación en la que el intermedio será clave primaria. El atributo intermedio permanece en la relación original.

En el caso del ejemplo, está claro que la relación R1 está en 3FN.

¿Y la relación R2?

¿Ciudad depende de proveedor? Si.

¿Pais depende de proveedor? No directamente. Pais depende de Ciudad, que a su vez depende de Proveedor.

Por lo tanto la relación R2 no cumple la 3FN.

Habría que dividir la relación R2 en dos relaciones: R21 y R22. La base de datos quedaría:

`R1(CodProveedor,CodPieza,Cantidad)`

`R21(CodProveedor,Ciudad)`

`R22(Ciudad,Pais)`

Y los datos

R1

CodProveedor	**CodPieza**	**Cantidad**
1	1	200
1	2	1300
1	4	345
2	1	24
2	6	988
3	2	200
4	3	34

R21

CodProveedor	**Ciudad**
1	Londres
2	Madrid
3	Paris
4	Londres

R22

Ciudad	**País**
Londres	Inglaterra
Madrid	España
Paris	Francia

Como se puede comprobar se han eliminado completamente las redundancias.

3.- Resumen

Como se ha intentado describir, el proceso de normalización es un proceso fácil y mecánico que permite mejorar un modelo relacional eliminando la mayoría de las redundancias para dejar sólo las imprescindibles para relacionar la información almacenada en las distintas tablas.

Dicho esto hay que reseñar asimismo que la normalización no siempre es un proceso deseable en la práctica pues a veces puede llevar a modelos en los que algunos tipos de consultas muy comunes son difíciles o costosas de realizar. En algunos casos se prefiere una pequeña cantidad de redundancia “innecesaria” para mejorar el funcionamiento global del sistema.

Tema 6
Elaboración del Modelo Físico

Tabla de Contenidos

1.- Introducción

Tal y como se describió anteriormente, los tres niveles de modelado de una base de datos son conceptual, lógico y físico. Tras la exposición de modelos concretos en los dos primeros niveles, en esta unidad se detalla el paso a un modelo físico concreto.

2.- Niveles de diseño de una base de datos

El modelado de bases de datos sigue tres niveles de diseño. El modelo conceptual es el más cercano al "mundo real". Los conceptos que maneja son muy cercanos a como vemos la realidad, aunque elimina información que no es aplicable a lo que queremos modelar pero no se asocia con ningún modelo informático. El resultado del modelo conceptual se puede implementar con un ordenador o con un fichero tradicional por escrito. El modelo más utilizado para representar el diseño conceptual es el modelo entidad-relación.

El segundo nivel lo da el modelo lógico. En éste se parte del modelo conceptual y lo adapta a un modelo que puede ser implementado en un ordenador. En este caso el modelo ya utiliza una abstracción que se puede implementar en un ordenador (como el modelo relacional) pero no está asociada a ninguna implementación en concreto, es decir, podemos utilizar el modelo lógico relacional para crear una base de datos en cualquier motor de bases de datos relacional, comercial o no.

El último nivel es el modelo físico. En éste nivel se adapta el modelo lógico, relacional en este caso, a un sistema de bases de datos concreto. Es en este nivel donde ya se obtiene un modelo que es utilizable físicamente en el ordenador y que puede utilizarse para almacenar y consultar datos reales.

En el diseño físico relacional se transforma un modelo lógico relacional a un sistema de base de datos en concreto. Para ello hay que:

- Mapear los dominios de los atributos a los tipos de datos que soporta el motor de base de datos elegido.
- Definir que campos son opcionales y por tanto pueden valer NULL.
- Establecer las claves primarias de cada tabla.
- Establecer las claves ajenas, a qué claves primarias se refieren y la acción a realizar cuando se actualiza o elimina una clave primaria.

Asimismo conviene realizar un análisis de las operaciones que se realizan más frecuentemente sobre los datos a fin de optimizar el almacenamiento o acceso.

3.- Introducción al lenguaje SQL

En los sistemas de bases de datos relacionales, el lenguaje que se utiliza para "conversar" con el sistema es el lenguaje SQL (Structured Query Language – Lenguaje de Consulta Estructurado).

SQL proviene del lenguaje SEQUEL que se diseñó originalmente para la base de datos relacional experimental de IBM System R. El nombre hubo que cambiarlo porque estaba sujeto a copyright.

A finales de los 70 la compañía que luego sería conocida como Oracle desarrolló su propia base de datos relacional e incluyó SQL como lenguaje de interacción.

El lenguaje SQL está estandarizado, primero por el ANSI y posteriormente por ISO, pero, desafortunadamente, la *implementación* de la estandarización no ha ido tan bien, por varios motivos:

- Cuando aparecieron los primeros estándares ya existían algunas implementaciones de SQL. Los fabricantes no se han atrevido a perder compatibilidad con código SQL antiguo por lo que mantienen estructuras y palabras no estándar.
- Las nuevas características que intentan introducir los fabricantes no tienen reflejo en el estándar, por lo que cada uno lo implementa a su manera. Cuando ya existe el estándar se aplica el caso anterior.
- La norma ha sido poco clara en algunos puntos y las implementaciones difieren entre fabricantes.

En resumen, que aunque al sintaxis y el funcionamiento es muy parecido entre fabricantes, hay diferencias, en algunos casos sustanciosas, entre las distintas implementaciones de SQL por lo que hay que suponer que sentencias escritas para un motor y versión concretos pueden no funcionar exactamente igual (o no funcionar en absoluto) en otros. Es conveniente revisar la documentación del motor que se está utilizando para conocer las diferencias.

4.- Clasificación de las sentencias de SQL

SQL nació como un lenguaje de consulta para expresar el álgebra relacional pero se le fueron añadiendo características y funcionalidades y hoy en día cubre todas las necesidades de interacción con al sistema de base de datos. Por esta razón, hoy en día se distinguen varios *subconjuntos* del lenguaje:

- DDL (Data Definition Language – Lenguaje de Definición de Datos). Parte del lenguaje que se encarga de crear y mantener la **estructura** de la base de datos.
- DML (Data Manipulation Language – Lenguaje de Manipulación de Datos). Es la parte del lenguaje que define como interactuar con los datos para añadir, modificar, borrar y consultar datos.
- DCL (Data Control Language - Languaje de Control de Datos). Es la parte del lenguaje que permite establecer permisos.

Inicialmente, en esta unidad vamos a revisar la parte correspondiente al DDL porque es la parte que necesitamos para crear y mantener la estructura de la base de datos.

5.- Características generales de SQL

El lenguaje SQL consta de ***sentencias*** que se pueden ver como comandos que realizan una tarea completa. Cada sentencia tiene una sintaxis que indica las opciones y datos que se pueden proporcionar a la sentencia. Tanto las sentencias como las distintas partes de cada una se separan utilizando palabras especiales, llamadas *palabras reservadas*, significando que el uso de cada palabra sólo puede ser el que le da la sintaxis SQL y no puede utilizarse para ninguna otra cosa, como por ejemplo nombre de tabla o columna. Las palabras reservadas no son sensibles a mayúsculas o minúsculas por lo que tanto SELECT, como Select como SeLeCt son la misma palabra clave, aunque se recomienda, para mejorar la legibilidad de las sentencias que se utilice siempre el mismo esquema de mayúsculas/minúsculas (todas mayúsculas, sólo la primera y el resto minúsculas, todas minúsculas, etc.).

Las sentencias se terminan con un punto y coma (;), que sirve de separador entre sentencias, si hubiera más de una en un mismo bloque o archivo.

Otro elemento importante de SQL son los identificadores, que referencian elementos de la base de datos, generalmente tablas y columnas (también llamadas campos o atributos).

También se pueden utilizar en las sentencias valores constantes como números o cadenas y operadores, aunque éstos se verán más adelante en otras unidades.

Usualmente, el intérprete SQL permite que se utilicen saltos de línea y tabuladores en cualquier sitio donde está permitido que aparezca un espacio en blanco, lo que puede mejorar la legibilidad del lenguaje, sobre todo en sentencias largas.

6.- Tipos de datos en SQL

SQL define los siguientes tipos de datos:

Tipo de datos	Descripción
CHARACTER(n)	Cadena de n caracteres de longitud fija, esto es, siempre contiene n caracteres. Si se le asigna un valor más corto se rellena el resto con espacios. Si se le asigna uno más largo se trunca a n.
VARCHAR(n)	Cadena de longitud variable con n caracteres máximo. Si se asigna un valor más corto que n sólo la longitud del valor asignado. Si se asigna uno más largo se trunca a n.
BINARY(n)	Array de n bytes. Longitud fija a n.
VARBINARY(n)	Array de bytes de longitud variable con un máximo de n bytes.
BOOLEAN	TRUE / FALSE
INTEGER(n)	Entero (sin decimales) de n número de cifras.
SMALLINT	Entero de 16 bits con signo.
INTEGER	No confundir con INTEGER(n). Entero de 32 bits con signo.
BIGINT	Entero de 64 bits con signo.

Tipo de datos	Descripción
DECIMAL(p,s)	Número con decimales. Tendrá como mucho p dígitos en la parte entera y s en la decimal.
NUMERIC(p,s)	Igual que DECIMAL
FLOAT(n)	Número en punto flotante con n bits de precisión en la mantisa.
REAL	Igual que FLOAT(7)
FLOAT	Igual que FLOAT(16)
DATE	Una fecha (dia, mes y año)
TIME	Una hora dentro de un día (hora, minuto y segundo)
INTERVAL	Un intervalo de tiempo

Estos serían los tipos estándar. Lamentablemente, como ya se ha comentado anteriormente, cada SGBD tiene su propia colección de tipos de datos con sus peculiaridades, por lo que hay que leer la documentación del SGBD concreto para conocer como describir un campo.

6.1.- Constantes en SQL

En muchas sentencias SQL hay que utilizar valores constantes, por lo que es importante el determinar cómo se escriben los mismos. Por desgracia, también aquí hay cierta diferencia entre SGBD, por lo que habrá que consultar la documentación del que se vaya a utilizar. De todas formas y de manera general, la sintaxis es la siguiente:

- Constantes de cadena. Se escriben entre comillas dobles (“) o simples (‘), aunque algunos SGBD sólo permiten una de las dos. Dentro de una cadena se pueden utilizar los caracteres de escape para crear caracteres difíciles de conseguir con el teclado o que tienen un significado especial, aunque esto depende fuertemente del SGBD. Un carácter de escape comienza por el símbolo (\ - backslash) y es seguido de un código numérico o una secuencia de caracteres que indican el símbolo que están representando.
- Constantes binarias. Son parecidas a las constantes de cadena pero hay que utilizar secuencias de escape cuando los bytes a almacenar no tienen representación visual, ésta es difícil de obtener mediante el teclado.
- Constantes enteras. Se escriben con números y opcionalmente un signo menos delante.
- Constantes numéricas reales. Constan de: Un signo opcional, parte entera, un punto, parte decimal. Si no hay parte decimal se puede omitir, así como el punto aunque la parte entera es obligatoria. También se pueden expresar en notación científica que es similar pero utilizando la letra e (ó E), un signo y el exponente. Por ejemplo -2.676E4 significa $-2.676\,x\,10^4$ ó 26700.
- Constantes de fecha y hora. Aquí hay una amplia diversidad entre los SGBDs, tanto en la forma de separar los distintos componentes (utilizando / ó -) como en el orden de los mismos (dia,mes y año;mes,dia y año; año, mes y dia, etc.). Lo mismo ocurre con ocurre con las horas, aunque estas utilizan normalmente el símbolo (:) y suelen ir siempre en el mismo orden, aunque

algunos SGBDs aumentan el tipo TIME para contener fracciones de segundo y esto influye en el formato de las constantes.

- Constantes booleanas. Algunos SGBDs tienen las constantes TRUE y FALSE para representar los valores verdadero y falso, respectivamente.
- Constante NULL. El valor NULL se representa con la palabra reservada NULL.

7.- Objetos que componen una base de datos

Un SGBD relacional contiene los siguiente objetos:

- Diccionario de datos. Es una metabase de datos, es decir, una base de datos *sobre* la base de datos. Dicho en otras palabras es un lugar en el que se almacena las definiciones del resto de objetos que componen la base de datos. El sistema la utiliza de forma automática y transparente al realizar la mayoría de las operaciones que se le ordenan y muchas de ellas lo modifican de forma indirecta. Muchos SGBDs implementan el diccionario como otra base de datos más pero con un status especial y con un acceso mucho más restringido.
- Usuarios y grupos: Para gestionar la seguridad de las bases de datos, muchos SGBDs implementan un sistema de permisos, similar al empleado en los sistemas de ficheros de los sistemas operativos. Se definen usuarios y grupos. Cada usuario tiene unas credenciales de acceso y tiene permiso para realizar ciertas operaciones sobre ciertos objetos de la base de datos. Los grupos permiten asignar permisos a múltiples usuarios relacionados al mismo tiempo, aunque no todos los sistemas los poseen. Algunos no poseen ni usuarios, no disponiendo de sistema de seguridad de acceso.
- Base de datos. Una base de datos es una colección de tablas, índices, vistas, eventos y programas. Asimismo una base de datos define un espacio de nombres para cada uno de estos objetos, esto es, pueden existir dos tablas con el mismo nombre en dos bases de datos distintas pero no dos tablas con el mismo nombre en la misma base de datos. Además, una base de datos tiene ciertas propiedades físicas (forma de almacenamiento, forma de ordenar los campos de texto, etc.) que le son propias. Usualmente es una buena práctica colocar datos de aplicaciones distintas en distintas bases de datos.
- Tabla. Una tabla es un espacio de almacenamiento de una serie de registros (filas o tuplas). Se define como una serie de campos o columnas, cada una con su tipo. Las filas se añaden y quitan pero tienen que dar valores para cada uno de los campos y este valor debe cumplir las restricciones del tipo al que pertenece el mismo. Una tabla se define con un nombre único entre las tablas y una serie de campos o columnas, cada uno con su nombre único entre los campos de una misma tabla y un tipo de datos.
- Índice. Un índice es un diccionario que se asocia con un campo o campos de una tabla, de forma que se accede más rápidamente a la fila que contiene un valor especificado de un campo. Algunos índices se crean de forma automática y otros se pueden crear explicitamente.

- Vista. Una vista es una tabla “virtual” que se obtiene al restringir las columnas o filas de un conjunto de datos de la base de datos. Se utilizan para dar una visión restringida de la base de datos a determinados usuarios, aunque su uso tiene un coste y tiene algunas limitaciones, sobre todo con las operaciones que modifican datos.
- Programa. Programa escrito en un lenguaje de programación adaptado a la base de datos y que puede actuar sobre los datos de ésta. Se utilizan para realizar funciones directamente sobre la base de datos, siendo muy rápidas y eficientes, aunque pueden producir efectos laterales indeseados.
- Eventos. Son condiciones que hacen que se dispare un programa. Se pueden establecer sobre los datos de la base de datos de forma que se ejecute un programa cuando los datos cumplan ciertas condiciones, cuando se elimine o modifique un dato, etc. Se propósito inicial es ayudar a mantener las reglas de integridad de la información aunque se pueden utilizar para cualquier cosa.

Todos estos objetos se pueden administrar directa o indirectamente a través del LDD (Lenguaje de Definición de Datos de SQL). La creación del modelo físico de la base de datos consiste precisamente en la creación de una serie de sentencias SQL, adaptadas al SGBD destino, que creen o ajusten estos objetos.

8.- El Lenguaje de Definición de Datos de SQL

El Lenguaje de Definición de Datos (LDD) es un subconjunto de SQL que permite administrar los objetos de la base de datos, en contraposición al Lenguaje de Manipulación de Datos (LMD) que permite la manipulación de los datos mantenidos por esos objetos.

En esta sección se va a describir el LDD de modo general.

8.1.- Convenciones

En este documento se van a a utilizar las siguientes convenciones a la hora de indicar los comandos SQL.

Los comandos y sus parámetros se indican con `fuente monoespacio`.

Las palabras reservadas del lenguaje se escriben con LETRAS MAYÚSCULAS. Estas palabras **no deben usarse como nombres para los elementos de las bases de datos**, como tablas o columnas.

Las palabras escritas con letras minúsculas indican parámetros o datos que hay que proporcionar a los comandos.

Las sentencias se incluyen en un recuadro con fondo gris.

Cuando se vaya a proporcionar una lista, se utilizaran puntos suspensivos (...) para indicar los elementos que no se van a escribir por abreviar pero que deberían proporcionarse cuando se utilice el lenguaje en un entorno real.

Las partes opcionales se colocan entre paréntesis cuadrados ([]). Estos paréntesis sólo sirven para delimitar en la explicación las partes opcionales y no se deben utilizar cuando se escriban comandos reales. Sólo se debe incluir, en caso de que se quiera usar la opción, el contenido de los paréntesis cuadrados.

Por ejemplo si se indica el comando:

```
COMANDO nombre [NULL]
```

esto indica que hay un comando llamado `COMANDO` que necesita como parámetro un `nombre` y puede opcionalmente ir seguido de la palabra `NULL`.

8.2.- Gestión de bases de datos

Hay tres operaciones que se pueden realizar sobre una base de datos: Crear una base de datos, modificar una base de datos existente y eliminar una base de datos.

8.2.1.- Creación de una base de datos

Una base de datos se crea utilizando la sentencia `CREATE DATABASE` con la siguiente sintaxis:

```
CREATE DATABASE nombredb [opciones];
```

donde

- `nombredb` es el nombre que se le desea dar a la nueva base de datos. No debe haber ya una base de datos con ese nombre o se producirá un error.
- `opciones` son distintas opciones generales de configuración que se pueden dar a la base de datos. Estas opciones dependen fuertemente del motor de base de datos elegido.

Ejemplo:

Crea una base de datos con nombre `mybase`

```
CREATE DATABASE mybase;
```

8.2.2.- Modificación de una base de datos

La configuración general de la base de datos se puede modificar utilizando la sentencia `ALTER DATABASE` con la siguiente sintaxis:

```
ALTER DATABASE nombredb nuevasopciones;
```

donde

- `nombredb` es la base de datos cuya configuración se desea modificar. Debe ser una base de datos existente.

- `nuevasopciones` son los nuevos valores de la configuración. Al igual que con `CREATE DATABASE`, estas opciones dependen del SGBD utilizado.

Ejemplo (en MariaDB):

Modifica la base de datos mybase para que utilice el juego de caracteres utf8 para las cadenas de caracteres.

```
ALTER DATABASE mybase CHARACTER SET utf8;
```

8.2.3.- Borrado de una base de datos

Se puede eliminar una base de datos completamente utilizando la sentencia `DROP DATABASE` con la siguiente sintaxis:

```
DROP DATABASE nombredb;
```

donde:

- `nombredb` es el nombre de la base de datos a eliminar. Si la base de datos no existe se produce un error.

Hay que tener mucha precaución en el uso de este comando porque la base de datos se elimina completamente y con ella todos los objetos que contenga: tablas, índices, vistas, etc., junto con los datos almacenados.

Ejemplo:

Eliminar la base de datos `mybase`

```
DROP DATABASE mybase;
```

8.3.- Gestión de tablas

Las tablas son los objetos más importantes que contiene una base de datos pues son el sitio donde se almacena la información y sobre los que basan su operación el resto de los objetos de la base de datos.

Las operaciones que se pueden realizar sobre las tablas son:

- Crear una tabla
- Modificar una tabla (su estructura, no sus datos)
- Eliminar una tabla.

8.3.1.- Creación de tablas

Las tablas se crean utilizando la sentencia `CREATE TABLE` con la siguiente sintaxis:

```
CREATE TABLE nombretabla (definicioncol1, definicioncol2, ..., definicioncoln)
opciones;
```

donde

- `nombretabla` es el nombre de la tabla a crear. No puede existir una tabla con ese nombre en la base de datos o el comando fallará.
- `definicioncol`*X*. Cada una de ellas es, o bien una definición de columna (campo o atributo) o bien una restricción de tabla. Se elaborará mas adelante. El conjunto de las definiciones va encerrado entre paréntesis y las definiciones van separadas por comas.
- `opciones`. Opciones generales de la tabla. Dependen fuertemente del SGBD, pero pueden especificar desde el tipo de almacenamiento de la tabla como la ubicación en disco, etc.

La definición de una columna sigue la siguiente sintaxis:

```
nombrecol tipocol restricciones
```

donde

- `nombrecol` es el nombre de la columna. Debe ser único entre los nombres de columna de la tabla, esto es, no puede haber dos columnas con el mismo nombre o se producirá un error.
- `tipocol` es el tipo de la columna. Uno de los tipos indicados en secciones anteriores o uno propio del SGBD utilizado.
- `restricciones` son restricciones adicionales que se pueden aplicar a la columna y que pueden ser:
  - `NULL`. Indica que la columna puede admitir valores nulos. Es la opción por defecto, por lo que no es necesario utilizarlo aunque nosotros vamos a utilizarlo siempre para que quede clara la intención.
  - `NOT NULL`. Indica que la columna **NO** puede admitir valores nulos. Esta opción hay que indicarla obligatoriamente cuando se necesite porque de lo contrario se aplicaría la opción por defecto que es permitir los valores nulos. Sólo se puede especificar uno de los dos (o `NULL` o `NOT NULL`).
  - `DEFAULT valorpordefecto`. Indica el valor por defecto que recibirá la columna si se crea una fila en la tabla y no se proporciona valor para la columna. Si no se da un valor por defecto, depende del SGBD el valor que se asignará.
  - `PRIMARY KEY`. Indica que la columna es la clave primaria de la tabla. Esto usualmente implica que en esta columna los valores deben ser únicos, esto es, que no puede haber dos filas con el mismo valor en esta columna y que se creará un índice sobre esta columna. Sólo se puede utilizar para una columna. Si se quiere crear una clave primaria que conste de más de una columna hay que utilizar restricciones (`CONSTRAINT`) (ver más adelante).

- `UNIQUE`. Indica que la columna contendrá valores únicos por fila, esto es, que el valor de la columna debe ser distinto para cada fila y usualmente implica crear un índice sobre la columna.

Ejemplo:

Se desea crear una tabla llamada `consulta` con los siguientes campos:

- `cod_consulta`. Entero. Clave primaria.
- `id_med`. Entero. Código del médico que realiza la consulta. Es clave ajena a la tabla `medico` y el campo `id_med`.
- `fecha`. Fecha. No puede ser nulo.
- `diagnostico`. Caracter de hasta 200 caracteres. No puede ser nulo.
- `id_pac`. Entero. Código del paciente que ha recibido la consulta. Es clave ajena a la tabla `paciente` y el campo `id_pac`.

La sentencia de creación sería:

```
CREATE TABLE consulta (
     cod_consulta INTEGER PRIMARY KEY,
     id_med INTEGER NOT NULL,
     fecha DATE NOT NULL,
     diagnostico VARCHAR(200) NOT NULL,
     id_pac INTEGER NOT NULL
);
```

No se han definido las claves ajenas porque para ello hay que utilizar restricciones de tabla, que se verán a continuación.

Hay que hacer notar que las dos claves ajenas (`id_med` e `id_pac`) se han definido como `NOT NULL`. Esto indica que tanto la relación entre `consulta` y `medico` como entre `consulta` y `paciente` no es opcional. Si lo fuera debería de indicarse como `NULL` (o no poner nada y que tome el valor por defecto, que es `NULL`).

Anteriormente, cuando se ha hablado de la definición de la sentencia CREATE TABLE se han dejado sin comentar las restricciones de tabla. Esto ha sido así para poder introducir la sintaxis básica y ahora se van introducir.

Las restricciones de tabla son restricciones que se aplican a la tabla completa o a un conjunto de columnas de la tabla y que, por tanto, no se pueden definir como opción de una sola columna. Las restricciones de tabla pueden ser una de:

- `CONSTRAINT PRIMARY KEY (col1, con2, …, coln)`. Define la clave primaria de la tabla que consta de varias columnas (`col1`, `col2`, …, `coln`).
- `INDEX [nombre] (col1, col2, …, coln)`. Define un índice sobre las columnas `col1`, `col2`, …, `coln`. Al índice se le denomina `nombre` (si se da, si no el sistema le proporciona un nombre automáticamente).
- `CONSTRAINT UNIQUE (col1, col2, …, coln)`. Define una restricción de unicidad. La combinación de los valores de las columnas dadas debe ser único para cada fila de la tabla.
- `CONSTRAINT FOREIGN KEY [nombre] (col1, col2, …, coln) references`. Define una clave ajena con el `nombre` dado (si no se da, el sistema asigna uno). La definición de la parte `references` es:
  - `REFERENCES tablaref (columnaref) [ON DELETE opcion_ref1] [ON UPDATE opcion_ref2]`. Indica que el campo es una clave ajena que referencia el campo `columnaref` de la tabla `tablaref` (que debería ser la clave primaria de la tabla, aunque esto no es obligatorio). Las opciones `opcion_ref1` y `opcion_ref2` indican la acción a tomar cuando se elimina o modifica, respectivamente, un valor en la columna `columnaref` de la tabla `tablaref` que coincide con el valor de esta columna en alguna fila de la tabla. Las opciones pueden ser:
    - `RESTRICT`. No se permite la operación. Es la opción por defecto.
    - `CASCADE`. La fila o filas con el valor coincidente se elimina(n) o su valor se modifica para que sea igual al nuevo valor asignado en la tabla referenciada, de forma que se mantenga la referencia.
    - `SET NULL`. En la fila o filas con el valor coincidente se cambia(n) este por `NULL`. Es un error si se ha declarado el campo como `NOT NULL`.

Ejemplo:

Se va a hacer la misma definición del ejemplo anterior, pero utilizando restricciones de tabla, cuando se pueda:

```
CREATE TABLE consulta (
     cod_consulta INTEGER,
     id_med INTEGER NOT NULL,
     fecha DATE NOT NULL,
     diagnostico VARCHAR(200) NOT NULL,
     id_pac INTEGER NOT NULL,
     CONSTRAINT PRIMARY KEY (cod_consulta),
```

```
     CONSTRAINT FOREIGN_KEY (id_med) REFERENCES medico(id_med),
     CONSTRAINT FOREIGN KEY (id_pac) REFERENCES paciente(id_pac)
);
```

8.3.2.- Modificar una tabla

La modificación de una tabla se realiza utilizando la sentencia `ALTER TABLE`. La sentencia tiene múltiples sintaxis porque puede ser utilizada para realizar muchas modificaciones distintas sobre una tabla. Por lo tanto se van a definir por separado. Aun asi, todas comparten un inicio comun. La sintaxis comun es:

```
ALTER TABLE nombretabla modificacion;
```

donde:

- `nombretabla` es el nombre de la tabla que se va a modificar.
- `modificacion` es la modificación que se quiere realizar que se detallará en las secciones siguientes.

8.3.2.1.- Añadir una columna

Sintaxis:

```
ADD COLUMN nombrecol defcol;
```

donde:

- `nombrecol` es el nombre de la nueva columna. Debe ser distinto a los nombres de columnas ya existentes en la tabla.
- `defcol`. Definición de columna (tipo, `NULL`, etc.). Como en `CREATE TABLE`.

Ejemplo:

Se desea añadir a la tabla consulta, ya creada, un nuevo campo que se ha olvidado, receta, que es un campo de texto con una longitud de 500 caracteres. Se podría añadir utilizando la sentencia:

```
ALTER TABLE consulta ADD COLUMN receta VARCHAR(500) NOT NULL;
```

8.3.2.2.- Modificar una columna

Sintaxis:

```
MODIFY COLUMN nombrecol defcol;
```

donde el significado de `nombrecol` y `defcol` es el mismo que en la sintaxis anterior.

Hay que tener en cuenta que no siempre se puede cambiar el tipo de una columna ya que el nuevo tipo debe de poder ser convertido desde el anterior, por ejemplo, si se convierte un campo desde `VARCHAR`

a `INTEGER` es posible que nos encontremos con algún valor que no tiene equivalencia. En ese caso se producirá un error.

Ejemplo:

Se desea modificar la longitud del campo diagnostico de la tabla consulta para que pueda contener 500 caracteres en lugar de los 200 actuales. Se podria hacer con la sentencia:

```
ALTER TABLE consulta MODIFY COLUMN diagnostico VARCHAR(500);
```

8.3.2.3.- Modificar el nombre de una columna

Sintaxis:

```
CHANGE COLUMN nombrecolant nombrecolnuevo;
```

donde:

- `nombrecolant` es la columna cuyo nombre se desea cambiar
- `nombrecolnuevo`. Nuevo nombre que se le desea dar a la columna.

Ejemplo:

En la tabla consulta se desea modificar el nombre del campo receta a prescripcion. Para ello se utilizaría la sentencia:

```
ALTER TABLE CONSULTA CHANGE COLUMN receta prescripcion;
```

8.3.2.4.- Modificar el valor por defecto de una columna

Sintaxis:

```
ALTER COLUMN nombrecol definicion_defecto;
```

donde:

- `nombrecol` es el nombre de la columna cuyo valor por defecto se desea modificar.
- `definicion_defecto` es el nuevo valor por defecto que puede ser uno de los siguientes:
  - `SET DEFAULT valor_defecto`. Cambia el valor por defecto de la columna a `valor_defecto`.
  - `DROP DEFAULT`. Elimina el valor por defecto. Si no se le da un valor al campo tomará `NULL`, si puede, o dará un error si no puede ser `NULL`.

Ejemplo:

Se va a poner un valor por defecto "`Ninguno`" a la columna diagnostico de la tabla consulta cuando no se especifique ningún diagnóstico. Para ello usaríamos:

```
ALTER TABLE consulta ALTER COLUMN diagnostico SET DEFAULT "Ninguno";
```

8.3.2.5.- Eliminar una columna

Sintaxis:

```
DROP COLUMN nombrecol;
```

donde `nombrecol` es el nombre de la columna a eliminar. Tanto la columna como los datos que contenía desaparecerán.

Ejemplo:

Se desea eliminar la columna `prescripcion` de la tabla `consulta`. Se podría utilizar:

```
ALTER TABLE consulta DROP COLUMN prescripcion;
```

8.3.2.6.- Añadir una clave primaria

Sintaxis:

```
ADD CONSTRAINT PRIMARY KEY (col1, col2, ..., coln);
```

Añade una clave primaria a la tabla compuesta por las columnas `col1`, `col2`, ..., `coln`.

Ejemplo:

Si se supone que se ha olvidado indicar la clave primaria de la tabla consulta en su creación, se podría hacer posteriormente utilizando:

```
ALTER TABLE consulta ADD CONSTRAINT PRIMARY KEY (cod_consulta);
```

8.3.2.7.- Añadir una clave ajena

Sintaxis:

```
ADD CONSTRAINT FOREIGN KEY nombre (col1, col2, ..., coln) references;
```

donde:

- `nombre` es el nombre de la clave ajena. Si no se especifica, el sistema asigna un nombre automáticamente.
- `col`*X*. Columnas que forman parte de la clave ajena.
- `references`. Igual que la parte homónima en la restricción de tabla de `CREATE TABLE`.

Ejemplo:

Si en consulta se ha olvidado incluir que `id_pac` es una clave ajena a la clave primaria `id_pac` de la tabla `paciente`, se puede hacer más tarde utilizando la sentencia:

```
ALTER TABLE consulta ADD CONSTRAINT FOREIGN KEY id_pac_ext (id_pac) REFERENCES
paciente(id_pac);
```

8.3.3.- Eliminar una tabla

Una tabla se elimina utilizando la sentencia `DROP TABLE`. La sintaxis es la siguiente:

```
DROP TABLE nombre_tabla;
```

donde `nombre_tabla` es el nombre de la tabla a eliminar.

Hay que tener cuidado al utilizar este comando porque no se solicita confirmación y elimina la tabla, su contenido y los índices y claves primarias y ajenas vinculadas a ella.

Esta operación no puede deshacerse.

Ejemplo:

Si se quiere eliminar la tabla `consulta`, se podría hacer con la sentencia:

```
DROP TABLE consulta;
```

9.- Índices

Los índices son estructuras asociadas a uno o más columnas de una tabla y que permiten realizar búsquedas en esas columnas de forma más rápida. Funcionan como un diccionario en el que se ordenan los valores de las columnas que forman parte del índice y por cada una de ellas se almacena la dirección de la fila o filas en las que están contenidas y que contienen los datos correspondientes, permitiendo un acceso directo a la información en lugar de recorrer toda la tabla buscando coincidencias.

Por lo tanto son estructuras útiles si se van a realizar frecuentemente búsquedas o uniones de tablas utilizando los valores de dichos campos.

Sin embargo no todo son ventajas, ya que los índices hay que mantenerlos y cada vez que se inserta o elimina información en una tabla con índices o se modifica un valor de una columna que entra en un índice hay que actualizar los índices correspondientes para que sigan siendo válidos y esas operaciones tienen un costo.

Así, se recomienda el uso de los índices sobre todo en:

- Claves primarias de las tablas, ya que se suelen realizar muchas búsquedas sobre ellas, tanto para localizar una fila determinada como cuando se realizan uniones de tablas, ya que suele ser la columna o columnas que se utilizan para realizar la unión.
- Claves ajenas de tablas. Por la misma razón que las claves primarias respecto a la unión de tablas.

- Campos sobre los que se realizan muchas búsquedas. Por ejemplo, si se utiliza mucho la columna CodigoPostal para localizar a personas que viven en una determinada zona, quizá sería aconsejable crear un índice sobre ese campo.

9.1.- Crear un índice

Los índices se pueden crear de dos formas:

- Implicitamente. Determinadas operaciones sobre la estructura de una tabla crean índices de manera implícita. Por ejemplo, muchos SGBDs crean un índice al definir la clave primaria o claves ajenas de una tabla de forma transparente.
- Explicitamente. Se indica, mediante una sentencia SQL, que se desea crear un índice sobre una columna o columnas. Esta es la sintaxis que se va a describir en esta sección.

Para crear un índice de forma explícita se utiliza la siguiente sintaxis:

```
CREATE [UNIQUE] INDEX nombreindice ON nombretabla (col1, col2, ..., coln);
```

donde:

- `nombreindice` es el nombre del nuevo indice. El nombre se utiliza para poder gestionarlo desde SQL, ya que su uso es automático.
- `nombretabla`. Tabla sobre la que se va a crear el índice.
- `colx`. Columna o columnas que forman parte del índice.

Si se utiliza el modificador `UNIQUE`, el índice será único, esto es, no permitirá entradas duplicadas sobre los campos del índice. Para un índice sobre una sola columna, esto implica que no podrá haber valores repetidos en dicha columna. Para un índice sobre varias columnas, esto implica que, para cada fila, la combinación de los valores de las columnas que forman el índice debe ser única.

Ejemplo:

Si se desea crear un índice sobre la columna `prescripcion` de la tabla `consulta` se podría realizar de la siguiente forma:

```
CREATE INDEX idxprescripcion ON consulta (prescripcion);
```

9.2.- Eliminar un índice

Un índice puede eliminarse si se considera que ya no es necesario.

La sintaxis para eliminar un índice es:

```
DROP INDEX nombreindice ON nombretabla;
```

Ejemplo:

Para eliminar el índice `idxprescripcion` anteriormente creado se utilizaría:

```
DROP INDEX idxprescripcion ON consulta;
```

10.- Resumen

En esta unidad se ha descrito la implementación de un modelo lógico relacional en un modelo físico general, no ligado a ningún SGBD concreto.

Para ello se ha descrito la sintaxis general de SQL y la sintaxis particular del LDD, de forma que se ha descrito la forma de mantener bases de datos y, dentro de ellas, tablas e índices. Este conocimiento nos permitirá implementar un modelo lógico relacional en un SGBD concreto.

Apéndice A.- SGBD MariaDB (MySQL)

En esta sección se describirá con más detalle el LDD concreto del SGBD MariaDB (fork de MySQL). La versión que se describe es la 10.6.5.

Tipos de datos

Los tipos de datos de MariaDB (tal y como se puede ver en la página del manual https://mariadb.com/kb/en/data-types/) son los siguientes:

- Enteros
  - `TINYINT`: 8 bits
  - `SMALLINT`: 16 bits
  - `MEDIUMINT`: 24 bits.
  - `INTEGER` ó `INT`: 32 bits
  - `BIGINT`: 64 bits.
- Reales de punto fijo (número fijo de decimales)
  - `DECIMAL(n, m)` ó `NUMERIC(n,m)`: Número de punto fijo con `n` dígitos en el número completo y `m` en la parte decimal. Dicho de otra manera, el número tiene `n` dígitos, de los que `m` forman la parte decimal y el resto (`n-m`) la parte entera.
- Reales en punto flotante
  - `FLOAT`: 32 bits
  - `DOUBLE`: 64 bits.
  - `FLOAT(T,D)`, `DOUBLE(T,D)`: Número real con `T` dígitos, de los cuales hasta `D` pueden ser decimales.

Adicionalmente los tipos enteros pueden contener el atributo `UNSIGNED` que hace que se almacene el número sin signo. También pueden tener el atributo `AUTO_INCREMENT` que genera una secuencia automática de números. Para ello, cada vez que se inserta una fila con un valor `NULL` ó `0` en esta columna, el SGBD genera el siguiente número de la secuencia y lo inserta en lugar de `NULL` ó `0`. (**para ello es importante utilizar el atributo `NOT NULL` en la declaración del campo al crear la tabla**). Si se da un valor numérico distinto de `0`, es ese valor el que se inserta y la secuencia se reinicializa a dicho valor, tomando la próxima vez el siguiente valor al proporcionado.

- Fecha y hora
  - `DATE`. Contiene una fecha, en formato `AAAA-MM-DD`, donde `AAAA` es el año, `MM` el mes y `DD` el día, todos en número.
  - `DATETIME`. Contiene una fecha y una hora dentro de esa fecha, en formato "`AAA-MM-DD HH:MM:SS.FFFFFF`" donde `AAAA`, `MM` y `DD` son como en `DATE`, `HH` son horas, `MM` minutos, `SS` segundos y `FFFFFF` fracciones de segundo.
  - `TIMESTAMP`. Similar a `DATETIME` pero con un rango de fechas distinto a `DATETIME`.
  - `TIME`. Contiene una hora en formato "`HHH:MM:SS.FFFFFF`". Formato similar a `DATETIME`. La hora puede tener tres dígitos si lo que se quiere representar una duración, en lugar de una hora del día.
  - `YEAR(n)`. Representa un año. `n` puede ser 2 ó 4, aunque no se recomienda utilizar el valor 2 porque está marcado para desaparecer en futuras versiones.
- Texto:
  - `CHAR(n)`. Cadena de tamaño fijo de `n` caracteres, donde `n` puede ser hasta 255. La cadena se almacena siempre con esa longitud. Si se almacena una cadena más corta se rellena con espacios, que se eliminan al consultar el dato.
  - `VARCHAR(n)`. Cadena de tamaño variable de hasta `n` caracteres, donde n puede ser hasta 65535. La cadena se almacena en formato `(longitud, datos)` por lo que sólo ocupa lo mínimo necesario más 2 bytes para almacenar la longitud.
  - `BINARY.(n)`. Como `CHAR` pero contiene bytes en lugar de caracteres, es decir, puede contener cualquier número binario.
  - `VARBINARY(n).` Como `VARCHAR` pero contiene bytes en lugar de caracteres.
  - `TEXT` y `BLOB`. Pueden contener texto (`TEXT`) o datos binarios (`BLOB`) de longitud arbitraria. Su uso no es muy recomendado porque puede afectar fuertemente el rendimiento del sistema.

Sintaxis CREATE DATABASE

```
CREATE [OR REPLACE] DATABASE [IF NOT EXISTS] nombredb
[CHARACTER SET nombrejuegocar]
[COLLATE nombreordenacion];
```

donde:

- La opción `OR REPLACE` comprueba si ya existe una base de datos con el nombre proporcionado y la elimina si éste es el caso. Si no existe una base de datos con el nombre proporcionado no tiene ningún efecto. **No puede ser utilizada junto a la opción `IF NOT EXISTS`.**
- La opción `IF NOT EXISTS` hace que se cree la base de datos sólo si no existe una ya con el nombre proporcionado. Si ya existe no se hace nada. Si no se especifica y ya existe una base de datos con ese nombre se producirá un error (y tampoco se hace nada). **No puede ser utilizada junto a la opción `OR REPLACE`.**
- `nombredb`. Nombre de la nueva base de datos a crear. Debe ser distinto a los nombres ya existentes o se producirá un error (si no se proporciona `IF NOT EXISTS`).
- `nombrejuegocar` es el nombre del juego de caracteres que se va a utilizar por defecto en la nueva base de datos para almacenar los valores de los campos de tipo texto. Es importante que sea adecuado al idioma que se va a utilizar en los valores para que no haya problemas con determinadas letras, como la eñe o vocales acentuadas. La lista de juegos de caracteres es muy larga para incluirla aquí pero se puede consultar en la web de MariaDB.
- `nombreordenacion` es el nombre del sistema de ordenación que se va a utilizar al comparar cadenas en la base de datos. El sistema de ordenación establece cual es el orden alfabético entre los caracteres de un juego de caracteres. Por ejemplo, el sistema de ordenación español establece que la ñ va después de la n y antes de la o. Si se establece otro sistema es posible que la ñ esté en otra posición y al obtener una consulta ordenada las palabras que comienzan por Ñ no estén en la posición que deberían ocupar en la lista.

Página del manual: https://mariadb.com/kb/en/create-database/

Sintaxis ALTER DATABASE

```
ALTER DATABASE nombredb
[CHARACTER SET nombrejuegocar]
[COLLATE nombreordenacion];
```

La sintaxis y las opciones, junto con su significado son muy similares a las de `CREATE DATABASE`

Página del manual: https://mariadb.com/kb/en/alter-database/

Sintaxis `DROP DATABASE`

```
DROP DATABASE [IF EXISTS] nombredb;
```

donde:

- `IF EXISTS`, si se usa, previene que ocurra un error al intentar eliminar una base de datos que no existe. Si se usa y la base de datos no existe no ocurre nada. Si no se usa y la base de datos no existe se produce un error.
- `nombredb`. Nombre de la base de datos a eliminar.

Página del manual: https://mariadb.com/kb/en/drop-database/

Sintaxis `CREATE TABLE`

Además de lo ya indicado en la descripción general y al uso de los tipos propios de MariaDB, no hay mucho más que añadir, excepto que se pueden indicar una serie de opciones, entre ellas, por ejemplo:

- `OR REPLACE`. Si se utiliza la opción `OR REPLACE` (justo entre `CREATE` y `TABLE`), se comprueba si ya existe una tabla con dicho nombre en la base de datos y se elimina antes de realizar la creación. De esta forma se puede reemplazar una tabla por otra con una nueva definición y sin datos. **No se puede utilizar junto a la opción `IF NOT EXISTS`.**
- `IF NOT EXISTS`. Esta opción, que si se usa debe ir justo después de TABLE, indica que si la tabla ya existe, en lugar de dar un error, se termina silenciosamente sin hacer nada. **No se puede utilizar junto a la opción `OR REPLACE`.**
- `ENGINE nombremotor`. MariaDB posee la particularidad de que puede utilizar distintas implementaciones físicas o *motores* para almacenar la información de las tablas, cada uno con sus peculiaridades y adaptado a un tipo de tarea. De entre ellos, el más utilizado es el InnoDB que es más moderno y permite el uso de funcionlidades avanzadas, como transacciones y claves ajenas, que no son soportadas en otros motores, por lo que se recomienda su uso, si no está claro cual motor utilizar. Otra opción es el motor MyISAM, que carece de funcionalidad avanzada pero es eficiente en espacio y en tiempo y puede ser utilizado si las características avanzadas no se requieren. También hay motores especializados en interactuar con datos en formatos de otras aplicaciones o de sólo lectura. Hay que consultar la documentación oficial y la no oficial para determinar cual es el motor más adecuado a nuestras necesidades. En este curso se utilizará el motor InnoDB si no se especifica expresamente lo contrario. Para más información consultar (https://mariadb.com/kb/en/choosing-the-right-storage-engine/)
- `CHARACTER SET nombrejuegocaracteres`. Permite utilizar un juego de caracteres en esta tabla distinto al general especificado para la base de datos. Si no se especifica se utiliza el que se haya definido para la base de datos.
- `COLLATE nombreorden`. Igual que `CHARACTER SET`
- `AUTO_INCREMENT valor`. Valor inicial para las columnas `AUTO_INCREMENT` de la tabla.

- `COMMENT “comentario”`. Permite introducir un comentario sobre la tabla. Útil para almacenar descripciones o información para los administradores / desarrolladores.
- Para crear un índice no hay que utilizar la palabra clave `CONSTRAINT`, vista en la sintaxis general. El resto es idéntico.

Página del manual: https://mariadb.com/kb/en/create-table/

Sintaxis ALTER TABLE

Sin comentarios adicionales sobre la definición general.

Página del manual: https://mariadb.com/kb/en/alter-table/

Sintaxis DROP TABLE

```
DROP TABLE [IF EXISTS] nombre_tabla;
```

Si se utiliza `IF EXISTS` y la tabla a borrar no existe no se produce error.

Página del manual: https://mariadb.com/kb/en/drop-table/

Apéndice B.- Uso de la consola de MariaDB

MariaDB (y MySQL) se pueden administrar mediante la consola de comandos.

Para ello hay que abrir un terminal y en el prompt ejecutar la consola mysql con el comando:

```
mysql -u root -p
```

La opción `-u usuario`, indica que se dese iniciar sesión como el usuario indicado. En este caso se utilizan el usuario `root`, que es el usuario administrador de la base de datos. Normalmente es una mala idea explotar una base de datos utilizando el usuario administrador, ya que un error puede provocar efectos a nivel de todo el SGBD pero hasta que se describa el sistema de permisos, es el que se va a utilizar. La opción `-p` indica que la consola debe solicitar la contraseña del usuario `root`. Introducir esta contraseña.

Si todo va bien, se deberá ver un mensaje de MySQL y el prompt de la consola

```
MariaDB>
```

o bien

```
MySQL>
```

A partir de este momento se pueden comenzar a escribir comandos que se ejecutarán al pulsar la tecla ENTER.

Además de las sentencias SQL, MariaDB dispone de algunos comandos de control que pueden ser muy útiles:

- `show databases;` Muestra una lista de las bases de datos existentes en el sistema. Útil para comprobar si una base de datos existe ya o no.
- `use nombredb;` Hace que la base de datos con el nombre nombredb sea la base de datos activa o seleccionada. A partir de ese momento, las sentencias que se ejecuten lo harán sobre dicha base de datos.
- `show tables;` Cuando hay una base de datos seleccionada muestra las tablas que contiene.
- `quit` No necesita punto y coma al final. Termina la sesión de consola de MySQL y sale al prompt del S.O.

Otro comando muy útil es `mysqldump`.

`mysqldump` hace un volcado de una base de datos ya existente en formato SQL, esto es, genera, a partir de una base de datos existente, un archivo SQL con las sentencias necesarias para crear la base de datos, sus objetos y rellenar las tablas con los datos que tenía.

La sintaxis básica que se va a utilizar de `mysqldump` es:

```
mysqldump -u root -p nombredb
```

donde `nombredb` es el nombre de la base de datos cuyo volcado se desea generar. Tras solicitar y obtener la contraseña del administrador, este comando volcará a consola las sentencias. Si se desea que estas se almacenen en un archivo para su uso posterior, habrá que redireccionar la salida estándar a un archivo con la siguiente forma:

```
mysqldump -u root -p nombredb >archivosalida
```

donde `archivosalida` es el nombre del archivo de salida que queremos generar. Si no existe, se creará y si ya existe, se sobreescribirá.

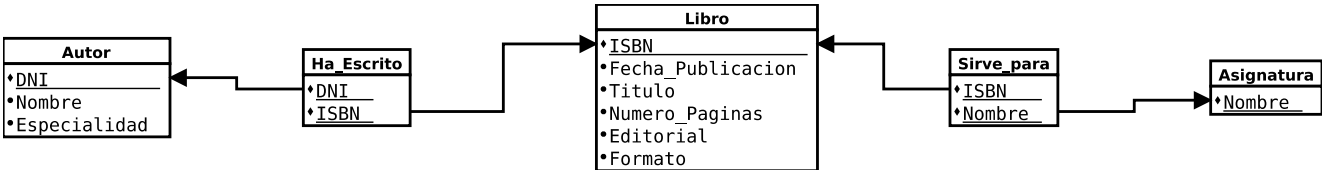
Por ejemplo, el comando:

```
mysqldump -u root -p hospital >hospital.sql
```

creará un archivo llamado `hospital.sql` con las sentencias necesarias para crear y rellenar la base existente hospital.

Tema 6
Elaboración del Modelo Físico
Ejemplo de conversión de modelo lógico relacional a modelo físico utilizando MariaDB

Sea el siguiente modelo relacional:



Obtener una secuencia de sentencias SQL que creen la base de datos y las tablas en MariaDB 10.6

En primer lugar hay que crear la base de datos en si. Dado que no se nos proporciona un nombre, podemos llamarla publicacion. Por lo tanto, la creación de la base de datos se realizaría:

```
CREATE OR REPLACE DATABASE publicacion CHARACTER SET utf8 COLLATE utf8_spanish2_ci;
```

Esta sentencia crea la base de datos `publicacion` con el juego de caracteres `utf8` y las reglas de comparación de español en `utf8`. Es una apuesta más o menos segura si el texto de los campos va a contener palabras o nombres españoles. Si no probablemente sea más seguro utilizar `utf8_general_ci`, que es el orden por defecto en `utf8`, si no se especifica, por lo que las dos sentencias siguientes son equivalentes:

```
CREATE DATABASE publicacion CHARACTER SET utf8 COLLATE utf8_general_ci;
```

Y

```
CREATE DATABASE publicacion CHARACTER SET utf8;
```

Nosotros nos quedaremos con el primero que hemos hecho ya que suponemos que nuestros datos de texto estarán en español.

Una vez creada la base de datos, la seleccionamos para comenzar a utilizarla:

```
use publicacion;
```

Para evitar problemas con las referencias, lo mejor es crear las tablas sin utilizar claves ajenas y, una vez creadas, establecer las claves ajenas. Esto es así porque algunos SGBDs (entre ellos MariaDB) no crean una clave ajena si la tabla y clave primaria a la que referencian no existe. Por lo tanto tenemos dos opciones: O bien creamos las tablas, comenzando por las que no referencian a nadie y añadiendo las que sólo referencian a las ya creadas hasta que acabemos o creamos todas las tablas, en el orden que queramos, sin incluir las claves ajenas y posteriormente las añadimos utilizando `ALTER TABLE` para

ello. Vamos a seguir el segundo método ya que es más seguro y siempre se puede aplicar dado que a veces el primero no es aplicable si hay tablas que se referencian unas a otras mutuamente.

La creación de las tablas, sin claves ajenas sería:

```
CREATE OR REPLACE TABLE autor (
    dni CHAR(9) NOT NULL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    especialidad VARCHAR(50) NOT NULL
);
CREATE OR REPLACE TABLE ha_escrito (
    dni CHAR(9) NOT NULL,
    isbn CHAR(13) NOT NULL,
    CONSTRAINT PRIMARY KEY (dni,isbn)
);
CREATE OR REPLACE TABLE libro (
    isbn CHAR(13) NOT NULL PRIMARY KEY,
    fecha_publicacion DATE NOT NULL,
    titulo VARCHAR(150) NOT NULL,
    numero_paginas INTEGER NOT NULL,
    editorial VARCHAR(100) NOT NULL,
    formato VARCHAR(50) NOT NULL
);
CREATE OR REPLACE TABLE sirve_para (
    isbn CHAR(13) NOT NULL,
    nombre VARCHAR(50) NOT NULL,
    CONSTRAINT PRIMARY KEY (isbn, nombre)
);
CREATE OR REPLACE TABLE asignatura (
    nombre VARCHAR(50) NOT NULL PRIMARY KEY
);
```

Esto crearía las tablas, sin establecer las claves ajenas, aunque las columnas necesarias están ahi.

Hay que hacer algunos comentarios sobre los comandos:

- `dni` se ha tomado como que son 9 caracteres y siempre 9 (8 números y una letra).
- `isbn` son 13 dígitos, aunque se almacena como una cadena porque se desean mantener todos los dígitos, incluidos los ceros a la izquierda.
- Los campos de texto se han declarado como `VARCHAR` y la longitud se ha estimado.

A continuación creamos las claves ajenas:

```
ALTER TABLE ha_escrito ADD CONSTRAINT FOREIGN KEY (dni) REFERENCES
autor(dni);
ALTER TABLE ha_escrito ADD CONSTRAINT FOREIGN KEY (isbn) REFERENCES
libro(isbn);
ALTER TABLE sirve_para ADD CONSTRAINT FOREIGN KEY (isbn) REFERENCES
libro(isbn);
ALTER TABLE sirve_para ADD CONSTRAINT FOREIGN KEY (nombre)
REFERENCES asignatura(nombre);
```

Una vez hecho esto ya tenemos creada nuestra primera base de datos. La secuencia completa de sentencias, sin interrupciones sería:

```
CREATE OR REPLACE DATABASE publicacion CHARACTER SET utf8 COLLATE
utf8_spanish2_ci;
use publicacion;
CREATE OR REPLACE TABLE autor (
     dni CHAR(9) NOT NULL PRIMARY KEY,
     nombre VARCHAR(100) NOT NULL,
     especialidad VARCHAR(50) NOT NULL
);
CREATE OR REPLACE TABLE ha_escrito (
     dni CHAR(9) NOT NULL,
     isbn CHAR(13) NOT NULL,
     CONSTRAINT PRIMARY KEY (dni,isbn)
);
CREATE OR REPLACE TABLE libro (
     isbn CHAR(13) NOT NULL PRIMARY KEY,
     fecha_publicacion DATE NOT NULL,
     titulo VARCHAR(150) NOT NULL,
     numero_paginas INTEGER NOT NULL,
     editorial VARCHAR(100) NOT NULL,
     formato VARCHAR(50) NOT NULL
);
CREATE OR REPLACE TABLE sirve_para (
     isbn CHAR(13) NOT NULL,
     nombre VARCHAR(50) NOT NULL,
     CONSTRAINT PRIMARY KEY (isbn, nombre)
);
CREATE OR REPLACE TABLE asignatura (
     nombre VARCHAR(50) NOT NULL PRIMARY KEY
);
```

```
ALTER TABLE ha_escrito ADD CONSTRAINT FOREIGN KEY (dni) REFERENCES
autor(dni);
ALTER TABLE ha_escrito ADD CONSTRAINT FOREIGN KEY (isbn) REFERENCES
libro(isbn);
ALTER TABLE sirve_para ADD CONSTRAINT FOREIGN KEY (isbn) REFERENCES
libro(isbn);
ALTER TABLE sirve_para ADD CONSTRAINT FOREIGN KEY (nombre)
REFERENCES asignatura(nombre);
```

Tema 7
Realización de Consultas Simples

Tabla de Contenidos

1.- Introducción

En las unidades anteriores se ha descrito el medio de pasar de la especificación de un problema a un modelo físico implementado en un SGBD concreto. En esta unidad se describirá la forma de acceder a los datos almacenados en un SGBD utilizando las sentencias SQL adecuadas para ello.

Debido a su extensión esta unidad se ha dividido en dos partes, la primera dedicada a las consultas con una sola tabla y la segunda dedicada a las consultas que implican varias tablas.

2.- La sentencia SELECT

Las consultas en lenguaje SQL se realizan utilizando una única sentencia: `SELECT`. Esta sentencia es muy versátil e implementa gran cantidad de variaciones y opciones que se irán desgranando a lo largo de la unidad. La forma general de la sentencia `SELECT` es:

```
SELECT [cualificador] [columnas_expresiones]
```

```
FROM [tablas]
WHERE [condicion_filas]
GROUP BY [columnas_agrupacion]
HAVING [condicion_agrupada]
ORDER BY [columnas_ordenar]
```

donde:

- cualificador es una opción que indica el modo en que se devuelven las filas. Puede ser uno de:
  - `ALL`. Devuelve todas las filas que cumplen las condiciones de la consulta, incluyendo filas duplicadas. Es la opción por defecto si no indicamos ningún cualificador, por lo que si queremos filas duplicadas podemos poner `ALL` o no poner nada.
  - `DISTINCT`. Devuelve sólo filas con valores únicos. Este tipo de consulta es menos eficiente que la `ALL` porque el SGBD se tiene que asegurar de que se eliminan las filas duplicadas por lo que sólo hay que utilizarla si es realmente necesario.
- columnas_expresiones son las columnas que va a tener la relación resultado. Pueden ser columnas de tablas ya existentes o ser el resultado de un cálculo realizado sobre otras columnas utilizando expresiones.
- tablas son las tablas de las cuales se van a obtener las columnas anteriores. Puede ser una tabla o varias tablas combinadas de distintas formas.
- condicion_filas. Condición que debe cumplir cada fila para que los valores que contiene aparezca en el resultado. Se aplica a cada fila de las tablas y determina si la misma va a aparecer en el resultado o no.
- campos_agrupacion es una lista con las columnas sobre las que se va a agrupar.
- condicion_agrupada es una condición que se aplica a cada elemento de la tabla después de agrupar.
- columnas_ordenar es un criterio de ordenación de filas. Las filas del resultado aparecerán ordenadas por el criterio que se especifique en esta claúsula.

La sentencia `SELECT` realiza una consulta sobre una tabla o tablas de la base de datos y devuelve una tabla temporal como resultado. Esta tabla no existe realmente en la base de datos y se construye expresamente en el momento de ejecutar la consulta y sólo es válida hasta el momento en que se le indica al SGBD que ya no se va a utilizar más, momento en que se descarta. Asimismo no se pueden modificar los contenidos de esta tabla, sólo se puede leer pero no se puede modificar.

Se irán visitando las distintas claúsulas según vayan siendo necesarias para implementar consultas más complejas.

3.- Seleccionar las columnas en una consulta

La forma que se va a estudiar en esta parte es la referente a consultas sobre una sola tabla. En este caso la forma más simple de la sentencia `SELECT` es:

```
SELECT columnas FROM tabla
```

donde `columnas` es la especificación de las columnas que aparecerán en el resultado y `tabla` es la tabla desde la cual se va a realizar la consulta. En este caso, la operación que se realizaría sería muy similar a la operación de proyección del algebra relacional.

La especificación de las columnas es un poco más compleja, ya que se disponen de varias opciones. En general la especificación de columnas es una lista de especificadores de columna, separados por comas. Cada especificador puede producir una o más columnas en el resultado. No se garantiza el orden en que aparecerán las columnas en el resultado por lo que siempre hay que utilizar el nombre de columna para referirse a un dato en concreto en lugar de fiar la localización del dato a la posición que ocupa la columna entre el resto de ellas en el resultado.

Los especificadores de columnas pueden ser:

- **nombre de una columna**. Si se especifica un nombre de una columna, ésta aparecerá en el resultado con el mismo nombre.
- ***** **(asterisco)**. Un asterisco especifica que se desean incluir en el resultado TODAS las columnas de la tabla origen, con el nombre original.
- **expresión**. Una expresión puede utilizar los operadores aritméticos y funciones para realizar un cálculo. En el resultado aparecerá una columna con el resultado de ese cálculo. **La expresión se calcula para cada fila del resultado utilizando los valores de esa misma fila, esto es, no se pueden utilizar valores de otra fila para calcular una expresión**. Por ejemplo, si la expresión indica que se desea el resultado de sumar dos columnas, se sumarán, para cada fila, el contenido de esas columnas para dicha fila y el resultado aparecerá en la columna del resultado.

Ejemplos:

Supongamos que tenemos una tabla con la siguiente definición:

```
CREATE TABLE EMPLEADO (
     dni CHAR(9) NOT NULL,
     nombre VARCHAR(200) NOT NULL,
     sueldo_base NUMERIC(6,2) NOT NULL,
     complementos NUMERIC(6,2) NOT NULL,
     CONSTRAINT PRIMARY KEY (dni)
);
```

Y con los datos:

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramirez	2333,76	777,22

Para consultar los nombres y dni de los empleados se podría emplear la consulta:

```
SELECT nombre,dni FROM EMPLEADO;
```

que devolvería el resultado

nombre	dni
Jose Lopez	11111111A
Ana Sánchez	22222222B
Julia Ramirez	33333333C

Si se deseara consultar la tabla completa:

```
SELECT * FROM EMPLEADO;
```

con el resultado

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramirez	2333,76	777,22

Si se supone que el sueldo completo es el resultado de sumar el sueldo base y los complementos, se podría obtener una tabla con dni, nombre y sueldo completo utilizando la siguiente consulta:

```
SELECT dni, nombre, sueldo_base + complementos FROM EMPLEADO;
```

con el resultado:

dni	nombre	sueldo_base+complementos
11111111A	Jose Lopez	1809,00
22222222B	Ana Sánchez	1780,64
33333333C	Julia Ramirez	3110,98

Como se puede comprobar, la última consulta ha devuelto una columna con un nombre un poco especial, ya que proviene del resultado de una expresión y no proviene de una columna ya existente, por lo que no tiene un nombre. Por lo tanto el sistema "se inventa" un nombre que usualmente tiene que ver con la expresión utilizada. Si esta tabla va a ser utilizada para algo más que para un vistazo a los datos, probablemente sería conveniente dar a la columna un nombre a la vez más corto y más manejable y que represente mejor lo que realmente contiene. En este caso, el nombre más conveniente sería algo como `sueldo_completo`. Para renombrar una columna en el resultado se utiliza la expresión `AS nuevo_nombre`, donde `nuevo_nombre` es el nuevo nombre que se quiere dar a esa columna. Por lo tanto, se podría rehacer la consulta introduciendo el renombrado y quedaría:

```
SELECT dni, nombre, sueldo_base + complementos AS sueldo_completo
FROM EMPLEADO;
```

Y el resultado sería ahora:

dni	nombre	sueldo_completo
11111111A	Jose Lopez	1809,00
22222222B	Ana Sánchez	1780,64
33333333C	Julia Ramirez	3110,98

Es importante hacer notar que la palabra clave AS es opcional, por lo que la consulta anterior y la que se describe a continuación:

```
SELECT dni, nombre, sueldo_base + complementos sueldo_completo FROM
EMPLEADO;
```

serían equivalentes. Esta opcionalidad de AS puede producir sutiles errores si se olvida poner una coma en la lista de columnas, ya que el intérprete SQL puede deducir que lo que se quiere es renombrar una columna en lugar de consultar dos.

3.1.- Operadores aritméticos

Como se ha visto, se pueden utilizar operadores para calcular valores de columna sobre la marcha durante una consulta. Asimismo se pueden utilizar funciones, que realizan cálculos más complejos.

Los operadores aritméticos que se pueden utilizar son los habituales:

- `+` tanto para sumar dos números como para indicar que un número es positivo (en cuyo caso es opcional).
- `-` tanto para restar dos números como para indicar que un número es negativo.
- `*` para multiplicar dos números
- `/` para dividir dos números. Hay que tener cuidado cuando el contenido de un campo puede ser cero ya que el resultado puede ser un error que detenga la consulta. Esto depende del SGBD utilizado. En caso contrario se realiza la división. Si esta no es exacta el resultado tendrá decimales.

Estos son los operadores estándar. El SGBD MariaDB añade unos cuantos no estándar:

- `DIV`. División entera. Divide dos enteros y sólo devuelve la parte entera del cociente, sin decimales.
- `%` ó `MOD` (se pueden utilizar ambos indistintamente). Realiza la división entera entre dos argumentos enteros y devuelve el *resto* de la división.

En cuanto a las funciones, su número y disponibilidad dependen del SGBD utilizado aunque hay algunas que son estándar. Las veremos más adelante.

Ejemplo:

Supongamos la misma tabla de los ejemplos anteriores. Se nos dice que para calcular el sueldo neto del empleado es necesario sumar el sueldo base y los complementos y deducir un 15% de retenciones a cuenta del IRPF. La consulta que realizaría esto sería:

```
SELECT dni, nombre, (sueldo_base + complementos) * 0.85 AS sueldo_neto FROM EMPLEADO;
```

Al multiplicar por 0.85 se le está calculando el 85 % de la cantidad, lo que es lo mismo que restarle un 15%.

4.- Seleccionar las filas en una consulta

Para seleccionar las filas que devuelve una consulta se utilizan las clausulas `WHERE` y `HAVING`. El uso es muy similar pero por ahora sólo se utilizará la clausula `WHERE` ya que la `HAVING` sólo tiene sentido cuando se realiza agrupamiento de filas utilizando la clausula `GROUP BY`. Ambas se verán más adelante.

La clausula `WHERE` tiene como parámetro una condición sobre las columnas de la tabla. Es importante tener en cuenta que se puede utilizar cualquiera de las columnas de la tabla, aunque algunas de ellas no aparezcan en la lista de columnas del resultado.

La selección funciona de la siguiente manera. Se toman una a una las filas de la tabla y se le aplica la condición a los valores en cada una de ellas. Si la condición es cierta (devuelve `TRUE` o `VERDADERO`) la fila se incorpora al resultado. Si la condición no es cierta (devuelve `FALSE` o `FALSO`) la fila se descarta y no se incorporará al resultado.

Para realizar las condiciones se necesitan más operadores que los que se han discutido hasta ahora. En especial se necesitan los operadores relacionales y los operadores lógicos.

4.1.- Operadores relacionales

Los operadores relacionales o de comparación comparan dos valores para ver si cumplen una condición determinada. Si la cumplen devuelven `VERDADERO` o `TRUE`. Si no la cumplen devuelven `FALSO` o `FALSE`.

Los operadores relacionales son:

Operador	Nombre	Descripción
<	Menor que	Devuelve `VERDADERO` si el primer operando es menor que el segundo o `FALSO` en cualquier otro caso
<=	Menor o igual que	Devuelve `VERDADERO` si el primer operando es menor que el segundo o es igual al segundo o `FALSO` en cualquier otro caso.
>	Mayor que	Devuelve `VERDADERO` si el primer operando es mayor que el

		segundo o FALSO en cualquier otro caso.
>=	Mayor o igual que	Devuelve VERDADERO si el primer operando es mayor que el segundo o es igual al segundo.
=	Igual que	Devuelve VERDADERO si el primer operando tiene igual valor que el segundo o FALSO en cualquier otro caso.
<>	Distinto que	Devuelve VERDADERO si el valor del primer operando es distinto al del segundo o FALSO en cualquier otro caso

Los valores a comparar pueden ser directamente valores de columnas o también se pueden utilizar resultados de expresiones aritméticas o funciones.

4.2.- Operadores lógicos

Los operadores lógicos toman valores VERDADERO o FALSO y los combinan para producir otro valor VERDADERO o FALSO. Se utilizan para crear condiciones complejas en las que se tienen que dar varias condiciones distintas, ya sea a la vez o alternativamente. Los operadores lógicos son:

Operador	Nombre	Descripción
NOT	No	Toma un solo operando y devuelve VERDADERO si el operando es FALSO o FALSO si el operando es VERDADERO.
AND	Y	Toma dos operandos y devuelve VERDADERO si ambos tienen el valor VERDADERO. En cualquier otro caso devuelve FALSO.
OR	O	Toma dos operandos y devuelve VERDADERO si alguno de los dos (o los dos) es VERDADERO. Si ambos son FALSO devuelve FALSO.
XOR	O exclusivo	Toma dos operandos y devuelve VERDADERO sólo si uno de los dos es VERDADERO y el otro es FALSO. Si ambos son VERDADERO o ambos son FALSO devuelve FALSO.

4.3.- Precedencia de los operadores

Cuando se crea una expresión se pueden utilizar todos los operadores vistos hasta ahora para crearla: Aritméticos, relacionales y lógicos. Se pueden presentar problemas cuando en una expresión hay más de un operador pues no se sabe cual de las operaciones se realizará antes. En muchos casos esto no tiene importancia porque el resultado sería el mismo independientemente del orden en que se realicen las operaciones pero en otros casos distintos órdenes pueden producir distintos resultados. Por lo tanto es importante el conocer en qué orden se van a realizar las operaciones para obtener resultados predecibles.

Por ejemplo, si se tiene la expresión `1 + 2 * 3` ¿Cuando vale?. Si se supone que la suma se realiza primero la expresión valdría 9. Si se supone que la multiplicación se realiza primero, la operación valdría 7. Se puede ver, por tanto, que es necesario definir una regla que especifique qué orden se sigue al evaluar los operadores, de forma que no se produzcan errores de cálculo.

Respecto a los tipos de operadores, los operadores aritméticos se calculan antes que los relacionales y estos a su vez antes que los lógicos. Por lo tanto, si en una expresión hay mezclados operadores aritméticos, relacionales y lógicos, los operadores aritméticos se evalúan antes, después los relacionales y por último los lógicos.

Esto tiene sentido si se piensa con cuidado puesto que los operadores relacionales pueden utilizar los resultados de operadores aritméticos para sus propios cálculos, pero lo inverso no es aplicable. Por lo tanto tiene lógica que se realicen antes las operaciones aritméticas que las relacionales. Algo similar ocurre entre los operadores relacionales y los lógicos.

Pero, ¿qué ocurre si hay varios operadores del mismo tipo? Pues entonces hay que establecer un orden entre ellos. En el caso de los operadores aritméticos, el orden es:

a) `-` (cambio de signo)

b) `*`, `/`, `DIV`, `%`, `MOD`.

c) `+`, `-`

Si se encuentran varios operadores del mismo nivel (por ejemplo una suma y una resta) se realizan de izquierda a derecha.

Los operadores relacionales no tienen precedencia entre ellos porque no pueden mezclarse directamente, es decir, no se puede utilizar el resultado de un operador relacional como operando de otro operador relacional.

Los operadores lógicos tienen el siguiente orden:

a) `NOT`

b) `AND`

c) `XOR`

d) `OR`

Al igual que con los aritméticos, si se combinan varios del mismo nivel, se evalúan de izquierda a derecha.

Como recordar estas reglas puede ser difícil o peligroso, existe una forma de forzar una precedencia, es decir, de indicar que queremos que unas operaciones se realicen antes que otras. La forma de lograrlo es utilizando paréntesis.

Las expresiones que estén entre paréntesis se realizan antes que cualquier otra, sea cual sea su precedencia. Si hay paréntesis dentro de otros paréntesis, primero se realizan los cálculos de los paréntesis más internos, luego los siguientes hacia afuera hasta llegar a los más externos.

Es una buena práctica el utilizar paréntesis para indicar el orden de precedencia de los operadores aunque según las reglas anteriormente descritas se sepa que la expresión se evalúa correctamente. Así,

un lector ocasional o el mismo creador de la sentencia puede saber de un vistazo cómo se va a evaluar una expresión sin necesitar recurrir al manual o a otra documentación.

4.4.- Ejemplos de consultas

Una vez que se conocen las herramientas necesarias para crear condiciones se pueden realizar selecciones de filas más sofisticadas.

Si se tiene la misma tabla de los ejemplos anteriores:

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramirez	2333,76	777,22

Si se quiere acceder al empleado cuyo `DNI` es `11111111A`, se podría emplear la sentencia:

```
SELECT * FROM EMPLEADO WHERE dni = '11111111A';
```

Nótese que los valores constantes de cadena de texto se deben delimitar con comillas simples. El resultado sería:

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88

Si se quieren consultar los datos de todos los empleados excepto el que tiene el dni `11111111A`, la sentencia sería:

```
SELECT * FROM EMPLEADO WHERE dni <> '11111111A';
```

con el resultado:

dni	nombre	sueldo_base	complementos
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Si se quieren obtener los empleados que cobran un sueldo base de más de `1500` euros al mes, la consulta sería:

```
SELECT * FROM EMPLEADO WHERE sueldo_base > 1500;
```

con el resultado:

dni	nombre	sueldo_base	complementos
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Que el resultado sea idéntico al del ejemplo anterior es pura coincidencia.

Los empleados que cobran un sueldo total mayor que `2500` euros se pueden consultar con:

```
SELECT * FROM EMPLEADO WHERE (sueldo_base + complementos) > 2500;
```

dni	nombre	sueldo_base	complementos
33333333C	Julia Ramírez	2333,76	777,22

Para practicar lo visto hasta ahora, realiza la relación de ejercicios 01.

5.- Operadores avanzados de selección

5.1.- Operador BETWEEN

Un tipo de selección que se presenta muy a menudo es el de seleccionar filas en las que el valor de una columna está comprendido en un intervalo determinado. Por ejemplo, en la tabla:

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramirez	2333,76	777,22

Si se quiere seleccionar a los empleados que tienen un sueldo base de `1000` a `1500` euros se podría emplear la sentencia:

```
SELECT * FROM EMPLEADO WHERE sueldo_base <= 1000 AND sueldo_base >=1500;
```

Esta sentencia, aunque correcta es un poco engorrosa de crear y no queda clara a simple vista la intención de la condición. Es para este tipo de consultas para las que se creó el operador `BETWEEN` que es de la forma:

```
SELECT ..... WHERE expresion [NOT] BETWEEN valor_min AND valor_max
```

Para cada fila se evalúa `expresión`, que puede ser una columna o una expresión aritmética sobre una o más columnas y si el valor obtenido es mayor o igual a `valor_min` y a la vez es menor o igual a `valor_max`, devuelve `VERDADERO`. En caso contrario devuelve `FALSE`. Si se incluye la palabra opcional `NOT`, entonces devuelve justamente el valor opuesto.

Hay que tener en cuenta un detalle muy importante. El comportamiento que se ha dado es el estándar pero hay SGBDs que no incluyen los límites inferior y superior en el intervalo, esto es, si expresión vale exactamente igual a `valor_min` o `valor_max`, el valor devuelto no es `VERDADERO` sino `FALSO`. Otros motores incluyen los dos, otros incluyen uno si y otro no, por lo que hay que consultar la documentación del SGBD para saber como se comporta exactamente `BETWEEN` para ese SGBD. Por esta razón muchos usuarios de SGBDs prefieren utilizar la forma con `>=` y `<=` en lugar de `BETWEEN`, ya que de esta manera se sabe perfectamente el comportamiento que va a tener la consulta en cualquier SGBD.

Ejemplos:

Si se tiene la tabla anterior, la consulta para buscar a los empleados con sueldo base entre `1000` y `1500` quedaría

```
SELECT * FROM EMPLEADO WHERE sueldo_base BETWEEN 1000 AND 1500;
```

Si se quisiera localizar a los empleados que cobran en total de 1000 a 2000 euros:

```
SELECT * FROM EMPLEADO WHERE sueldo_base + complementos BETWEEN 1000 AND 2000;
```

Y, por último, si se quisiera localizar a los empleados que cobran menos de 1000 ó mas de 2000 euros:

```
SELECT * FROM EMPLEADO WHERE sueldo_base + complementos NOT BETWEEN 1000 AND 2000;
```

5.2.- Operador IN

Imaginemos que se quiere obtener al empleado cuyo DNI es `11111111A`. Para ello se emplearía la consulta:

```
SELECT * FROM EMPLEADOS WHERE dni = '11111111A';
```

Hasta ahora nada nuevo pero, ¿y si se quieren los empleados con DNI `11111111A` o `33333333C`? La respuesta podría ser:

```
SELECT * FROM EMPLEADOS WHERE dni = '11111111A' OR dni = '33333333C';
```

Conforme se vayan añadiendo empleados la lista se hace más larga y verbosa. SQL proporciona una manera abreviada de hacer esto utilizando el operador IN. Su sintaxis es:

```
SELECT ..... WHERE expresion [NOT] IN (valor1, valor2, ...., valorN)
```

Se evalúa el valor de expresión y se compara con los distintos valores contantes que aparecen en la lista después de `IN`. Si hay coincidencia con alguno de ellos, se devuelve `VERDADERO`. Si no hay coincidencia con ninguno, se devuelve `FALSE`. Si se quiere saber si un valor NO está en la lista, se utiliza la opción `NOT`, que devuelve justo lo contrario.

Ejemplos:

La consulta anterior de los DNIs `11111111A` o `33333333C` quedaría así utilizando `IN`:

```
SELECT * FROM EMPLEADOS WHERE dni IN ('11111111A', '33333333C');
```

Si se quisieran obtener todos los empleados excepto los que tienen como DNI `11111111A` ó `33333333C`, entonces se utilizaría:

```
SELECT * FROM EMPLEADOS WHERE dni NOT IN ('11111111A', '33333333C');
```

5.3.- Operador LIKE

El operador `LIKE` es un poco especial debido a que los nuevos operadores introducidos en esta sección (`BETWEEN` e `IN`) servían para expresar algo que ya se podía hacer pero de forma más sencilla o legible. El operador `LIKE`, sin embargo, sirve para realizar una función que hasta ahora no se podía

realizar: La búsqueda de patrones en columnas de tipo cadena. Se puede ver también como comparaciones de cadena pero que se hacen sobre una parte de la misma en lugar de la cadena completa, que son las comparaciones que se realizan con los operadores ya descritos (= y <>).

La sintaxis de `LIKE` es:

```
SELECT .... WHERE expresion [NOT] LIKE 'patron';
```

expresión puede ser cualquier expresión que devuelva una cadena y patron puede constar de:

- El caracter `%` (porcentaje). Este carácter equivale a cero o más caracteres cualesquiera.
- El caracter _ (subrayado). Este carácter equivale a un carácter cualquiera.
- Cualquier otro carácter. Exactamente el carácter proporcionado.

Como cabría esperar, si se añade la partícula `NOT`, la condición se invierte.

Si se desea encontrar de forma literal un carácter `%` ó _, o sea, que no actúe como comodín sino por su valor propio, se deben 'escapar', colocando delante un carácter `\` (backslash). Asimismo, el carácter backslash debe también 'escaparse' para poder ser utilizado (`\\`).

Por ejemplo:

Patrón	Cadenas que corresponden
`no%`	no, nos, norte, nostalgia, novedad, 'no veo el momento de llegar'
`%no`	no, verano, langostino, vacuno, 'vamos de camino'
`%no%`	no, esnob, noble, verano, 'ellos no vienen'
`no_`	nos, noe, nod
`_no`	ono, ano, vno
`_no_`	anoe, vnod, hnog
`no\%`	no%
`no\\`	no\
`no\_`	no_

Ejemplos:

Encontrar todos los empleados cuyo nombre es `Jose`:

```
SELECT * FROM EMPLEADOS WHERE nombre LIKE 'Jose%';
```

Encontrar los empleados para los que la segunda letra del nombre no es una '`a`':

```
SELECT * FROM EMPLEADOS WHERE nombre NOT LIKE '_a%';
```

6.- Manejo de valores nulos

En el modelo relacional se necesita que se pueda especificar que el contenido de una casilla de una tabla pueda estar vacío. Para ello se utiliza el valor especial NULL.

La adición de NULL complica ligeramente la creación de expresiones ya que hay que definir cómo se comportan los operadores cuando alguno de los operando es NULL, circunstancia que se puede presentar en muchas situaciones.

6.1.- Operadores artíméticos

Si cualquiera de los operandos de un operador aritmético es NULL, el resultado también es NULL.

6.2.- Operadores relacionales

Si cualquiera de los operandos de un operador relacional es NULL, el resultado también es NULL.

Además se añaden los siguientes operadores relacionales para tratar con valores nulos.

IS NULL	Es NULL	Tiene un solo operando y devuelve VERDADERO si el valor del mismo es NULL o FALSO en otro caso
IS NOT NULL	No es NULL	Tiene un solo operando y devuelve VERDADERO si el valor del mismo no es NULL o falso en otro caso.

La introducción de estos operadores es necesaria ya que la expresión

```
SELECT ... WHERE campo = NULL
```

siempre devuelve NULL (ni VERDADERO ni FALSO)

Por ejemplo, la consulta:

```
SELECT * FROM EMPLEADO WHERE nombre IS NULL;
```

devolvería aquellas filas de la tabla EMPLEADO en las que el campo nombre esté vacío (suponiendo que dicho campo permita valores vacíos).

6.3.- Operadores lógicos

Al igual que el resto de operadores, si alguno de los operandos en NULL, el resultado también lo será.

7.- Funciones

Al hablar de las expresiones se comentó que podrían constar tanto de operadores como de funciones. En esta sección se describirán algunas de las funciones existentes, ya que cada SGBD define su propio juego de funciones con su propia sintaxis y significado.

No obstante, hay algunas funciones que provienen del estándar SQL y que se implementan más o menos de forma similar en todos los SGBDs:

- CURRENT_DATE(). Devuelve la fecha actual, esto es, la fecha del sistema en el momento de realizar la consulta.

- `CURRENT_TIME()`. Igual que el anterior pero para la hora.
- `CURRENT_TIMESTAMP()`. Devuelve la fecha y hora actuales.
- `ABS(expr)`. Devuelve el valor absoluto de `expr`, que debe ser un valor numérico. El valor absoluto de un número es el mismo valor pero sin signo.
- `CHAR_LENGTH(expr)`. Devuelve un valor entero indicando la longitud (en caracteres) de la cadena dada en `expr`.
- `POSITION(cad1 IN cad2)`. Devuelve la posición (valor entero) en que aparece la cadena `cad1` dentro de la cadena `cad2`. Si `cad1` no aparece dentro de `cad2`, devuelve 0.
- `POWER(base, exponente)`. Devuelve el resultado de elevar `base` a `exponente`, esto es, realiza el cálculo de potencias.
- `SQRT(expr)`. Calcula la raiz cuadrada de `expr`, que tiene que ser una expresión numérica.
- `LOWER(expr)`. `expr` debe ser una cadena y devuelve la misma cadena expresada completamente en letras minúsculas.
- `UPPER(expr)`. Igual que `LOWER` pero pasa toda la cadena a mayúsculas.
- `SUBSTRING(cadena FROM comienzo [FOR longitud])`. Extrae una parte de `cadena`, comenzando por el carácter que ocupa la posición `comienzo`. Si se especifica `FOR` longitud, extrae `longitud` caracteres y si no se especifica extrae hasta el final de la cadena de origen. Algunos SGBD no soportan `FOR` y extraen siempre hasta el final de la cadena.

Además de estas funciones se tienen las que proporcione cada SGBD y las agregadas que se verán más adelante.

8.- Ordenar los resultados de una consulta

El modelo relacional no garantiza ningún orden tanto en las columnas como en las filas del resultado de una operación de consulta, por lo que no se pueden hacer suposiciones sobre en qué posición aparecerá cada resultado ni en el orden en que se presentarán los mismos. Si se desea especificar la ordenación, hay que especificar la clausula `ORDER BY`.

La sintaxis de la claúsula `ORDER BY` es:

```
ORDER BY espec1, espec2, ..., especN
```

donde `especX` es una especificación de ordenación. Se pueden dar más de una y el comportamiento es ordenar por la primera especificación. La segunda y siguientes se utilizan en caso de que haya ambigüedad al usar el primero. Por ejemplo, si en una tabla especificáramos que queremos ordenar por apellido primero y por nombre después, se ordenaría por apellido y se utilizaría la ordenacíón por nombre sólo en caso de que se encontrara mas de una fila con el mismo apellido.

La especificación de ordenación tiene dos partes:

- Indicador de columna. Indica sobre que columna, de entre las seleccionadas en la sentencia `SELECT`, se va a ordenar. La columna se puede indicar de tres formas:
  - Por nombre de columna.
  - Por alias (asignado utilizando `AS` alias)
  - Por posición, siendo 1 la posición de la primera columna seleccionada, 2 el de la segunda, etc.
- Indicador de dirección. Indica si la ordenación sobre esa columna se desea en orden ascendente (los menores primero) o descendente (los mayores primero). En el primer caso se indica mediante la palabra clave `ASC` o no indicando ninguna, ya que es la opción por defecto. En el segundo se emplearía la palabra `DESC`.

Ejemplos:

Si se quiere consultar todos los empleados, ordenados de menor a mayor sueldo base se podría emplear:

```
SELECT * FROM EMPLEADO ORDER BY sueldo_base;
```

Si sólo se desea saber el dni y nombre de los empleados, ordenados por nombre de forma descendente, se podría utilizar:

```
SELECT dni, nombre FROM EMPLEADO ORDER BY nombre DESC;
```

o bien

```
SELECT dni, nombre FROM EMPLEADO ORDER BY 2 DESC;
```

Si se quieren ordenar los empleados por sueldo base y, en caso de que los sueldos sean iguales, por nombre, se emplearía:

```
SELECT * FROM EMPLEADO ORDER BY sueldo_base, nombre;
```

Por supuesto se puede combinar selección de columnas con filas y ordenación:

```
SELECT dni, nombre, (sueldo_base + complementos) AS sueldo_total FROM
EMPLEADO WHERE (sueldo_base + complementos) > 1000 ORDER BY nombre,
sueldo_total;
```

Mostraría el dni y nombre de aquellos empleados cuyo sueldo total (base + complementos) es mayor de 1000 euros, ordenados por nombre y si coinciden estos, por sueldo total.

Para practicar lo visto hasta ahora, realiza la relación de ejercicios 02.

9.- Funciones de columna

En apartados anteriores se han descrito los operadores y funciones, pero todos tenían algo en común: todos operaban a nivel de fila, esto es, por cada fila de la tabla se realizaba el cálculo, que formaba a ser otra columna "virtual".

A veces, sin embargo, es conveniente realizar operaciones que impliquen los valores de una columna de la tabla, en lugar de fila a fila.

Por ejemplo, puede que interese realizar la suma de los importes de las facturas para saber lo que se ha facturado. Esto no se puede hacer con lo que se ha descrito hasta ahora sin utilizar un programa externo que consulte la base de datos y realice los cálculos por su cuenta. Sin embargo, probablemente el lector esté de acuerdo de que este tipo de cálculos, debido su relativa frecuencia, podrían ser realizados en el SGBD.

Para realizar este tipo de tareas se introdujeron las llamadas *funciones de columna* o *funciones agregadas*. Las funciones de columna son funciones cuyo ámbito de aplicación son un conjunto de valores, en lugar de un único valor. Si se aplica a una columna, realizará la operación indicada sobre todos los valores de dicha columna.

Hay que tener en cuenta, asimismo, que las funciones de columna sólo incluyen los valores seleccionados mediante la claúsula `WHERE`, es decir, si una consulta utiliza una función de columna sobre una columna determinada, sólo se realiza sobre los valores de esa columna correspondientes a filas que cumplan la condición dada en la claúsula `WHERE`, no sobre todas las filas de la tabla. Dicho en otras palabras: primero se realiza la selección de filas mediante `WHERE` y al resultado se le aplica la función de columna. Si se quiere hacer algún tipo de selección sobre los resultados de las funciones de columna, habría que utilizar la claúsula `HAVING` que se expondrá más adelante.

Las funciones de columna disponibles son:

- `AVG`. Calcula la media aritmética de los valores de la columna. El tipo de datos de la columna debe ser numérico.
- `COUNT`. Cuenta el número de valores no nulos en la columna. Es posible que hayan valores repetidos y cada uno cuenta.
- `MAX`. Devuelve el mayor valor de la columna, según el tipo de dato. En el caso de números es comparación simple. En el caso de las columnas de tipo texto depende del cotejamiento.
- `MIN`. Devuelve el menor valor de la columna, con las mismas consideraciones que MAX.
- `SUM`. Devuelve la suma de los valores de la columna, que debe ser de tipo numérico.

Ejemplo:

Siguiendo con el ejemplo de los empleados

dni	nombre	sueldo_base	complementos
11111111A	Jose Lopez	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramirez	2333,76	777,22

Si se quisiera calcular el salario neto (sueldo base mas complementos) medio de la empresa se podría utilizar:

```
SELECT AVG(sueldo_base + complementos) AS salario_medio FROM EMPLEADO;
```

Nótese el uso de AS para renombrar la columna de resultado a un nombre más parecido al de una columna "normal".

Si, en cambio, se deseara saber el número de empleados de la empresa:

```
SELECT COUNT(*) AS empleados FROM EMPLEADO;
```

O el numero de empleados cuyo salario neto es mayor de 1800 euros:

```
SELECT COUNT(*) AS empleados FROM EMPLEADO WHERE (sueldo_base + complementos) > 1800;
```

El mayor salario de la empresa:

```
SELECT MAX(sueldo_base + complementos) AS sueldo_maximo FROM EMPLEADO;
```

O el menor:

```
SELECT MIN(sueldo_base + complementos) AS sueldo_minimo FROM EMPLEADO;
```

El importe mensual de la nomina de la empresa se podría calcular fácilmente con:

```
SELECT SUM (sueldo_base + complementos) AS total_nomina FROM EMPLEADO;
```

Realizar la relación de ejercicios 03.

10.- Agrupación de filas

Las funciones de columna que se han visto en el apartado anterior son muy útiles pero en la forma que hemos descrito anteriormente están limitadas a realizar operaciones sobre toda la tabla, lo que limita un poco su uso.

Supongamos la siguiente tabla:

OFICINA					
oficina	**ciudad**	**region**	**dir**	**objetivo**	**venta**
11	Valencia	Este	106	575000	693000
12	Alicante	Este	104	800000	735000
13	Castellon	Este	105	350000	368000
21	Badajoz	Oeste	108	725000	836000
22	A Coruña	Oeste	108	300000	186000
23	Madrid	Centro	108	225000	195000
24	Madrid	Centro	108	250000	150000
26	Pamplona	Norte		0	
28	Valencia	Este	0	900000	700000

Con la sentencia:

```
SELECT SUM(venta) FROM OFICINA;
```

se obtendría el total de ventas de la empresa, el cual es un dato útil pero, ¿y si se desea el total de ventas *de cada ciudad*? Con lo visto hasta ahora no se podría realizar tal tarea puesto que las funciones de columna, que son las que realizan la tarea lo hacen sobre **toda** la columna. La solución es la *agrupación de filas*.

La agrupación de filas utiliza el valor de una columna de la tabla para crear grupos de filas, cada uno de ellos con las filas que tienen un valor distinto de la tabla. Por ejemplo, si se agrupara la tabla anterior por el nombre de la ciudad se formarían 6 grupos. Uno formado por las filas 1 y 9 (Valencia), otro por la fila 2 (Alicante), otro por la fila 3 (Castellon), otro por la fila 4 (Badajoz), otro por la fila 5 (A Coruña), otro por las filas 6 y 7 (Madrid) y, por último, otro con la fila 8 (Pamplona). Si se aplica una función de columna sobre una agregación, se realiza la operación indicada *por grupo* en lugar de hacerlo sobre toda la tabla, con lo que se consigue efectivamente el realizar funciones de columna sobre subconjuntos de la tabla.

La agrupación de filas también se puede realizar sobre más de una columna. En este caso los grupos estarán formados por las filas que tengan una combinación específica de los valores contenidos en las columnas especificadas.

Cuando se realiza agrupación de filas, en la sección de selección de columnas de la sentencia `SELECT` no puede aparecer cualquier nombre de columna sino o bien sólo aquellas por las que se agrupa o bien funciones de columna sobre las demás (las que no se agrupan). No puede aparecer una columna sobre la que no se agrupe si no es dentro de una función de columna.

La agrupación de filas se lleva a cabo mediante la claúsula `GROUP BY`, en la forma:

```
GROUP BY columna1, columna2, ..., columnaX
```

donde `columna1`, `columna2`, `columnaX` son las columnas sobre las que se desea agrupar.

Ejemplo:

En el caso de la tabla `OFICINAS`, vista anteriormente, si se desean conocer las ventas totales por ciudad, la consulta que lo realizaría sería:

```
SELECT ciudad, SUM(venta) AS ventas FROM OFICINA GROUP BY ciudad;
```

que produciría el siguiente resultado:

ciudad	ventas
Valencia	1393000
Alicante	735000
Castellon	368000
Badajoz	836000
A Coruña	186000
Madrid	300000
Pamplona	

Como se puede ver las filas correspondientes a Valencia y Madrid incluyen los totales calculados y segmentados. El resto de ciudades no se ve afectado porque están solas en sus respectivos grupos por lo que la suma no hace nada.

10.1.- Selección de filas agrupadas

La clausula `GROUP BY` realiza agrupación de columnas y produce datos resumidos, ideales para informes o resúmenes.

Desgraciadamente, con lo que se ha visto hasta ahora es imposible seleccionar *en el resumen* aquellas filas que cumplan un criterio determinado. Esto es asi porque la clausula WHERE selecciona las filas *antes* de realizar la agrupación, por lo que no afecta a las filas agrupadas.

Para proporcionar esta funcionalidad se introduce la claúsula `HAVING` con la siguiente sintaxis:

```
HAVING expresion
```

donde `expresion` es una expresión sobre las columnas del resumen que se evalúa por cada fila del resumen. Si al aplicar la expresión sobre la fila se obtiene un valor `TRUE` o `VERDADERO`, la fila se incorporará al resultado. Si no lo obtiene la fila *no* se incorporará al resultado.

Ejemplos:

Si se desease conocer las ventas totales por ciudad, ya se ha visto la consulta que habría que realizar en el ejemplo del apartado anterior pero, ¿Y si se quisiera obtener las ventas totales para las oficinas que han vendido más de 500000 euros?. La solución sería utilizar la clausula `HAVING`:

```
SELECT ciudad, SUM(venta) AS ventas FROM OFICINA GROUP BY ciudad
HAVING ventas > 500000;
```

Realizar la relación 4 de ejercicios.

11.- Resumen

En esta primera parte de la unidad se ha introducido la sintaxis de SQL para realizar consultas sobre una sola tabla incluyendo:

- Selección de columnas.
- Selección de filas.
- Uso de expresiones matemáticas para calcular datos a partir de las columnas almacenadas.
- Ordenar los resultados.
- Realizar resúmenes sobre una o más columnas.
- Realizar resúmenes agrupando filas por valores de una o más columnas.
- Seleccionar filas de los resúmenes.

Tema 8
Consultas Compuestas y Vistas

Tabla de Contenidos

1.- Introducción

En la unidad anterior se ha introducido la sintaxis básica de la sentencia `SELECT` para realizar consultas sobre una sola tabla.

De esta forma se pueden realizar consultas bastante elaboradas pero no permite utilizar datos que están relacionados entre tablas, con lo que se pierde gran parte de la potencia del modelo relacional.

En esta unidad se describirá el uso de `SELECT` para combinar información proveniente de varias tablas, así como la forma de almacenar consultas utilizando el mecanismo de las vistas.

2.- Producto cartesiano

Para realizar el producto cartesiano de dos o más tablas sólo hay que indicar las tablas que van a participar en la clausula `FROM`, separadas por comas.

Si se tiene la siguiente base de datos:

empleado		
nss	**nombre**	**puesto**
111	Juan Pérez	Jefe de Área
222	Jose Sánchez	Administrativo
333	Ana Díaz	Administrativo

email	
nss	**email**
111	jefe2@ecn.es
111	juanp@ecn.es
222	jsanchez@ecn.es
333	adiaz@ecn.es
333	ana32@ecn.es

puesto	
puesto	**salario**
Administrativo	1500
Jefe de Área	3000

y se realiza la consulta:

```
SELECT * FROM empleado, puesto;
```

se obtendría el siguiente resultado:

nss	nombre	puesto	puesto	salario
111	Juan Pérez	Jefe de Área	Administrativo	1500
111	Juan Pérez	Jefe de Área	Jefe de Área	3000
222	Jose Sánchez	Administrativo	Administrativo	1500
222	Jose Sánchez	Administrativo	Jefe de Área	3000
333	Ana Díaz	Administrativo	Administrativo	1500
333	Ana Díaz	Administrativo	Jefe de Área	3000

como se puede ver hay dos columnas con el nombre `puesto`, lo cual puede dar lugar a confusiones. Para distinguir entre varias columnas con el mismo nombre en diferentes tablas se utiliza la notación `tabla.columna` para referirse a una columna sin ambigüedades.

Por ejemplo, la consulta anterior se podría hacer de la siguiente forma para que sólo se muestre la columna puesto proveniente de `empleado`:

```
SELECT empleado.*, puesto.salario FROM empleado, puesto;
```

que produciría el siguiente resultado:

nss	nombre	puesto	salario
111	Juan Pérez	Jefe de Área	3000
222	José Sánchez	Administrativo	3000
333	Ana Díaz	Administrativo	3000
111	Juan Pérez	Jefe de Área	1500
222	José Sánchez	Administrativo	1500
333	Ana Díaz	Administrativo	1500

Tal y como se puede ver, el producto cartesiano en si no es una operación muy útil a la hora de obtener datos relacionados entre distintas tablas.

Es por eso por lo que se necesitan operaciones más útiles, como las combinaciones o join.

3.- Combinación natural (INNER JOIN)

La combinación natural o `INNER JOIN` se define como el producto cartesiano de varias tablas del que se eliminan aquellas filas en las que no coinciden los valores de los campos que sirven para relacionar las tablas (claves primarias y ajenas).

Por ejemplo, en el caso de la consulta anterior:

```
SELECT * FROM empleado, puesto;
```

se obtenía el siguiente resultado:

nss	nombre	puesto	puesto	salario
111	Juan Pérez	Jefe de Área	Jefe de Área	3000
222	José Sánchez	Administrativo	Jefe de Área	3000
333	Ana Díaz	Administrativo	Jefe de Área	3000
111	Juan Pérez	Jefe de Área	Administrativo	1500
222	José Sánchez	Administrativo	Administrativo	1500
333	Ana Díaz	Administrativo	Administrativo	1500

Como se puede ver, la columna puesto relaciona ambas tablas y, por lo tanto, en el resultado anterior sólo tienen significado aquellas filas en las que los valores de ambas columnas puesto coinciden, lo que daría el siguiente resultado:

nss	nombre	puesto	puesto	salario
111	Juan Pérez	Jefe de Área	Jefe de Área	3000
222	José Sánchez	Administrativo	Administrativo	1500
333	Ana Díaz	Administrativo	Administrativo	1500

Es evidente que el dato salario de cada fila ahora es coherente con el resto de datos del empleado.

Hay distintas sintaxis para conseguir una combinación natural.

3.1.- Combinación natural con clausula WHERE

En este caso se realiza un producto cartesiano y en la clausula `WHERE` se incluye una condición para dejar sólo las filas en las que los campos clave coincidan:

Ejemplo:

Siguiendo con el ejemplo anterior:

```
SELECT * FROM empleado, puesto WHERE empleado.puesto =
puesto.puesto;
```

3.2.- Combinación natural con clausula `INNER JOIN` y condición

Ahora se introduce una nueva clausula (`INNER JOIN`) que sirve exclusivamente para realizar este tipo de combinaciones. La clausula se introduce dentro de la clausula `FROM` con la siguiente sintaxis:

```
tabla1 [INNER] JOIN tabla2 ON condicion;
```

donde:

- `tabla1` y `tabla2` son las dos tablas a combinar.
- `condicion` es la condición que deben cumplir las filas que pasarán al resultado.

La palabra `INNER` es opcional, pudiéndose utilizar únicamente `JOIN`.

Ejemplos:

Siguiendo con el ejemplo de los apartados anteriores, con esta sintaxis quedaría:

```
SELECT * FROM empleado INNER JOIN puesto ON empleado.puesto =
puesto.puesto;
```

O bien:

```
SELECT * FROM empleado JOIN puesto ON empleado.puesto =
puesto.puesto;
```

3.3.- Combinación natural con clausula `INNER JOIN` y claves con el mismo nombre

En el caso de que la(s) columna(s) clave que une(n) las tablas **tenga el mismo nombre en ambas tablas**, se puede utilizar una sintaxis alternativa y abreviada de la clausula `INNER JOIN`:

```
tabla1 [INNER] JOIN tabla2 USING (columnas_clave);
```

donde todo es igual que en el apartado anterior excepto `columnas_clave`. Tanto en `tabla1` como en `tabla2` debe haber unas columnas con esos nombres y son sobre las que se realizará la combinación. Si hay más de una columna se deberán separar por comas.

Un efecto adicional del uso de esta sintaxis es que la clave por la que se realiza la combinación no aparecerá dos veces en el resultado, como con las anteriores, sino que sólo aparecerá una vez, de forma similar al resultado de la operación en álgebra relacional.

Ejemplos:

Siguiendo con el ejemplo:

```
SELECT * FROM empleado INNER JOIN puesto USING(puesto);
```

O bien:

```
SELECT * FROM empleado JOIN puesto USING(puesto);
```

NOTA: el `puesto` que aparece entre los paréntesis se refiere al nombre del campo, no de la tabla.

3.4.- Combinación natural con clausula `NATURAL JOIN` y claves con el mismo nombre

En el caso de que la columna clave que une las tablas tenga el mismo nombre en ambas tablas, y sean las únicas columnas de entre las dos tablas cuyos nombres coincidan, se puede utilizar una sintaxis alternativa y mucho más abreviada de la clausula `INNER JOIN`:

```
tabla1 NATURAL JOIN tabla2;
```

donde todo es igual que en el apartado anterior excepto que desaparece cualquier mención de las columnas clave. Se examinan ambas tablas y si hay alguna coincidencia entre los nombres de alguna columna en ambas tablas, se realiza automáticamente un `USING` sobre dichas columnas. Si hay más de una columna cuyos nombres coincidan, se realiza la unión sobre todas ellas simultáneamente.

Al igual que con `USING`, un efecto adicional del uso de esta sintaxis es que la(s) columna(s) por la(s) que se realiza la combinación no aparecerá(n) dos veces en el resultado.

Ejemplos:

Siguiendo con el ejemplo:

```
SELECT * FROM empleado NATURAL JOIN puesto;
```

Ya que la única columna cuyo nombre coincide en ambas tablas es `puesto`, se realiza la unión por ésta.

4.- Combinación externa (`OUTER JOIN`)

La combinación natural debe su nombre a que es la forma natural de combinar tablas pero cuando se presentan campos con valor nulo en las claves, las filas que los contienen no aparecen en los resultados. Usualmente, este es el comportamiento deseado pero a veces se desea que aparezcan en el resultado de una combinación filas que no participan en la misma porque contienen valores nulos en las claves de relación. La combinación que produce estos resultados es la combinación externa (`OUTER JOIN`).

En la combinación externa u `OUTER JOIN`, siempre hay una tabla maestra y una subordinada. En el resultado aparecerán **TODAS las filas de la tabla maestra** y las filas correspondientes de la tabla subordinada con el mismo valor del campo clave. Si una fila de la tabla maestra contiene `NULL` en el valor clave o no se encuentra una coincidencia del valor de su campo clave con el campo

correspondiente en la tabla subordinada, las columnas correspondientes a la tabla subordinada aparecerán todos con valor `NULL`.

Hay dos tipos de combinación externa según las posiciones relativas de las tablas maestra y subordinada: combinación externa izquierda o combinación izquierda derecha. Realmente sólo haría falta una de las dos puesto que son la misma operación y sólo cambia el orden en que aparecen las tablas.

4.1.- Combinación externa izquierda

Se incluiría dentro de la clausula `FROM` con la siguiente sintaxis:

```
tabla1 LEFT [OUTER] JOIN tabla2 ON condicion
```

donde:

- `tabla1` es la tabla maestra. Todas las filas de esta tabla aparecerán en el resultado.
- `tabla2` es la tabla subordinada. En el resultado aparecerán las filas de esta tabla relacionadas con valores de la tabla maestra.
- `condicion` es la condición que se debe cumplir para que las filas se unan.

Una sintaxis alternativa sería:

```
tabla1 LEFT [OUTER] JOIN tabla2 USING (columnas)
```

donde `columnas` es una lista de nombres de columna separados por comas. Ambas tablas deben tener columnas con esos nombres y se mostrarán en el resultado todas las filas de `tabla1` y las que tengan los mismos valores en los campos dados en la `tabla2`. Si no hay ninguna con los mismo valores, los campos de `tabla2` en el resultado tendrán el valor `NULL`.

Ejemplo:

Supongamos las tablas:

empleado

numempl	nombre	oficina
101	Antonio Viguer	12
102	Alvaro Jaumes	21
103	Juan Rovira	12
104	Jose Gonzalez	NULL
105	Vicente Pantall	13
106	Luis Antonio	NULL
107	Jorge Gutierrez	22
108	Ana Bustamante	NULL
109	Maria Sunta	11
110	Juan Victor	15

oficina

oficina	ciudad
11	Valencia
12	Alicante
13	Castellon
21	Badajoz
22	A Coruña
23	Madrid
24	Madrid
26	Pamplona
28	Valencia

Si se hace una combinación externa izquierda entre `empleado` y `oficina`, utilizando:

```
SELECT * FROM empleado LEFT JOIN oficina USING (oficina);
```

proporcionaría el resultado:

numempl	nombre	oficina	ciudad
101	Antonio Viguer	12	Alicante
102	Alvaro Jaumes	21	Badajoz
103	Juan Rovira	12	Alicante
104	Jose Gonzalez	NULL	NULL
105	Vicente Pantall	13	Castellon
106	Luis Antonio	NULL	NULL
107	Jorge Gutierrez	22	A Coruña
108	Ana Bustamante	NULL	NULL
109	Maria Sunta	11	Valencia
110	Juan Victor	15	NULL

4.2.- Combinación externa derecha

La combinación externa derecha es idéntica a la izquierda pero con la sintaxis:

```
tabla1 RIGHT [OUTER] JOIN tabla2 ON condicion
```

o

```
tabla1 RIGHT [OUTER] JOIN tabla2 USING (columnas)
```

El significado es exactamente el mismo que el de la combinación izquierda pero cambiando los papeles de `tabla1`, que ahora es la tabla subordinada, y `tabla2`, que ahora es la maestra. De hecho, a fin de evitar confusiones se recomiendo realizar siempre combinaciones izquierdas.

5.- Combinaciones reflexivas

En todas las combinaciones que se han realizado hasta ahora se han utilizado dos tablas. Sin embargo ocurren casos en que las combinaciones hay que realizarlas entre una tabla consigo misma debido a una relación reflexiva.

Para ilustrar la situación se presenta la siguiente tabla:

persona					
dni	nombre	apellidos	direccion	dnipadre	dnimadre
11111111A	Juan	Gallardo	C/ Ventura, 23	NULL	NULL
22222222B	Lucia	Sanchez	C/ Ventura, 23	NULL	NULL
33333333C	Santiago	Gallardo	C/ Pozo,16	11111111A	22222222B

Esta tabla presenta dos relaciones consigo misma. La primera, a partir de la columna `dnipadre`, refleja la relación de una persona con otra que es su padre. Si el padre no está registrado, el valor de

`dnipadre` sería `NULL`. La otra relación es idéntica pero referente a la madre y se realiza a partir del atributo `dnimadre`.

Si ahora se quisiera consultar el nombre de la persona con `dni` 33333333C, la consulta sería:

```
SELECT nombre FROM persona WHERE dni='33333333C';
```

que devolvería:

nombre
Santiago

Hasta aqui ninguna novedad pero, ¿qué pasaría si además del nombre de la persona, se quisiera conocer el nombre del padre (o de la madre)? ¿Cómo debería ser la consulta? Una primera (apresurada y errónea) contestación podría ser:

```
SELECT persona.nombre, persona.nombre FROM persona INNER JOIN
persona ON persona.dnipadre = persona.dni WHERE
persona.dni='33333333C';
```

El problema es que esto no funcionaría por varias razones:

- ¿A qué columna se refiere con `persona.nombre`? ¿A la de la persona o a la del padre?
- En la clausula `ON persona.dnipadre = persona.dni`, ¿a cual de las dos tablas se refiere `dnipadre`? y ¿Y `dni`?
- En la clausula `WHERE`, ¿a qué tabla se refiere `persona.dni`?

Para evitar este tipo de problemas se introduce el *alias* de tabla. El alias de tabla se incluye en la claúsula `FROM` y sirve para dar un nuevo nombre para alguna tabla de las que allí aparecen. Dicha tabla se podrá referenciar con su nuevo nombre o alias en el resto de clausulas de la sentencia `SELECT`. Esta funcionalidad no sólo se puede utilizar con consultas con relaciones reflexivas sino que se puede utilizar cuando se desee. La única consideración es que, en caso de que se quieran realizar combinaciones entre una tabla consigo misma, este renombrado no sólo es conveniente sino que es necesario.

Ahora, la consulta "imposible" anterior quedaría como:

```
SELECT persona.nombre, padre.nombre AS nombrePadre FROM persona
INNER JOIN persona AS padre ON persona.dnipadre = padre.dni WHERE
persona.dni='33333333C';
```

Ahora si queda perfectamente claro a qué columnas se refieren tanto la lista de selección como las clausulas `FROM` y `WHERE`, y se tendría el resultado:

nombre	nombrePadre
Santiago	Juan

6.- Unión

Las uniones permiten combinar datos de varias consultas en una sola. La sintaxis es:

```
SELECT columnas1 [resto_select1]
UNION
SELECT columnas2 [resto_select2]
UNION
SELECT columnas3 [resto_select3]
UNION
…
UNION
SELECT columnasX [resto_selectX];
```

donde:

- `columnasN` son las columnas que se seleccionan en cada `SELECT`
- `resto_selectN` son las clausulas `SELECT` usuales (`WHERE`, `ORDER BY`, etc.)

Los distintos juegos de columnas (`columnas1`, `columnas2`, etc.) deben tener el mismo número de campos y los campos que ocupen la misma posición en cada uno de ellos deben ser del mismo tipo. En caso contrario se producirá un error.

Además, las filas duplicadas se eliminarán del resultado.

Ejemplo:

Si se tienen las tablas:

empleado		
dni	**nombre**	**fecha_nac**
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Lalo Sánchez	06/08/1986
55555555K	Pitita Gendres	01/02/1954

propietario		
dni	**fec_nac**	**nombre_completo**
55555555K	01/02/1954	Pitita Gendres
66666666L	02/02/1964	Hector Dinerin

La consulta:

```
SELECT dni, nombre, fecha_nac FROM empleado
UNION
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

produciría el resultado:

dni	nombre	fecha_nac
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Lalo Sánchez	06/08/1986
55555555K	Pitita Gendres	01/02/1954
66666666L	Hector Dinerin	02/02/1964

Se puede comprobar que sólo hay una fila para el `dni` '55555555K' a pesar de que aparece en las dos tablas.

7.- Intersección

La intersección permite seleccionar sólo las filas que están en dos resultados. Su sintaxis es:

```
SELECT columnas1 [resto_select]
INTERSECT
SELECT columnas2 [resto_select];
```

donde `columnas1` y `columnas2` deben cumplir las mismas condiciones que para las uniones.

Ejemplos:

Si se toman las mismas tablas de ejemplo que para la unión, la sentencia:

```
SELECT dni, nombre, fecha_nac FROM empleado
INTERSECT
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

produciría

dni	nombre	fecha_nac
55555555K	Pitita Gendres	01/02/1954

8.- Diferencia

La diferencia entre dos tablas devuelve las filas que están en la primera tabla pero que no están en la segunda. Su sintaxis es:

```
SELECT columnas1 [resto_select]
MINUS
SELECT columnas2 [resto_select]
```

donde `columnas1` y `columnas2` deben cumplir las mismas condiciones que para las uniones o intersecciones.

Ejemplos:

Si se siguen utilizando las mismas tablas, la sentencia:

```
SELECT dni, nombre, fecha_nac FROM empleado
MINUS
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

devolvería

dni	nombre	fecha_nac
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Lalo Sánche	06/08/1986

NOTA: En MariaDB el operador `MINUS` se escribe como `EXCEPT`.

Realizar los ejercicios de la relación 01.

9.- Subconsultas

Una subconsulta es una consulta que aparece dentro de otra consulta.

Las subconsultas proporcionan un mecanismo muy potente para realizar consultas que, de otra forma, serían muy difíciles o imposibles de realizar.

Una subconsulta debe aparecer siempre entre paréntesis y tiene la misma sintaxis que una consulta `SELECT` normal con alguna limitación, principalmente que no se puede utilizar clausulas `ORDER BY` sin limitar a 1 el número de filas devueltas.

Para limitar el número de filas devueltas se utiliza la clausula `TOP`, justo después de `SELECT` y con un número que indica el número de filas a devolver. MariaDB utiliza la clausula `LIMIT`, con la misma funcionalidad pero después de `ORDER BY`.

Una subconsulta puede anidarse en la clausula `WHERE` o `HAVING` de otra consulta o bien en otra subconsulta, de forma que las consultas se pueden anidar unas dentro de otras como muñecas Matroska. Hay que tener en cuenta que un excesivo anidamiento puede penalizar fuertemente el rendimiento de la consulta en su conjunto, por lo que no se deben anidar a la ligera. La consulta más externa se denomina consulta externa y las subconsultas, consultas internas.

Los ejemplos siguientes utilizan la base de datos `empleados`, utilizada para las relaciones de ejercicios.

Un ejemplo de consulta con subconsultas, sería, para listar los empleados que no han cubierto aún su cuota de ventas, la siguiente:

```
SELECT nombre FROM empleado WHERE cuota >= (SELECT SUM(importe) FROM
pedido WHERE rep = numempl);
```

Por cada fila de la tabla de `EMPLEADO` (consulta externa) se calcula la subconsulta y se evalúa la condición (`<=`). Como se puede ver, hay que realizar una consulta *por cada fila* de la tabla `empleado`. Una opción muy potente, pero que puede ralentizar el rendimiento.

Las consultas que utilizan subconsultas se pueden realizar en la mayoría de los casos utilizando otros métodos, pero, como se puede ver arriba, son más fáciles de seguir e interpretar que las consultas que utilizan composición y, en los nuevos SGBDs su eficiencia está a la par.

Hay que tener en cuenta que no se pueden utilizar columnas de la subconsulta en el resultado de la consulta externa.

En las subconsultas puede aparecer lo que se denomina *referencia externa*. A menudo es necesario acceder, desde dentro de una subconsulta, al valor de una columna de la fila actual en la consulta externa. La referencia a la columna de la consulta externa desde el interior de la subconsulta se denomina referencia externa.

En el ejemplo anterior, dentro de la subconsulta se hace referencia a la columna `numemp`. Esta es una referencia externa, ya que dicha columna no está entre las columnas de la(s) tabla(s) que aparecen en la clausula `FROM` de la subconsulta. Cuando aparece un nombre de columna dentro de una subconsulta se busca primero entre los nombres de columna de las que forman parte de las tablas en la clausula `FROM` de la subconsulta. Si no aparece alli, intenta buscarla en la consulta externa. Si se encuentra allí, se considera como una referencia externa. Si tampoco se encuentra allí, se busca la siguiente consulta externa, hasta llegar a la más externa. Si se sigue sin encontrar es un error.

Cuando se encuentra una referencia externa, su aparición se sustituye por el valor que tiene en la fila que se está procesando en cada momento en la consulta externa y se ejecuta la subconsulta como si la columna fuera un valor constante. Por ejemplo, en la consulta anterior, para cada fila se obtiene el valor de `numemp` y se coloca en el lugar que ocupa el nombre de la columna en la subconsulta. Acto seguido se ejecuta la subconsulta y el resultado se utiliza para compararlo con el valor de cuota *de esa misma fila*.

Según su forma de funcionar, hay cuatro tipos de subconsultas:

- Subconsultas que devuelven un único valor.
- Subconsultas que devuelven **una** columna con varias filas.
- Subconsultas que devuelven tablas.
- Subconsultas que aparecen en la clausula `FROM` (Tablas derivadas).

9.1.- Subconsultas que devuelven un único valor

Este tipo de subconsultas debe devolver un único valor. Se pueden utilizar para:

- Incluir un valor proveniente de una subconsulta en la lista de selección de la consulta externa.
- Como operando de un operador relacional (`<`, `>`, `<=`, `>=`, `=`, `<>`) en la clausula `WHERE` o `HAVING` de la consulta externa.

En este caso, la subconsulta podrá tener ***sólo una columna en la lista de selección*** y devolver una única fila como mucho. El valor único que se devuelva se utilizará para:

- Incluir ese valor en la fila del resultado correspondiente
- Realizar la comparación con otro valor.

Si la subconsulta no devolviera ninguna fila, se considerará que devuelve `NULL`.

Si la subconsulta devolviera más de una fila, se producirá un error.

Ejemplo:

La consulta que se utilizó como ejemplo en el apartado anterior,

```
SELECT nombre FROM empleado WHERE cuota >= (SELECT SUM(importe) FROM
pedido WHERE rep = numempl);
```

es una consulta con subconsulta de resultado único.

Realizar ejercicios de la relación 02.

9.2.- Subconsultas que devuelven una columna con varias filas

En el caso de subconsultas que devuelven una única columna con varias filas (que pueden ser cero, una o varias), se toma el resultado de la subconsulta como un *conjunto* de valores, que se puede utilizar en la consulta externa.

Este tipo de subconsultas se puede utilizar para realizar una comparación *extendida*. Una comparación extendida es una extensión de los operadores relacionales (<, >, <=, >=, =, <>) para comparar un valor con un conjunto de valores. Hay dos tipos de extensiones, parcial y total.

9.2.1.- Comparación extendida parcial

La comparación extendida parcial tiene la sintaxis:

```
operando operador ANY (subconsulta)
```

donde:

- `operando` es el valor que se va a comparar
- `operador` es un operador relacional
- `subconsulta` es una subconsulta que devuelve una única columna con varias filas.

El operador se calcula entre `operando` y todos los valores devueltos por la subconsulta. Si algún resultado es `VERDADERO` el operador devuelve también `VERDADERO`. Si ningún resultado es `VERDADERO` devuelve `FALSO`.

Hay que tener en cuenta que si la subconsulta no devuelve ninguna fila, el operador devolverá `FALSO`, no `NULL`. Para devolver `NULL`, *todos* los valores devueltos por la subconsulta deben ser `NULL`.

Se puede utilizar la palabra reservada `SOME` en lugar de `ANY`, pero no se recomienda ya que es poco usada.

Hay que hacer notar también que se puede sustituir `= ANY` con `IN`. De hecho, es la forma más utilizada, por lo que se recomienda seguirla.

Ejemplo:

Para consultar los empleados de oficinas del Este:

```
SELECT * FROM empleado WHERE oficina = ANY (SELECT oficina FROM
oficina WHERE region = ‘Este’);
```

o, utilizando IN

```
SELECT * FROM empleado WHERE oficina IN (SELECT oficina FROM oficina
WHERE region = ‘Este’);
```

9.2.2.- Comparación extendida total

La comparación extendida total tiene la sintaxis:

```
operando operador ALL (subconsulta)
```

donde `operando`, `operador` y `subconsulta` tienen el mismo significado que en la comparación parcial.

El `operador` se calcula entre `operando` y todos los valores devueltos por la subconsulta. Si todos los resultados del operador son `VERDADERO`, el operador extendido devuelve `VERDADERO`. Si alguno devuelve `FALSO`, el operador extendido devuelve `FALSO`. Si la subconsulta devuelve un conjunto vacío de datos, el operador extendido devuelve `VERDADERO`.

Al igual que con la comparación parcial, se puede sustituir `<> ALL` como `NOT IN`. de hecho, también es la forma más utilizada.

Ejemplos:

Si se quiere obtener los empleados de las oficinas que NO son del Este, la consulta podría ser:

```
SELECT * FROM empleado WHERE oficina <> ALL (SELECT oficina FROM
oficina WHERE region = ‘Este’);
```

o bien

```
SELECT * FROM empleado WHERE oficina NOT IN (SELECT oficina FROM
oficina WHERE region = ‘Este’);
```

Realizar la relación de ejercicios número 03.

9.3.- Subconsultas que devuelve cualquier numero de filas y columnas

En el caso de las subconsultas que devuelven cualquier numero de filas y columnas, su utilidad es la de determinar si la consulta ha devuelto algo o no, pero no se pueden utilizar los valores obtenidos. Para ello se utiliza la clausula `EXISTS` con la siguiente sintaxis:

```
EXISTS (subconsulta)
```

`subconsulta` puede ser cualquier consulta que devuelve una o más columnas.

La subconsulta se evalúa. Si devuelve alguna fila `EXISTS` devuelve `VERDADERO`. Si no devuelve ninguna `EXISTS` devuelve `FALSO`. Se puede preceder a `EXISTS` por el operador `NOT` para cambiar el sentido de la comparación.

Ejemplo:

Si se quieren obtener los empleados que tienen algún pedido del fabricante ACI, la consulta se podría hacer de la siguiente forma:

```
SELECT * FROM empleado WHERE EXISTS (SELECT * FROM pedido WHERE
numempl = rep AND fab = 'ACI');
```

Cuando se utiliza el operador `EXISTS` será necesario incluir una referencia externa, esto es, el resultado de la subconsulta deberá depender de la fila que se esté procesando en la consulta externa. Si no es el caso, la subconsulta siempre devolverá el mismo resultado por lo que, o bien será siempre `VERDADERO` o siempre `FALSO`, lo cual no será lo deseado en ningún caso.

Otro punto a considerar es que como sólo se valora la existencia o no de resultados, sin que importen las columnas que tengan estos, lo usual es utilizar `*` como lista de selección de columnas.

Realizar la relación de ejercicios número 04.

9.4.- Tablas derivadas

Las subconsultas se pueden utilizar también para “crear” tablas que se pueden utilizar en la consulta externa como si fueran tablas “reales” de la base de datos. Esto es lo que se conoce por *tablas derivadas*.

Para poder utilizar las tablas derivadas las subconsultas deben aparecer dentro de la clausula `FROM` de la consulta externa, en la forma:

```
SELECT lista_de_campos FROM (subconsulta) AS nombre_tabla
```

La “tabla” asi creada se puede utilizar como si fuera una tabla real en la consulta externa. Cuando se utilizan de este modo, las subconsultas tienen diferencias con lo que se ha visto hasta ahora:

1. La subconsulta **no** se realiza varias veces sino que sólo se realiza una vez. Por lo tanto **no** pueden utilizar valores de campos que no estén en la lista `FROM` de la subconsulta.

2. Es obligatorio el dar un nombre a la tabla derivada utilizando la clausula `AS` después de la subconsulta.
3. **Todos las columnas de la subconsulta** deben tener un nombre válido.

Si se cumplen todas las condiciones necesarias, las tablas derivadas se pueden utilizar en el lugar en que esté permitido utilizar una tabla, incluidas composiciones (`JOIN`).

Ejemplo:

Supongamos que se tiene la consulta

```
SELECT * FROM empleado WHERE cuota >= (SELECT SUM(importe) FROM
pedido WHERE rep = numempl);
```

para obtener los datos de los empleados que no han cubierto su cuota. Si se quieren obtener los clientes de estos empleados habría que hacer una composición entre este resultado y la tabla `cliente`. Esto se puede hacer utilizando una subconsulta en la clausula `FROM` de la forma:

```
SELECT cliente.nombre FROM cliente INNER JOIN (SELECT * FROM
empleado WHERE cuota >= (SELECT SUM(importe) FROM pedido WHERE rep =
numempl)) AS empleados2 ON cliente.repclie = empleados2.numempl;
```

Como se puede ver, se utiliza la “tabla” resultado de la consulta previa y se compone con `cliente`. También se pueden ver dos consultas anidadas.

Realizar la relación de ejercicios número 05.

10.- Vistas

Muchas bases de datos son utilizadas por distintos usuarios que tienen distintas necesidades y distintos niveles de seguridad.

Supongamos la siguiente tabla de ejemplo :

empleado

codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000

En esta tabla se almacenan los empleados de la empresa, junto con datos económicos, de contrato, etc.

Si un usuario de dirección desea acceder a información de los empleados, probablemente necesite acceso completo a todas las columnas de la tabla, pero si un visitante quiere consultar los empleados para localizar en qué oficina trabaja un empleado concreto, bastaría con los campos `nombre` y `oficina`.

Otra necesidad que surge a veces es la necesidad de usar con mucha frecuencia datos agregados (medias, sumas totales, etc) pero en las tablas hay que almacenar los datos en detalle, ya que la información que contienen puede ser necesaria en otros contextos. En este caso sería interesante disponer de tablas “virtuales” que permitieran consultar los datos agregados mientras que se dispone de los datos desagregados en las tablas “reales”.

La solución a este problema es el concepto de *vista*.

Una vista es una tabla virtual que se crea a partir de una consulta sobre otras tablas (o vistas). Cuando se utiliza una vista, el SGBD realiza la consulta almacenada en la vista y proporciona al usuario la ilusión de que existe una tabla con la estructura y datos de la vista. Esto soluciona los dos problemas mencionados anteriormente:

- Si se desea dar acceso a un usuario a un conjunto restringido de columnas de una tabla, se crea una vista sobre la misma con sólo los campos necesarios y obviando el resto. El usuario utiliza la vista en lugar de la tabla “real”. Si la tabla real se actualiza (se modifican los datos), la vista reflejará este hecho.
- Si se quiere hacer una nueva tabla con datos agregados de otras se puede hacer una vista sobre la misma utilizando las funciones agregadas. La tabla real permanece con los datos desagregados pero la virtual los tiene ya agregados y se actualizará cuando cambien aquellos.

10.1.- Creación de vistas

Para crear una vista es necesario saber realizar consultas (Sentencia `SELECT`), ya que cada vista está basada en una consulta subyacente. La sintaxis para la creación de vistas es:

```
CREATE VIEW nombre_vista AS sentencia_select;
```

donde:

- `nombre_vista` es el nombre de la nueva vista a crear. No puede coincidir con el nombre de otra vista o tabla ya existente en la base de datos.
- `sentencia_select` es la sentencia `SELECT` que va a proporcionar el esquema y los datos de la vista.

Ejemplo:

Para crear la vista de visitante `empleado_v` para la tabla `empleado` vista anteriormente, la sintaxis sería:

```
CREATE VIEW empleado_v AS SELECT nombre, oficina FROM empleado;
```

La sentencia `SELECT` puede utilizar todas las clausulas y subconsultas que necesite, aunque sólo puede especificar una clausula `ORDER BY` si también incluye una clausula `TOP`. De todas formas, no se garantiza el orden de las filas de la vista. Si se desean ordenar por algún criterio hay que especificarlo al crear la sentencia `SELECT` que consulte los datos de la vista.

Ejemplo:

Si se quiere crear una vista con los 5 empleados con mas ventas, se podría hacer utilizando:

```
CREATE VIEW empleado_top5 AS SELECT TOP 5 * FROM empleado ORDER BY
ventas_obtenidas DESC;
```

Esto crearía una vista llamada `empleado_top5` con los empleados con más ventas pero no garantiza que al hacer:

```
SELECT * FROM empleado_top5;
```

las filas aparezcan ordenadas por las ventas del empleado. Si se quiere asegurar que las filas devueltas estarás ordenadas por este criterio, hay que declararlo expresamente al realizar la consulta sobre la vista:

```
SELECT * FROM empleado_top5 ORDER BY ventas_obtenidas DESC;
```

Otra consideración importante del uso de vistas es que los cambios a las tablas subyacentes (las tablas sobre las que se basa la consulta) no modifican esta automáticamente, por lo que se pueden producir problemas. Por ejemplo, si se elimina un campo de una tabla subyacente que es utilizado en la vista, la vista fallará al utilizarla. Esto es así porque el sistema almacena sólo la consulta y la intenta ejecutar al utilizar la vista. Como ahora la consulta no es válida porque se utiliza un campo no existente se produce un error.

Por último hay que decir que, dado que la vista genera una "tabla", es necesario que en la lista de selección se den nombre (utilizando `AS`) a las columnas calculadas, de forma que la vista tenga una columna con un nombre válido.

10.2.- Modificación y eliminación de vistas

Para modificar una vista hay dos sintaxis distintas pero que realizan una función equivalente:

```
CREATE OR REPLACE VIEW nombre_vista AS sentencia_select;
```

o bien

```
ALTER VIEW nombre_vista AS sentencia_select;
```

En ambas tanto `nombre_vista` como `sentencia_select` tienen el mismo significado que en la sentencia `CREATE VIEW`. En principio la mayoría de los SGBDs soportan ambas sintaxis aunque hay que revisar la documentación del SGBD en que se esté trabajando para ver si una sintaxis es soportada.

Una sutil diferencia entre las dos sintaxis es que `CREATE OR REPLACE VIEW` puede utilizarse siempre, aunque la vista no esté ya creada, mientras que `ALTER VIEW` necesita que la vista ya esté creada.

Por tanto, se recomienda el uso de `CREATE OR REPLACE VIEW` siempre que sea posible.

Ejemplo:

Si ahora en nuestro "directorio" de empleados se desea incluir el número del mismo para facilitar su localización, se podría hacer utilizando:

```
CREATE OR REPLACE VIEW empleado_v AS SELECT codigo, nombre, oficina
FROM empleado;
```

O bien:

```
ALTER VIEW empleado_v AS SELECT codigo, nombre, oficina FROM
empleado;
```

Para eliminar una vista la sintaxis es:

```
DROP VIEW nombre_vista;
```

Ejemplo:

```
DROP VIEW empleado_v;
```

11.- Resumen

Para aprovechar completamente la potencia de un sistema de base de datos relacionales es necesario integrar la información proveniente de varias tablas, relacionada por valores de los campos clave.

Las consultas sobre varias tablas se pueden realizar utilizando combinaciones, que es la forma tradicional o subconsultas que es la forma que han ido añadiendo los SGBDs en las últimas versiones. Ambas permiten tener un abanico amplio de posibilidades de creación de consultas que, si el diseño de la base de datos es correcto, sólo tiene como límite la imaginación de creador de las consultas.

Asimismo, se han descrito las vistas, que se utilizan para proporcionar una vista limitada de los datos y para almacenar consultas.

Tema 9
Manipulación de los Datos

Tabla de Contenidos

1.- Introducción

En anteriores unidades se ha repasado ampliamente las maneras de acceder a la información contenida en la base de datos y extraer información útil para consultas o para aplicaciones.

Sin embargo, dicha información no está sólo para ser consultada, ya que, para que existan datos es necesario que primero se hayan añadido a la base de datos.

Asimismo, la información de una base de datos debe poder ser modificada, tanto para corregir errores como por cambios en los datos, que deben quedar reflejados.

Por último, a veces es necesario eliminar información que ya no se necesita o no está actualizada.

En esta unidad se verán los mecanismos que se emplean en las bases de datos relacionales para realizar estas operaciones.

Además se hablará de la problemática del acceso concurrente a los datos de la base de datos y de los mecanismos ofrecidos para que el acceso concurrente se realice con las garantías necesarias para evitar pérdidas de información o la aparición de inconsistencias.

2.- Añadir filas a una tabla

Para añadir filas a una tabla se utiliza la sentencia `INSERT`. Hay que tener en cuenta que la inserción se puede realizar *sobre una sola tabla*. Si se desean insertar filas en tablas distintas hay que utilizar una sentencia por tabla.

Existen dos formas de funcionamiento de `INSERT`. En una de ellas se especifica en la sentencia los datos a insertar en la tabla y en la otra se obtienen datos desde una consulta y se insertan como filas en una tabla. En las siguientes secciones se detallan estas dos formas.

Es importante tener en cuenta que la inserción de una nueva fila puede fallar aunque los datos que contenga sean perfectamente válidos. Esto puede ser debido a que la nueva fila viola alguna restricción de integridad establecida sobre la tabla en la que se inserta. Mas específicamente:

- No se pueden introducir valores duplicados en una clave primaria. Si ya existe un registro con el mismo valor de la clave primaria, la nueva fila no se podrá insertar.
- No se pueden introducir valores duplicados en una columna o columnas con la restricción `UNIQUE`, por la misma razón que la anterior.
- Si la tabla tiene claves ajenas, el sistema comprueba que los valores de dichas claves ajenas en la fila a insertar se corresponden con valores existentes en la tabla o tablas referenciadas. Si no se encuentra correspondencia, la fila no puede ser insertada.

2.1.- Indicando los datos directamente

La forma de `INSERT` para insertar una fila proporcionando los datos directamente es:

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

donde

- `tabla` es la tabla en la que se va a realizar la inserción
- `campo1`, `campo2`, `campoN` son los nombres de los campos en los que se va a proporcionar un valor para la fila a insertar.
- `valor1`, `valor2`, `valorN` son los valores que se proporcionan para los campos de nombre `campo1`, `campo2`, `campoN`, respectivamente.

Adicionalmente, hay que tener en cuenta las siguientes cuestiones sobre el funcionamiento de `INSERT`:

- Si en la lista de nombres de campos no aparecen todos los campos de la tabla, en aquellos que no aparezcan se insertará el valor por defecto para el campo correspondiente. El valor por defecto podrá ser:
  - El valor por defecto indicado por el modificador `DEFAULT` del campo al crear la tabla.
  - Si no se indicó valor por defecto y el campo no permite `NULL`, se insertará el valor por defecto para el tipo del campo, determinado por el SGBD.
  - Si no se indicó valor por defecto y el campo *permite* `NULL`, se insertará el valor `NULL`.

- Los valores para los distintos campos se pueden proporcionar utilizando constantes (por ejemplo, 'Jose' ó 1) o expresiones que utilicen funciones u operadores. El sistema intentará convertir el valor proporcionado o calculado al tipo de datos de la columna en que se va a almacenar. Si no puede realizar dicha conversión por la razón que sea, la sentencia producirá un error.
- Si, para un campo que aparece en la lista de nombres se quiere dar el valor por defecto de dicho campo, de forma explícita, se puede utilizar el valor especial `DEFAULT` para indicar que se desea almacenar el valor por defecto de ese campo.
- Si para un campo que aparece en la lista de nombres se quiere dar el valor `NULL`, bastará con indicarlo utilizando `NULL` como valor.

Ejemplos:

Supongamos que tenemos una tabla con la siguiente definición:

```
CREATE TABLE empleado (
     dni CHAR(9) NOT NULL,
     nombre VARCHAR(200) NOT NULL,
     sueldo NUMERIC(6,2) NOT NULL,
     complementos NUMERIC(6,2) DEFAULT 0 NOT NULL,
     fecha_nacimiento DATE NOT NULL,
     telefono CHAR(9) NULL,
     CONSTRAINT PRIMARY KEY (dni)
);
```

y con los datos:

empleado					
dni	**nombre**	**sueldo**	**complementos**	**fecha_nacimiento**	**telefono**
11111111A	José López	1232,12	576,88	1987-03-21	666666666
22222222B	Ana Sánchez	1656,87	123,77	1972-05-12	NULL
33333333C	Julia Rámirez	2333,76	777,22	1982-08-24	666777777

Para insertar al nuevo empleado, `Jaime Reyes`, con DNI `44444444D`, sueldo `987,67` euros, complementos `127,23` euros, fecha de nacimiento `23/07/1997` y con teléfono `666888888`, el comando a ejecutar sería:

```
INSERT INTO empleado (dni, nombre, sueldo, complementos,
fecha_nacimiento, telefono) VALUES ('44444444D', 'Jaime Reyes',
987.67, 127.23, '1997-07-23', '666888888');
```

Si se hubiera dado el caso de que `Jaime Reyes`, inicialmente no va a cobrar complementos, la inserción podría haberse realizado:

```
INSERT INTO empleado (dni, nombre, sueldo, complementos,
fecha_nacimiento, telefono) VALUES ('44444444D', 'Jaime Reyes',
987.67, 0, '1997-07-23', '666888888');
```

O, dado, que el valor por defecto de complementos es `0`, se podría haber hecho de la manera alternativa:

```
INSERT INTO empleado (dni, nombre, sueldo, fecha_nacimiento,
telefono) VALUES ('44444444D', 'Jaime Reyes', 987.67, '1997-07-23',
'666888888');
```

Obteniéndose en ambos casos la misma inserción.

Si el caso hubiera sido que `Jaime Reyes` no dispusiera de teléfono, la consulta debería haberse hecho:

```
INSERT INTO empleado (dni, nombre, sueldo, complementos,
fecha_nacimiento, telefono) VALUES ('44444444D', 'Jaime Reyes',
987.67, 127.23, '1997-07-23', NULL);
```

O bien

```
INSERT INTO empleado (dni, nombre, sueldo, complementos,
fecha_nacimiento) VALUES ('44444444D', 'Jaime Reyes', 987.67,
127.23, '1997-07-23');
```

2.2.- Utilizando subconsultas

A veces se quiere hacer permanente información que se obtiene mediante consultas. Para ello debería haber un mecanismo que permita almacenar datos provenientes de una o más tabla de la base de datos en otra. Este mecanismo es la segunda variación de la sentencia `INSERT`. En esta variación, la sintaxis es:

```
INSERT INTO tabla (campo1, campo2, ..., campoN) sentencia_select;
```

donde

- `tabla` es la tabla en la que se va a realizar la inserción.
- `campo1`, `campo2`, etc. son los campos a los que se le va a proporcionar un valor.
- `sentencia_select` es una sentencia `SELECT`, como las ya descritas en temas anteriores que es la que va a proporcionar los valores para insertar en la tabla.

Funciona como una sentencia `INSERT` en la que se indican los datos directamente pero utilizando una sentencia `SELECT` para obtener los mismos. La sentencia `SELECT` tiene unas restricciones:

- El número de columnas del resultado debe coincidir con el número de campos de la lista de campos de la sentencia `INSERT`.

- Los valores devueltos por la consulta para cada columna deben de poder convertirse al tipo de datos de la columna de destino. Si no se puede realizar una conversión, se producirá un error.

Ejemplos:

Supongamos que se ha decidido, por parte de la empresa, el eliminar los complementos y utilizar únicamente un salario. Para ello se he creado una nueva tabla para almacenar los datos de los empleados con la siguiente estructura:

```
CREATE TABLE empleado_nuevo (
     dni CHAR(9) NOT NULL,
     nombre VARCHAR(200) NOT NULL,
     salario NUMERIC(6,2) NOT NULL,
     fecha_nacimiento DATE NOT NULL,
     telefono CHAR(9) NULL,
     CONSTRAINT PRIMARY KEY (dni)
);
```

Como se puede comprobar es muy similar a la actual pero desaparece el campo `complementos` y el campo sueldo se ha renombrado a `salario`.

Si se quieren traspasar los datos de la tabla antigua a la nueva se podría hacer con la siguiente sentencia:

```
INSERT INTO empleado_nuevo (dni, nombre, salario, fecha_nacimiento,
telefono) SELECT dni, nombre, sueldo + complementos,
fecha_nacimiento, telefono FROM empleado;
```

2.3.- Copiar Tablas

Se puede realizar una copia de una tabla utilizando la sintaxis:

```
SELECT INTO tabla_nueva FROM tabla_antigua;
```

donde:

- `tabla_nueva` es la nueva tabla a crear. No debe existir ya una tabla con ese nombre o se producirá un error.
- `tabla_antigua` es la tabla a copiar.

Esta sentencia creará una nueva tabla igual que la original (estructura y datos).

NOTA: Esta sintaxis no funciona en MariaDB/MySQL. Se puede hacer lo mismo utilizando la sintaxis:

```
CREATE TABLE tabla_nueva SELECT * FROM tabla_antigua;
```

3.- Modificar filas de una tabla

La modificación de los datos de una tabla se realiza a través de la sentencia `UPDATE`, con la siguiente sintaxis:

```
UPDATE tabla SET campo1 = expresion1, campo2 = expresion2, ...,
campoN = expresionN [WHERE condicion];
```

donde:

- `tabla` es la tabla en la que está la fila o filas que se quieren actualizar (se pueden actualizar más de una fila con una sola sentencia `UPDATE`).
- `campoX` es un campo cuyo valor se quiere modificar.
- `expresionX` es el nuevo valor que se va a asignar al campo.
- `condicion` es una condición sobre los campos de cada fila de la tabla `tabla`. Aquellas filas que cumplan la condición *antes de ser actualizadas* se actualizaran. Aquellas que no la cumplan permanecerán sin cambios.

La sentencia funciona de la siguiente manera:

- Si se proporciona una clausula `WHERE` se procesa toda la tabla y se comprueba la condición para cada fila de la tabla. Aquellas filas en las que la condición proporcione un valor `VERDADERO` serán actualizadas. Las filas en las que la condición proporcione un valor `FALSO` **no** serán actualizadas.
- Si no se proporciona una clausula `WHERE` **todas las filas de la tabla serán actualizadas**. Es por esto por lo que hay que tener especial cuidado a la hora de utilizar `UPDATE` sin una clausula `WHERE`.
- Para crear la condición de la clausula `WHERE` se pueden utilizar la misma sintaxis que para las clausulas `WHERE` de la sentencia `SELECT`, incluidas subconsultas. Sin embargo, hay que tener en cuenta que para evaluar la condición se utilizan los valores que están actualmente almacenados en la tabla, o sea, ***los valores existentes antes de realizar la actualización***.
- Una vez seleccionadas las filas sobre las que se va a realizar la actualización, se recorren una a una y se van aplicando las actualizaciones indicadas por las parejas `campo = valor` indicadas en la clausula `SET`. En cada fila se cambia el valor del campo con nombre `campo` al nuevo valor indicado en la expresión `valor`.
- La expresión que da el nuevo valor puede ser un valor constante (a todas las filas se le va a asignar el mismo valor) o puede ser una expresión que utilice los valores actuales de la fila a actualizar para calcular los nuevos valores. Esta expresión puede utilizar operadores, funciones y subconsultas que devuelvan un único valor (una única columna y una única fila). En la subconsulta se pueden utilizar los valores de los campos de la fila que se está actualizando.

Al realizar actualizaciones, hay que tener especial cuidado con la modificación de columnas que participen en una regla de integridad referencial (`FOREIGN KEY`) o de identidad (`PRIMARY KEY` / `UNIQUE`) ya que pueden producirse situaciones en las que la actualización provocaría la violación de alguna de estas reglas. En ese caso la sentencia fallaría con un error.

Ejemplos:

Para subir el sueldo un 5% a todos los empleados cuyo salario neto (sueldo mas complementos) es menor de 1500 euros:

```
UPDATE empleado SET sueldo = sueldo * 1.05 WHERE (sueldo + complementos) < 1500;
```

Para almacenar el sueldo completo en sueldo y poner el complemento a cero, se utilizaría:

```
UPDATE empleado SET sueldo = sueldo + complementos, complementos = 0;
```

Como se puede ver, no se utiliza `WHERE` por lo que la actualización se realizará para *todos* los empleados.

4.- Eliminar filas de una tabla

La eliminación de filas se realiza mediante la sentencia `DELETE`, con la siguiente sintaxis:

```
DELETE FROM tabla [WHERE condicion];
```

donde:

- `tabla` es la tabla en la que se van a eliminar filas,
- `condicion` es una condición que se evalúa por cada fila de la tabla.

El funcionamiento es muy sencillo: Se recorre toda la tabla `tabla` y por cada fila se evalúa `condicion`. Si el resultado es `VERDADERO`, la fila se elimina de la tabla. Si el resultado es `FALSO`, la fila permanece. La clausula `WHERE` es opcional y si no se indica significa que se eliminarán **TODAS** las filas.

Como en `UPDATE`, en la condición se pueden emplear cualquier operador o función que es válido en la clausula `WHERE` de una sentencia `SELECT`, incluidas las subconsultas.

Ejemplos:

Para eliminar todos los empleados cuyo salario sea superior a los 3000 euros se emplearía la sentencia:

```
DELETE FROM empleado WHERE (sueldo + complementos) > 3000;
```

O para dejar vacía la tabla `empleado` se podría utilizar:

```
DELETE FROM empleado;
```

Realizar la relación de Ejercicios 01.

5.- Transacciones

Hasta ahora todas las operaciones sobre las bases de datos se han realizado por parte de un sólo usuario. Sin embargo, los SGBDs pierden utilidad si la información sólo puede ser manipulada por un usuario. Para funcionar a plena potencia se necesita permitir el acceso simultáneo de varios usuarios (el número variará según el SGBD) a los mismos datos.

Este acceso simultáneo o *concurrente*, lamentablemente, presenta algunos problemas que deben ser evaluados y corregidos. Entre los problemas que presenta el acceso concurrente a los datos tenemos:

- Pérdida de actualización. Imaginemos el siguiente escenario: Dos personas, A y B van a sacar dinero de la misma cuenta utilizando dos cajeros automáticos distintos. En la cuenta hay actualmente 100 euros y el usuario A quiere sacar 50 euros y el B 40. El usuario A lee el saldo y el sistema dice 100 euros. El usuario B también lee el saldo y es 100. El usuario B saca el dinero y se actualiza la cuenta que ahora tendrá 100 - 40 = 60 euros. A continuación el usuario A, que se ha retrasado un poco, saca también su dinero y actualiza la cuenta, que ahora tendrá 100 - 50 = 50 euros. Como se puede ver se ha producido un error porque la cuenta debería tener tras las dos retiradas un total de 100 - 50 - 40 = 10 euros pero termina teniendo 50 euros en su lugar.
- Lecturas de valores incorrectos. Imaginemos el mismo escenario anterior pero ahora la secuencia de pasos es distinta. El usuario A lee el saldo (100) y como hay suficiente, lo actualiza al nuevo saldo (100 - 50 = 50). El usuario B lee el saldo (50) y como tiene suficiente, lo actualiza al nuevo saldo (50 - 40 = 10). A continuación el usuario A decide cancelar la operación y deja el saldo de la cuenta como estaba (100). El usuario B continúa y obtiene su efectivo. Ahora la cuenta tiene un saldo de 100 pero realmente se han retirado 40 euros y debería tener un saldo de 100 - 40 = 60 euros.
- El problema del resumen incorrecto. Imaginemos un escenario distinto. El usuario A está calculando el total de las facturas de un mes determinado, recorriendo las facturas una a una y sumando su importe a un total. Al mismo tiempo el usuario B añade una factura nueva a ese mes que se había traspapelado y el usuario C modifica el importe de otra factura de ese mismo mes. Según el orden en que se hagan estas operaciones, el resumen dará un valor distinto, pero probablemente en ninguno de los casos sea correcto con el valor que debería tener si las tres operaciones se hubieran hecho una después de la otra.

Estos problemas pueden provocar inconsistencias cuando se accede concurrentemente a la misma información. Por lo tanto el SGBD debe poder asegurar de alguna forma que estos problemas no ocurren.

El mecanismo por el que se materializa la solución se denomina *transacción*. Una transacción es un grupo de operaciones sobre la base de datos que cumple las siguientes reglas:

- **Atomicidad.** O bien el efecto de todas las operaciones realizadas dentro de la transacción se almacenan de forma permanente o bien ninguna lo hace. Dicho en otras palabras, o bien todas

las operaciones se realizan o no se realiza ninguna (porque se produzca un error o la transacción se aborte). Todas las operaciones de la transacción funcionan como si fueran una sola.

- **Consistencia.** Cada transacción se encuentra la base de datos en un estado consistente antes de comenzar y la debe dejar en un estado consistente al terminar, ya se realicen sus operaciones o no (ver Atomicidad). Las reglas de integridad de la base de datos se pueden violar temporalmente mientras la transacción está en curso pero al terminar se deben cumplir, ya sea porque el efecto total de las operaciones realizadas cumple las reglas de integridad o bien porque la transacción se cancela, en cuyo caso la base de datos vuelve al consistente estado previo a la transacción.
- **Aislamiento.** Las transacciones no pueden interferir entre ellas, o sea, los efectos de una transacción que aún no ha terminado no deben ser visibles para otras transacciones. Este es el principal objetivo del control de acceso concurrente.
- **Durabilidad.** Cuando una transacción se ha confirmado con éxito, sus efectos deben permanecer en la base de datos aunque ocurran errores en la base de datos o el equipo se apague.

Estas cuatro reglas son las que se conocen como ACID, por las siglas en inglés de sus nombres (Atomicity, Consistency, Isolation, Durability).

Las transacciones utilizan distintas técnicas para obtener el cumplimiento de estas reglas. De ellas, las más utilizadas son:

- Bloqueo (Locking). Cada dato posee un bloqueo (lock). Si un usuario quiere acceder a un dato durante una transacción debe obtener el bloqueo antes. Si lo obtiene, ningún otro usuario podrá escribir a ese dato hasta que el bloqueo se libere y el nuevo usuario lo adquiera. El problema de esta técnica es que puede provocar problemas de esperas mientras se liberan los bloqueos y también problemas de interbloqueos (situaciones en que dos usuarios tienen un bloqueo cada uno y están esperando por el bloqueo que tiene el otro). Aún con todos sus problemas es uno de los sistemas más utilizados.
- Comprobación de grafos de serialización. El sistema reordena las transacciones en curso de forma que se ejecuten unas antes que otras. Si se bloquean entre ellas aborta una y deja seguir a la otra.
- Multiversión. Se mantienen varias versiones de cada dato con un timestamp que indica el momento de la modificación. Una transacción sólo accede a los datos en su versión anterior al inicio de su ejecución. Ocupa más espacio pero previene esperas y es muy rápido.

5.1.- Ejecutar una secuencia de sentencias en una transacción

Ejecutar sentencias en una transacción es muy sencillo.

Una transacción se comienza utilizando la sentencia:

```
BEGIN TRANSACTION;
```

La palabra `TRANSACTION` es opcional.

Esta sentencia indica que comienza una transacción. A partir de ese punto, y hasta que se termine la transacción, todas las sentencias forman parte de la transacción y se ejecutarán de forma atómica.

Una transacción se finaliza utilizando las sentencias:

```
COMMIT;
```

ó

```
ROLLBACK;
```

`COMMIT` confirma la transacción y finaliza la misma incorporando los cambios realizados a la base de datos.

`ROLLBACK` aborta la transacción y finaliza la misma descartando los cambios realizados a la base de daros, que permanecerá como estaba al iniciar la transacción.

Ejemplos:

Supongamos que tenemos una tabla `cuenta` en las que se almacenan las cuentas corrientes de los clientes. Entre otros datos cada cuenta tiene un id único de cliente y el saldo actual de la cuenta. Si se deseara retirar dinero (100 euros) de la cuenta número 222444555444, una posible secuencia de sentencias a ejecutar por parte de una aplicación sería:

```
BEGIN TRANSACTION;
SELECT saldo FROM cuenta WHERE idCliente =  222444555444;
.... El programa comprueba que el saldo es suficiente, esto es, que
es igual o mayor a lo que se desea retirar (100 euros)....
... Supongamos que el saldo es suficiente (525 euros)....
UPDATE cuenta SET saldo = 425 WHERE idCliente =  222444555444;
COMMIT;
```

Una posible secuencia si el saldo no fuera suficiente...

```
BEGIN TRANSACTION;
SELECT saldo FROM cuenta WHERE idCliente =  222444555444;
.... El programa comprueba que el saldo es suficiente, esto es, que
es igual o mayor a lo que se desea retirar (100 euros)....
... Supongamos que el saldo NO es suficiente (75 euros)....
ROLLBACK;
```

6.- Vistas actualizables

En la mayoría de los casos, las vistas son mecanismos de sólo lectura, esto es, pueden utilizarse para consultar datos pero no se pueden insertar, modificar ni eliminar filas.

En algunos casos, se puede permitir que una vista pueda ser modificable, actualizándose la tabla subyacente pero para que esto pueda ser posible debe existir una relación uno a uno entre las filas de la vista y las filas de la(s) tabla(s) subyacente(s). Si esto no se cumple, la vista es no actualizable. Esto ocurre especialmente cuando en la sentencia `SELECT` se utilizan:

- Funciones de agregación (`SUM`, `AVG`, etc.).
- `DISTINCT`
- `GROUP BY`
- `HAVING`
- `UNION`
- Subconsultas en la lista de columnas.
- Algunos tipos de composiciones (`JOIN`s).

También pueden ocurrir problemas si una vista no contiene todos los campos de la tabla subyacente y alguno de estos campos no puede ser `NULL` ni tiene un valor por defecto. En este caso se producirá un error al intentar utilizar la sentencia `INSERT` sobre la vista.

En general, se podrán actualizar con mayor o menor seguridad vistas sobre una sola tabla (sin subconsultas ni composiciones) y que tengan todos las columnas de la tabla original que no puedan valer `NULL` o no tengan valor por defecto.

Ejemplo:

Si se tiene la definición de la vista `empleado_v`, y suponiendo que los campos no contenidos en la vista pueden valer `NULL`, se podría realizar la siguiente inserción:

```
INSERT INTO empleado_v (codigo, nombre, oficina) VALUES (111, 'Lola Sanchez', 22);
```

que produciría el siguiente resultado sobre la tabla subyacente:

EMPLEADO								
CODIGO	**NOMBRE**	**EDAD**	**OFICINA**	**TITULO**	**FECHA_CONTRATO**	**CODIGO_JEFE**	**CUOTA_VENTAS**	**VENTAS_OBTENIDAS**
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000
111	Lola Sanchez	NULL	22	NULL	NULL	NULL	NULL	NULL

Un problema que se puede presentar al utilizar vistas es que pueden ocasionar problemas de seguridad. Tomemos el siguiente caso:

Supongamos la tabla `empleado` utilizada anteriormente. Por razones que no vienen al caso se decide que el usuario `gestor` debe tener acceso a los datos de los empleados de tipo `Representante` a fin de gestionar su trabajo, pero no debe tener acceso a otro tipo de empleados.

Para llevar esto a cabo se crea una vista, llamadas `empleado_gestor` con la siguiente sentencia:

```
CREATE OR REPLACE VIEW empleado_gestor AS SELECT * FROM empleado
WHERE TITULO = 'Representante';
```

Ésto crearía una vista, llamada `empleado_gestor` que sólo contiene los representantes. Como se puede comprobar, esta vista es actualizable ya que abarca todos los campos de una sola tabla y no se utilizan agregados, ni ninguna de las otras funcionalidades que impiden la actualización.

Sin embargo, a la hora de insertar registros en la vista se puede producir un problema de seguridad porque el usuario `gestor` podrá crear un director de ventas simplemente utilizando la sentencia:

```
INSERT INTO empleado_gestor (codigo, nombre, edad, oficina, titulo,
fecha_contrato, codigo_jefe, cuota_ventas, ventas_obtenidas) VALUES
(112. 'Falso Director' 99, 22, 'Director General', '2017-04-04',
NULL, 0, 0);
```

Además, otro efecto lateral es que esta nueva fila *no aparecerá en la vista a pesar de haber sido insertada a través de ella*. La razón es que la nueva fila *no cumple la condición de la sentencia `SELECT` de la vista* y, por lo tanto, no aparece en ella, aunque si se ha insertado en la tabla subyacente (`empleado`).

Para evitar esto se utiliza la clausula `WITH` al final de la sentencia `CREATE OR REPLACE VIEW`, de la forma:

```
CREATE OR REPLACE VIEW nombre_vista AS sentencia_select WITH
opcion_with;
```

donde `opcion_with` puede ser:

- `CHECK OPTION`. Esta opción obliga a que las modificaciones que se realicen a través de la vista deben cumplir la restricción de la misma, esto es, que la nueva fila o la fila actualizada debe formar parte de la vista después de la inserción ó modificación. En caso de que no se cumpla esta condición, la inserción o modificación producirá un error.
- `READ ONLY`. Esta opción obliga a que la vista sea de sólo lectura y se prohíba la inserción o modificación de filas. Esta opción no está soportada por MySQL.

7.- Resumen

En este tema se ha revisado la forma que tiene SQL de modificar los datos.

Se ha visto como se añaden datos (`INSERT`), modifican los existentes (`UPDATE`) o se eliminan (`DELETE`).

Asimismo se ha descrito el problema del acceso concurrente a los datos y la solución adoptada en forma de transacciones, asi como su expresión en el lenguaje SQL.

Por último se han visto las vistas actualizables, así como los problemas que pueden ocasionar y como solucionarlos.

8.- Apéndice A: Transacciones en MariaDB

MariaDB utiliza las transacciones siguiendo la sintaxis especificada en los apartados anteriores. Sin embargo es conveniente indicar algunas características particulares de MariaDB:

- Por defecto, MariaDB realiza una transacción por cada sentencia que ejecuta. A esto es llamado el modo `autocommit`.
- Al ejecutar una sentencia `BEGIN`, MariaDB suspende temporalmente este modo hasta que la transacción finaliza.
- MariaDB soporta una sintaxis alternativa para iniciar una transacción utilizando la sentencia `START TRANSACTION`. El funcionamiento es idéntico.
- Para que se soporten transacciones, el motor de almacenamiento de la base de datos debe ser InnoDB. El resto de motores de almacenamiento soportados por MariaDB no soportan transacciones.
- MariaDB comprueba las reglas de integridad referencial durante una transacción, por lo que estas no se pueden violar temporalmente, como es el comportamiento estándar.

Tema 10
Programación de Guiones

Tabla de Contenidos

1.- Introducción

La realización de consultas o manipulaciones de datos sueltos son ciertamente una herramienta potente, pero no permiten realizar operaciones sobre los datos que requieren un proceso más "inteligente", por ejemplo, la migración de una determinada parte de la base de datos a otra base de datos nueva.

En este caso una solución es la de realizar un programa, utilizando cualquier lenguaje compatible, y realizar desde este programa los comandos SQL adecuados para realizar el proceso.

Los modernos SGBD ofrecen una solución alternativa: Ofrecer ellos mismos un lenguaje en que se pueden realizar procesos más complejos que los permitidos utilizando simple SQL. A este tipo de "programas" incrustados y ejecutados por el mismo SGBD se les denomina guiones y son el objeto del presente tema.

2.- Lenguaje de programación

Existe una gran variedad de lenguajes de programación de guiones, puesto que esta característica está totalmente fuera de lo definido en los estándares, por lo que cada SGBD lo ha implementado siguiendo su propio "camino". En esta sección y las siguientes se describirán las características comunes a los lenguajes de guiones, sin referenciar un SGBD en concreto.

2.1.- Tipos de guiones

Según el motor de SGBD, se pueden tener los siguientes tipos de guiones:

- **Procedimientos almacenados.** Estos procedimientos se pueden invocar como si fueran sentencias del lenguaje. El usuario los invoca explícitamente utilizando su nombre y se le pueden pasar parámetros y devolver resultados. Se pueden tomar como sentencias escritas por el usuario en contraposición a las que vienen de serie con el SGBD.
- **Funciones de usuario.** Estas funciones se pueden utilizar desde sentencias del lenguaje como si fueran funciones de las que vienen predefinidas por el SGBD. Por ejemplo, el sistema usualmente proporciona una función `AVG` para calcular la media de un conjunto de valores pero no siempre proporcionar funciones estadísticas más avanzadas como mediana o la varianza. Un usuario podría definir una función `MEDIANA` que calculara la mediana de un grupo de valores y esta función podría utilizarse en sentencias `SELECT` o `INSERT`, por ejemplo.
- **Procesos disparados.** Estos procedimientos se invocan de forma automática por parte del SGBD cuando ocurre un evento o eventos determinados. Por ejemplo, se puede escribir un guión que es invocado cada vez que se va a eliminar una fila de determinada tabla. En este guión se podría realizar la "limpieza" de datos de otras tablas relaciondos con la fila que se va a eliminar.

2.2.- Características generales del lenguaje

Los guiones utilizan un lenguaje que suele ser el mismo SQL soportado por el SGBD, con la adición de:

- **Variables.** Permiten almacenar información durante el proceso del script.
- **Módulos.** Permiten crear scripts complejos a partir de scripts más pequeños utilizando la filosofía "divide y vencerás" para la creación de scripts.
- **Estructuras de control de flujo.** Para poder tomar decisiones y realizar iteraciones.
- **Control de excepciones.** Para poder decidir qué se hace si se produce algún problema durante el proceso de forma que se mantenga la integridad de los datos.

2.3.- Tipos de datos

Los tipos de datos definen qué puede contener una variable de un guión (los valores válidos de una variable) y qué operaciones se pueden realizar sobre ella.

Usualmente los tipos de datos que disponibles para las variables de un SGBD coinciden con los tipos de datos disponibles para las columnas de las tablas de dicho SGBD, es decir, si la base de datos dispone de un tipo `INTEGER` para definir columnas de tipo entero, también se podrán crear variables

de tipo `INTEGER`, que podrán contener valores enteros con las mismas restricciones que una columna de tipo `INTEGER`.

Asimismo, al igual que ocurre con las columnas, a una variable se le puede asignar el valor especial `NULL`.

2.4.- Identificadores

Las variables se identifican siempre por un nombre que define el usuario. Este nombre se denomina *identificador*. Aunque el nombre lo define el programador a su gusto, cada SGBD impone una serie de normas que deben cumplir los identificadores a fin de evitar problemas de sintaxis. Por ejemplo, los SGBDs suelen prohibir que se utilicen identificadores que coincidan con palabras clave del lenguaje SQL como `SELECT` o `GROUP`.

Para andar sobre camino seguro es recomendable seguir las siguientes reglas básicas:

- Utilizar sólo letras inglesas (nada de eñes o vocales acentuadas), números y subrayado (_).
- Comenzar siempre los identificadores por letra (los números suelen estar prohibidos como primer caracter de un identificador y algunos sistemas utilizan el subrayado como primer caracter de variables especiales de sistema).
- Suponer siempre que el sistema no distingue entre mayúsculas y minúsculas, esto es, que para el sistema Resultado, resultado, y ReSuLtAdO son la misma variable. Si no lo son no se habrá perdido nada. Si lo son, se habrá evitado un problema difícil de identificar.

2.5.- Declaración de variables

Los SGBDs suelen requerir que las variables se declaren antes de ser utilizadas. La declaración tiene como efecto que el sistema reserve espacio de almacenamiento para la variable y se le asigne un tipo de forma que se pueda conocer qué valores se pueden almacenar en dicho espacio y qué operaciones se permiten sobre dicha variable.

La declaración usualmente se realiza con una palabra clave reservada al efecto y una especificación del identificador y tipo de la variable. Opcionalmente también se suele permitir asignar a la variable un valor inicial.

2.6.- Constantes y literales

En el curso de la programación de guiones se hace frecuentemente necesario el de valores constantes.

Como viene siendo la tónica habitual, los SGBDs usualmente utilizan la misma sintaxis para los valores constantes o literales que utilizan en las sentencias SQL. Entre otros existen los siguientes tipos de literales:

- Números enteros

- Números reales en notación de coma fija.
- Números reales en notación científica.
- Cadenas y caracteres.
- Tiempos (fechas , horas, fechas/horas).

2.7.- Operadores

Los operadores que se pueden utilizar en la programación de guiones son los mismos que se pueden utilizar en las sentencias SQL. Haciendo un rápido repaso estos son:

- Operadores aritméticos. Se emplean para realizar cálculos aritméticos sobre cantidades numéricas:
  - Suma
  - Resta
  - Multiplicación
  - División
  - División entera
  - Resto de la división entera (módulo)
  - Exponenciación
- Operadores de relación o comparación. Se emplean para comparar dos valores y devuelven un resultado lógico (VERDADERO o FALSO).
  - Mayor que
  - Menor que
  - Mayor o igual que
  - Menor o igual que
  - Igual a
  - Distinto de
- Operadores lógicos. Se emplean para combinar valores lógicos y obtener expresiones para las sentencias de toma de decisiones. Operan sobre valores lógicos y devuelven valores lógicos:
  - Y (AND)
  - O (OR)
  - No (NOT)

- O exclusivo (XOR)
- No O (NOR)
- No Y (NAND)

2.8.- Funciones predefinidas

Los operadores vistos en el apartado anterior permiten la realización de muchas operaciones pero no son un juego completo. Habitualmente se necesitan realizar operaciones sobre los datos que, o bien son muy difíciles de implementar utilizando los operadores básicos o bien son totalmente imposibles de realizar utilizando los mismos.

Para esta eventualidad, los SGBDs ofrecen una librería más o menos amplias de *funciones predefinidas*. Una función, en un lenguaje de programación de guiones, es una construcción que toma ninguno, uno o más de un valor de entrada y devuelve un único valor de salida, que depende de los valores de entrada.

El conjunto de funciones predefinidas es fuertemente dependiente del SGBD en que se trate, por lo que sería inútil intentar dar una lista aquí. Sin embargo, todos los SGBDs tienen en común que proporcionan funciones encuadradas en las siguientes categorías:

- **Funciones de cadena.** Permiten realizar operaciones sobre cadenas de caracteres tales como buscar texto en una cadena, realizar conversiones (mayúsculas/minúsculas, etc.) o componer cadenas a partir de otras.
- **Funciones numéricas.** Permiten realizar cálculos con cantidades enteras o reales.
- **Funciones temporales.** Permiten manipular fechas, horas y duraciones.
- **Funciones de conversión.** Permiten realizar conversiones entre los distintos tipos de datos (de caracter a número, de fecha a texto, etc.).
- **Funciones varias.** Permiten realizar funciones que no encajan en una categoría concreta como obtener información del sistema, conocer el estado de una transacción, etc.

2.9.- Estructuras de control

Las estructuras de control determinan el orden en que se van a ejecutar las sentencias que componen el guión. Las secuencias se corresponden con las secuencias clásicas de la programación estructurada, aunque algunos SGBDs también incluyen características de orientación a objetos como encapsulación, herencia y polimorfismo, que no se tratarán aqui. Las estructuras que seguro se podrán encontrar en todos los lenguajes de guiones son:

- **Secuencia.** Es el flujo "natural" del guión. Consta de varias sentencias, una a continuación de la otra. El sistema las ejecuta en el mismo orden en que las encuentra.

- **Condición simple.** La condición simple bifurca la secuencia de ejecución en dos caminos, dependiendo del valor de una condición. Si la condición produce un valor VERDADERO se sigue un camino de ejecución, mientras que seguirá otro si se produce un valor FALSO. Sea cual sea el camino elegido, al final del mismo la ejecución sigue por un único camino al finalizar la estructura.
- **Condición múltiple.** La condición múltiple elige entre uno de varios caminos a partir de una condición que produce un resultado con más de un valor (en contraposición a un valor VERDADERO/FALSO, como la condición simple). El camino elegido se ejecuta y se continúa por el final de la estructura, como en la condición simple.
- **Ciclos.** Los ciclos son estructuras que repiten la ejecución de la estructura que contienen en su interior varias veces. El número de veces que se repite la ejecución depende del tipo de ciclo:
    - **Ciclo tipo MIENTRAS.** Un ciclo tipo MIENTRAS está gobernado por una condición que se evalúa al inicio de dicho ciclo. Si la condición produce un valor VERDADERO se ejecuta el contenido del ciclo. Si el valor es FALSO, se acaba el ciclo, su contenido no se ejecuta y se continúa por la siguiente estructura. Después de cada ejecución del contenido del ciclo se vuelve a evaluar la condición y el proceso se repite. Si no se es cuidadoso y se programa el ciclo de forma que la condición cambie eventualmente se puede producir un ciclo infinito sin salida.
    - **Ciclo tipo REPETIR.** Un ciclo tipo REPETIR es parecido al tipo MIENTRAS, en el sentido de que depende de una condición pero en este caso la condición se comprueba **al final** de la ejecución del ciclo y cuando el contenido de este se ha ejecutado al menos una vez.

2.10.- Subprogramas

Los programas simples son fáciles de realizar pero cuando los guiones se vuelven más complejos se hece necesaria la presencia de mecanismos que permitan lidiar con dicha complejidad. El mecanismo ofrecido es el de subprogramas.

Un subprograma es una parte de programa que realiza una función específica y que puede ser utilizada desde otra parte del programa. De esta forma los programas se construyen de forma *modular*.

Normalmente, un subprograma realiza una tarea concreta y usualmente necesita, para poder realizar su labor, el recibir información desde la parte del programa que requiere sus servicios y a veces también devuelve información como resultado. Otras veces no devuelve información ninguna o la información procesada se almacena en la base de datos.

A los módulos que no devuelven información se les denomina procedimientos y a los que SI devuelven información, funciones.

2.11.- Procedimientos

Un procedimiento es un subprograma que realiza una tarea pero no devuelve información directamente. Usualmente se emplean para realizar tareas en lugar del programa que los utiliza o para implementar una respuesta a un disparador.

Ejemplo:

Supongamos el caso de que se tiene una base de datos que se utiliza para almacenar información de un servicio de Internet (un foro por ejemplo). Una de las tablas que tendría esta hipotética base de datos sería una tabla de usuarios. Supongamos que se decide, como política del sistema, que aquellas cuentas que no han sido accedidas en los últimos seis meses deben ser eliminadas junto con los mensajes enviados desde dichas cuentas.

Ésta sería una tarea apropiada para un procedimiento ya que no devuelve información (directamente al menos, ya que la base de datos si se ve modificada como resultado de su acción).

2.12.- Funciones

Una función es un subprograma que realiza un procesado de información y devuelve un único valor. Las funciones pueden realizarse tanto para ser utilizadas desde otros programas como para servir en sentencias SQL como funciones definidas por el usuario, sirviendo efectivamente como una forma de ampliar la librería de funciones predefinidas del SGBD con funciones no disponibles pero que son necesarias o convenientes para un usuario u organización.

Ejemplo:

Supongamos que se tiene una base de datos y, para ciertas tareas estadísticas, se desea calcular la moda de la edad de los usuarios (la moda de una distribución aleatoria es el valor que más veces aparece en la muestra). Esta función no suele ser proporcionada por defecto por los SGBD por lo que debe ser calculada a base de consultas.

Con las funciones definidas por el usuario, un programador puede crear una funcíón MODA() a la que se le proporcione un conjunto de valores y devuelva el valor moda.

Ésta sería una tarea apropiada para una función ya que se recibe un conjunto de valores y se devuelve un único resultado.

2.13.- Tratamiento de errores (excepciones)

Un programa robusto debe ocuparse no sólo de realizar la tarea que debe realizar sin problemas cuando todo va bien sino saber enfrentarse a los problemas o situaciones excepcionales cuando se presentan. Por ejemplo, supongamos que se crea un procedimiento al que se le pasa un ID de usuario y elimina todo rastro de dicho usuario de la base de datos. En condiciones normales, el usuario del que se proporciona el ID existirá y se podrá realizar la operación pero, ¿qué ocurre si el usaurio no existe? En este caso estamos ante una excepción y el programa debe tratarla de forma que el programa no falle

estrepitosamente sino que se adapte a la nueva situación lo mejor posible e intente realizar lo máximo de la tareas que tiene encomendada o, en caso de que ésto no sea posible, deshacer los cambios realizados y, sobre todo, dejar los datos en un estado consistente siempre.

A este tipo de condiciones excepcionales que ocurren se les denominan *excepciones* y un lenguaje de guiones robusto tiene soporte tanto para generarlas (nuestro programa la generará cuando se encuentre en circunstancias que le impiden continuar) como para responder a ellas (realizando las labores de recuperación que se pueda antes de terminar).

3.- Cursores

Muchas veces se hace necesario el procesamiento de una o más tablas fila a fila dentro de un guión, realizando operaciones complejas a partir de los valores de dicha fila.

Para simplificar el desarrollo de dichas operaciones, muchos SGBDs disponen de una estructura de control denominada *cursor*.

El uso de los cursores se hace necesario cuando el procesamiento de la filas no se puede hacer en conjunto sino de forma individual para cada fila.

Un cursor se declara como si fuera una variable (de hecho lo es, pero de un tipo más complejo) junto con la consulta que realiza.

Una sentencia (`OPEN`) abre el cursor, evalúa la consulta y hace los resultados accesibles. El cursor se sitúa en la primera fila del resultado.

Otra sentencia (`FETCH`) obtiene los valores de las columnas de la fila actual en las variables especificadas y mueve el cursor a la siguiente fila.

Por último otra sentencia (`CLOSE`) se encarga de liberar los recursos asociados al cursor.

4.- Disparadores

Usualmente un guión, tal y como se ha visto hasta ahora, se ejecuta a voluntad del usuario que lo debe invocar directamente, pero en ocasiones sería conveniente el disponder de guiones que se ejecutaran cuando se produjeran determinadas condiciones sobre los datos de la base de datos, a fin de responder a estos eventos o condiciones de forma adecuada.

Muchos SGBDs implementan para realizar esta tarea el mecanismo conocido como *disparador* (`TRIGGER`).

Un disparador o trigger es un objeto que se asocia a una tabla y a una operación sobre la misma y se invoca cuando se realiza la operación indicada sobre la tabla.

Usualmente también se permite especificar si se debe realizar el disparo *antes* o *después* de que se ejecute la operación.

Asimismo, otro problema que se pueden presentar cuando se utilizan disparadores es la presencia de más de un disparador para una misma operación o tabla. Algunos SGBDs sólo permiten la presencia de un disparador por operación y tabla y la creación de otro nuevo o bien se prohibe o bien sobreescribe el ya existente. Otros SGBDs, sin embargo, permiten encadenar disintos disparadores, de forma que se invoquen en cadena cuando se produzca la condición de disparo. Esto hace necesario arbitrar algún mecanismo que permita indicar el orden de prioridad de los distintos disparadores así como el procedimiento a seguir cuando alguno de ellos falla o provoca un error.

5.- Temporizadores

Como se ha visto, los disparadores son formas reactivas de ejecutar guiones. Se responde a una acción concreta con la ejecución de un guión.

En otras ocasiones se desean realizar tareas de forma reactiva pero lo que determina la reacción es *el paso del tiempo*. Por ejemplo, supongamos que se tiene una base de datos que soporta un servicio de Internet (digamos que un foro). Usualmente estos servicios permiten el registro de usuarios sin intervención de ningún operador. Los usuarios introducen sus datos y el sistema les registra en el mismo de forma que pueden utilizarlo. Un requisito fundamental de dichos sistemas es que el usuario proporcione una cuenta de correo válida, usualmente por motivos publicitarios. Muchos usuarios comenzaron a proporcionar cuentas falsas o inventadas para evitar molestias, pero los administradores contraatacaron añadiendo un paso más al registro. Cuando el usuario finaliza de introducir sus datos se le envía un mensaje de correo electrónico con un enlace que deben visitar si desean que la nueva cuenta se active y se les da un periodo de tiempo razonable para que realicen dicha activación. Si transcurrido este tiempo no se ha producido la activación, el pre-registro se cancela.

En este caso se tiene una tabla, la de usuarios, que contendrá un número indeterminado de registros correspondientes a usuarios que se han pre-registrado pero que nunca han finalizado el registro. Esta información "basura" no tiene ningún valor y afecta al rendimiento del sistema.

La solución más sencilla es la de chequear la tabla periódicamente y eliminar las cuentas que no estén activas y para las que haya transcurrido un periodo superior al de espera desde la pre-activación.

Esta tarea no se puede realizar con disparadores puesto que no es reactiva a modificaciones en la tabla sino que depende del tiempo.

Para disparar acciones en función del tiempo se dispone de los temporizadores o eventos.

Un temporizador es un objeto que inicia la ejecución de un programa en momentos determinados por el creador del temporizador. Los hay de dos tipos:

- De un solo disparo (oneshot). Se programan para un momento determinado en el futuro. Cuando llega ese instante se disparan y se eliminan.
- De disparo recurrente (periodical). Se programan para que se disparen a intervalos definidos. Después de cada disparo el temporizador se rearma, espera el intervalo y se vuelve a disparar. Esto se realiza continuamente hasta que se eliminan manualmente.

6.- Resumen

En esta unidad se han descrito de forma general la forma en que los SGBDs permiten incorporar programa internamente a fin de realizar proceso con los datos de forma interna (guiones).

Se han descrito, asimismo las características generales de los lenguajes de programación de guiones así como los mecanismos adicionales, como disparadores y cursores, que son un mecanismo especializado para la programación es bases de datos.

Tema 10
Programación de Guiones en MariaDB

Tabla de Contenidos

1.- Introducción

Los guiones permiten realizar procesamiento sobre los datos en la misma base de datos. En este documento se describen las particularidades de los guiones para el SGBD MariaDB, versión 10.x

1.1.- Comentarios

Los comentarios son bloques de texto que acompañan a los programas y que no son interpretados como instrucciones. Su función es acompañar al programa y servir de documentación de éste.

MariaDB soporta 3 tipos de comentarios:

- De una sola línea. Si la línea comienza por un corchete (#) o dos guiones seguidos de un espacio (- -), el resto de la línea, hasta el final de esta, se considera un comentario.

- De múltiples líneas. El comentario comienza por la combinación (`/*`) y finaliza con la combinación (`*/`).

Ejemplos:

```
# Esto es un comentario de una sola linea
-- Y esto también
/* Este comentario
abarca mas
de una linea */
```

1.2.- Tipos de datos

Los tipos de datos que tiene MariaDB son los mismos que los tipos de datos de las columnas de las tablas.

1.3.- Identificadores

Se deben seguir las siguientes reglas:

- Las variables que se utilicen fuera de un procedimiento almacenado deben comenzar por el carácter arroba (@). Las que se declaren dentro de un procedimiento almacenado no lo necesitan.
- Utilizar sólo letras inglesas (nada de eñes o vocales acentuadas), números y subrayado (_).
- Comenzar siempre los identificadores por letra (los números suelen estar prohibidos como primer carácter de un identificador y algunos sistemas utilizan el subrayado como primer carácter de variables especiales de sistema).
- El sistema tiene un sistema complejo para los identificadores pero lo más seguro es utilizar siempre la misma forma (mayúsculas o minúsculas) y suponer que dos nombres con distinta combinación de mayúsculas y minúsculas pueden ser considerados iguales por el sistema

1.4.- Declaración de variables

Si una variable va a ser utilizada dentro de un procedimiento almacenado, necesita ser declarada antes. Si se utiliza fuera de un procedimiento almacenado, la variable se crea la primera vez que se le asigna un valor.

Para declara una variable se utiliza la sentencia DECLARE, que tiene la siguiente forma:

```
DECLARE identificador1, identificador2, …, identificadorX tipo
[DEFAULT valor]
```

donde:

- `identificador1`, `identificador2`, …, `identificadorX` son las variables que se van a declarar. Hay que hacer notar que si hay más de una variable en la lista todas se crearán del mismo tipo y con el mismo valor por defecto, si se especificara.
- `tipo` es el tipo de la variable o variables y debe ser uno de los tipos del sistema.
- DEFAULT valor es el valor inicial opcional que tendrán las variables al ser creadas. Si no se especifica el valor será NULL.

Ejemplo:

Para declarar las variables `x` e `y` como enteras y la variable `nombre` como cadena de caracteres de 50 caracteres de longitud máxima, con valores por defecto 0 y la cadena vacía, respectivamente, las sentencias serían:

```
DECLARE x, y INTEGER DEFAULT 0;
DECLARE nombre VARCHAR(50) DEFAULT '';
```

1.5.- Constantes y literales

Las contantes y literales son iguales que las ya especificadas para las sentencias SQL de MariaDB.

1.6.- Operadores

Los operadores son idénticos a los ya especificados para MariaDB.

1.7.- Funciones predefinidas

Las funciones se deben consultar en el manual de MariaDB ya que son muchas para describirlas aquí.

1.8.- Asignación de valores

Para asignar un valor a una variable se utiliza la sentencia `SET` con la sintaxis:

```
SET variable = valor;
```

donde

- `variable` es la variable cuyo valor se desea modificar.
- `valor` es el nuevo valor de la variable. Puede ser una constante, expresión, otra variable, incluso una subconsulta que devuelva un único valor.

1.9.- Estructuras de control

1.9.1.- Secuencia

En MariaDB las secuencias se realizan utilizando la notación:

```
sentencia1;
sentencia2;
...
sentenciaN;
```

esto es, terminando las sentencias con punto y coma “;”. Es importante porque es el marcador que determina que una sentencia ha terminado.

1.9.2.- Condición simple

En MariaDB la condición simple se expresa con la sintaxis:

```
IF condicion1 THEN sentencias1
   [ELSEIF condicion2 THEN sentencias2]
   [ELSE sentencias3]
END IF
```

donde:

- `condicionX` es una condición que evalúa a `VERDADERO` o `FALSO`. Si la condición se evalúa a `VERDADERO` se ejecutan las sentencias tras la clausula `THEN`. Si es `FALSO` se ejecutan la clausula `ELSE` o `ELSEIF` correspondiente.
- `sentenciasX` es una secuencia.
- `ELSEIF` se ejecuta si la condición del `IF` o `ELSEIF` correspondiente ha evaluado a `FALSO`. Se evalúa la condición que tiene si se porta como si fuera un `IF`.
- `ELSE` se ejecuta incondicionalmente si la condición del `IF` o `ELSEIF` correspondiente ha evaluado a `FALSO`.

Ejemplo:

El siguiente trozo de código obtiene una edad y calcula la etapa educativa a la que pertenece el niño

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
# Se lee la edad por algun metodo que no se describe aqui
# Si la edad corresponde a Infantil
IF edad >= 0 AND < 6 THEN
     SET etapa = 'Educacion Infantil';
# Si no es Infantil, se comprueba si pertenece a primaria
ELSEIF edad >=6 AND edad < 12 THEN
     SET etapa = 'Educación Primaria';
# No es Infantil ni Primaria, ¿es Secundaria Obligatoria?
ELSEIF edad >= 12 AND edad < 16 THEN
     SET etapa = 'Educación Secundaria Obligatoria';
```

```
# Si no es ninguna de las anteriores, sólo queda Bachillerato
ELSE
    SET etapa = 'Bachillerato';
END IF;
```

1.9.3.- Condición múltiple

La condición múltiple tiene dos sintaxis distintas y con distintos significados. La primera es

```
CASE expresion
    WHEN valor1 THEN secuencia1
    WHEN valor2 THEN secuencia2
    ….
    WHEN valorN then secuenciaN
    ELSE secuencia_else
END CASE
```

donde

- `expresion` es una expresión que se evalúa a un único valor.
- `valor1`, `valor2`, …, `valorN` son valores literales. El resultado de la expresión es compara con cada valor. Si hay coincidencia se ejecuta la secuencia situada después de la clausula `THEN` correpondiente
- `secuencia1`, `secuencia2`, …, `secuenciaN`. Secuencia que se ejecuta si el valor de la expresión coincide con el valor correspondiente.
- `secuencia_else`. Esta secuencia se ejecuta si el valor de la expresión no coincide con ninguno de los valores indicados en las clausulas `WHEN`.

Ejemplo:

La misma clasificación por etapas educativas que antes pero resuelto utilizando CASE, sería:

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
...Se lee la edad por algún método...
CASE edad
# Casos de Educacion Infantil
    WHEN 0 THEN SET etapa = 'Educacion Infantil';
    WHEN 1 THEN SET etapa = 'Educacion Infantil';
    WHEN 2 THEN SET etapa = 'Educacion Infantil';
    WHEN 3 THEN SET etapa = 'Educacion Infantil';
    WHEN 4 THEN SET etapa = 'Educacion Infantil';
    WHEN 5 THEN SET etapa = 'Educacion Infantil';
# Casos de Primaria
```

```
    WHEN 6 THEN SET etapa = 'Educacion Primaria';
    WHEN 7 THEN SET etapa = 'Educacion Primaria';
    WHEN 8 THEN SET etapa = 'Educacion Primaria';
    WHEN 9 THEN SET etapa = 'Educacion Primaria';
    WHEN 10 THEN SET etapa = 'Educacion Primaria';
    WHEN 11 THEN SET etapa = 'Educacion Primaria';
# Casos de ESO
    WHEN 12 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 13 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 14 THEN SET etapa = 'Educacion Secundaria Obligatoria';
    WHEN 15 THEN SET etapa = 'Educacion Secundaria Obligatoria';
# EL resto es Bachillerato
    ELSE SET etapa = 'Bachillerato';
END CASE;
```

La sintaxis alternativa es muy similar a la de la condición simple:

```
CASE
    WHEN condicion1 THEN secuencia1
    WHEN condicion2 THEN secuencia2
    ....
    WHEN condicionN THEN secuenciaN
    ELSE secuencia_else;
END CASE;
```

Aqui la diferencia es que no se proporciona expresión que de el valor a comparar sino que cada caso (`WHEN`) evalúa su propia condición. La primera que evalúa a `VERDADERO` ejecuta la secuencia asociada con la clausula `THEN`.

Si ninguna de las condiciones se cumple, entonces se ejecuta la secuencia asociada a la clausula `ELSE`.

Ejemplo:

El mismo ejemplo, quedaría ahora:

```
DECLARE edad INTEGER DEFAULT 0;
DECLARE etapa VARCHAR(50) DEFAULT '';
# Se lee la edad por algún método
# Se hace una linea por caso con la condicion correspondiente
CASE
    WHEN edad >= 0 AND < 6 THEN SET etapa = 'Educacion Infantil';
    WHEN edad >=6 AND edad < 12 THEN SET etapa = 'Educacion
Primaria';
    WHEN edad >= 12 AND edad < 16 THEN SET etapa = 'Educacion
Secundaria Obligatoria';
```

```
    ELSE SET etapa = 'Bachillerato';
END CASE;
```

1.9.4.- Ciclo Mientras

El ciclo Mientras tiene la sintaxis:

```
WHILE condicion DO
    secuencia
END WHILE
```

donde:

- `condicion` es la condición que se evalúa para determinar el fin del ciclo. Si la condición evalúa al valor `VERDADERO`, se ejecutan la secuencia. Si evalúa a `FALSO` se sale del ciclo. La condición se evalúa antes de realizar las sentencias.
- `sentencia` es una secuencia de sentencias que se ejecutará ninguna, una o varias veces, dependiendo de la condición.

Ejemplo:

Supongamos que deseamos introducir en una tabla la secuencia de los primeros 100 números. Se podría hacer con el siguiente código:

```
# Contador para el ciclo (se inicializa a cero)
DECLARE c INTEGER DEFAULT 0;
# Hasta que el valor del contador llegue a 100
WHILE c < 100 DO
    # Se inserta el valor en la tabla secuencia
    INSERT INTO secuencia (valor) VALUES (c);
    # Se incrementa el contador al siguiente valor
    SET c = c + 1;
END WHILE;
```

1.9.5.- Ciclo Repetir

La sintaxis sería:

```
REPEAT
    secuencia
UNTIL condicion
END REPEAT
```

donde:

- `secuencia` es la secuencia de sentencias que se ejecutará como parte del ciclo. La secuencia se ejecutará al menos una vez.

- `condicion`. Su valor se evalúa al final de cada iteración del ciclo. Si produce un valor `VERDADERO` se termina el ciclo. Si produce un valor `FALSO`, se realiza otra iteración.

Ejemplo:

El mismo ejemplo que para MIENTRAS, utilizando `REPEAT`:

```
# Se declara el contador y se inicializa a cero
DECLARE c INTEGER DEFAULT 0;
REPEAT
     #Se inserta el valor en la tabla secuencia
     INSERT INTO secuencia (valor) VALUES (c);
     # Se incrementa el contador en uno
     SET c = c + 1;
# El valor 100 no entra, como en el WHILE porque se utiliza >=
UNTIL c >= 100
END REPEAT;
```

Nótese que, dado que el ciclo se ejecuta al menos una vez y que la condición debe valer `VERDADERO` para *salir* del ciclo, la condición es distinta a la del ejemplo `MIENTRAS`. Además, el ciclo no se termina con la clausula `UNTIL`, como suele ser común en los lenguajes de programación de propósito general sino que necesita terminarse con `END REPEAT`.

1.10.- Subprogramas

MariaDB soporta los dos tipos de subprogramas, procedimientos y funciones.

1.10.1.- Procedimientos

Los procedimientos se declaran con la siguiente sintaxis:

```
DELIMITER //
CREATE PROCEDURE nombreproc (listaparametros)
BEGIN
     secuencia
END//
DELIMITER ;
```

donde:

- Las líneas `DELIMITER` se utilizan para modificar el delimitador de sentencia por defecto temporalmente a un nuevo valor distinto de `;`, que se usa dentro del procedimiento. El valor temporal habitual es `//` que no se presenta en las sentencias SQL normales ni es un operador válido.
- `nombreproc` es el nombre que se le va a proporcionar al nuevo procedimiento. Debe ser distinto al nombre de cualquier otro procedimiento que exista ya en la base de datos.

- `listaparametros` es la lista de parámetros del procedimiento. Puede estar vacía, en cuyo caso los paréntesis si deben ir, aunque sin nada entre ellos o puede constar de uno o más parámetros separados por comas. Cada parámetro consta de las siguientes partes:
    - **Indicador de dirección.** Indica si el parámetro es de entrada (`IN`), de salida (`OUT`) o de entrada/salida (`INOUT`). Si un parámetro es de entrada, su valor no se puede modificar dentro del cuerpo del procedimiento, si es de salida, su valor no se puede leer pero si escribir dentro del procedimiento y si es de entrada/salida se puede tanto leer como escribir dentro del procedimiento.
    - **Nombre del parámetro.** Rigen las mismas reglas que para los nombres de variables.
    - **Tipo del parámetro**. Igual que si fuera una variable
- `secuencia` es la secuencia que compone el cuerpo del procedimiento.

Ejemplo:

Supongamos que se tiene la tabla

empleado								
codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000

y se desea crear un procedimiento `ventas_por_oficina` que obtenga el total de ventas de una oficina, a partir del código de oficina que se le proporciona. La definición del procedimiento sería:

```
DELIMITER //
CREATE PROCEDURE ventas_por_oficina (IN ofi INTEGER, OUT ventas
INTEGER)
BEGIN
     # Obtiene la suma de las ventas de la oficina indicada y
almacena el resultado en el parametro de salida ventas
     SET ventas = (SELECT SUM(ventas_obtenidas) FROM empleado WHERE
oficina = ofi);
END//
DELIMITER ;
```

1.10.2.- Llamadas a procedimientos

Para llamar un procedimiento se utiliza la sentencia `CALL`, con la siguiente sintaxis:

```
CALL nombre_procedimento(listaparametros)
```

donde

- `nombre_procedimiento` es el procedimiento a invocar.
- `listaparametros` son los parámetros del procedimiento. Si no tiene parámetros hay que dejar la lista vacía `()`. Los parámetros de entrada pueden ser cualquier expresión pero los parámetros de salida o de entrada/salida deben ser variables ya que deben poder recibir un valor desde el procedimiento almacenado.

Ejemplo:

Para invocar el procedimiento creado en el apartado anterior para la oficina 12, se podría realizar de la siguiente manera:

```
# Invoca el procedimiento. Se pasa una nueva variable @ventas para
que reciba el valor calculado en el procedimiento
CALL ventas_por_oficina(12,@ventas);
# Forma "extraña" de SELECT. Si se hace SELECT y una variable se
genera un resultado con una columna y una fila conteniendo el valor
de la variable. Util para depuracion
SELECT @ventas;
```

1.10.3.- Modificar o eliminar un procedimiento

Un procedimiento se elimina utilizando la sintaxis:

```
DROP PROCEDURE nombre;
```

donde `nombre` es el nombre del procedimiento a eliminar. A partir de ese momento el procedimiento ya no se puede volver a invocar.

MariaDB no permite el modificar un procedimiento ya almacenado por lo que la forma de hacerlo es eliminarlo (utilizando `DROP PROCEDURE`) y volviéndolo a crear de nuevo con el nuevo contenido.

Ejemplo:

Para eliminar el procedimiento `ventas_por_oficina` ya creado se utilizaría:

```
DROP PROCEDURE ventas_por_oficina;
```

1.10.4.- Funciones

Existen dos formas de crear funciones en MariaDB:

- **Forma nativa.** En esta forma hay que desarrollar la función utilizando un lenguaje de programación estándar (C, en la mayoría de los casos) que conecta con el servidor MariaDB para realizar el cálculo requerido. Es la opción más potente pero más compleja de programa.
- **Forma SQL.** En esta forma se pueden utilizar el lenguaje de guiones de MariaDB para realizar la función pero el resultado es más limitado. Por ejemplo, con este método no se pueden crear

funciones agregadas, esto es, funciones que tomen un grupo de valores de tamaño indefinido y devuelvan un resultado. Para ello hay que desarrollar la función en forma nativa.

En esta exposición sólo se describirán las funciones en forma SQL, que se definen de una forma muy parecida a la empleada para describir los procedimientos. Esta sintaxis es la siguiente:

```
DELIMITER //
CREATE FUNCTION nombre_funcion (listaparametros) RETURNS tipo
[NOT] DETERMINISTIC
BEGIN
    secuencia
END//
DELIMITER ;
```

donde

- `nombre_funcion` es el nombre de la funcion, igual que para un procedimiento.
- `listaparametros` es la lista de parámetros que se define y funciona parecida a la de los procedimientos exceptuando el detalle de que todos los parámetros son de entrada y no se puede indicar ninguno de los calificadores `IN`, `OUT` o `INOUT`.
- `tipo` es el tipo del valor que devuelve la función. Este valor sustituirá la aparición de la función dentro de una expresión.
- `DETERMINISTIC` especifica que la función es determinista, esto es, que siempre que se le proporcione una combinación dada de valores de entrada producirá siempre el mismo valor de salida. Si no cumple estas condiciones hay que declararla como `NOT DETERMINISTIC`. Esto afecta a la forma en que MariaDB utiliza las funciones en las expresiones por lo que es importante el definirlo correctamente y de manera veraz, ya que MariaDB no comprueba que la función es o no determinística sino que se fía del programador. En general, funciones que se basan en información de tablas que pueden cambiar con el tiempo o que se basan en tiempos no suelen ser deterministas. Por ejemplo, una función que devuelve el día de la semana en que se invoca no es determinista. En cambio otra que devuelve el día de la semana que corresponde a una fecha dada si que lo es.
- `secuencia`. Secuencia de sentencias.

Hay que hacer notar que la misma restricción con el uso de los delimitadores que se tenía con los procedimientos se aplica a las funciones por lo que hay que utilizar el mecanismo de sustitución temporal del delimitador con la sentencia `DELIMITER`.

Para devolver el valor de retorno de la función se emplea la sentencia `RETURN`, con la siguiente sintaxis:

```
RETURN expresion;
```

donde `expresion` es la expresión que se evalúa y su valor se devuelve como valor de retorno de la función. Si no es del tipo especificado en la clausula `RETURNS` se intenta convertir a dicho tipo, produciéndose un error si esta conversión no es posible. Además se termina la ejecución de la función inmediatamente.

Ejemplo:

Supongamos la tabla ya utilizada anteriormente:

empleado								
codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Alvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	Jose Gonzalez	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutierrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	Maria Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Victor	41	22	Representante	13/01/10	104	500000	760000

Imaginemos que deseamos obtener un listado en que se quiere ver el nivel de rendimiento de los empleados, clasificándolos en:

- MUY BAJO. Si sus ventas no llegan al 50% de su cuota
- BAJO. Si sus ventas estan entre el 50 y 100% de la cuota.
- NORMAL. Si sus ventas están en el 100% de la cuota.
- BUENO. Si sus ventas están entre el 100% y el 150% de su cuota
- MUY BUENO. Si sus ventas son superiores al 150% de su cuota.

Para ayudar en esta tarea se va a crear una función a la que se le proporciona la cuota y las ventas y devuelve la clasificación. Esta función se llamará rating:

```
DELIMITER //
CREATE FUNCTION rating(cuota DECIMAL(10,2), ventas DECIMAL(10,2))
RETURNS VARCHAR(10)
DETERMINISTIC
BEGIN
    # La ratio es la relación entre las ventas y la cuota. Se
declara como variable para que sólo haya que calcularlo una vez
    DECLARE ratio DECIMAL(10,2) DEFAULT 0;
    # El resultado es el valor que se va a devolver
    DECLARE clase VARCHAR(10);
    # Se calcula el ratio
    SET ratio = ventas / cuota;
    /* Se va a utilizar una secuencia de IF/ELSEIF/ELSE.
       Se podría haber hecho tambien con CASE */
```

```
    IF ratio < 0.5 THEN
         SET clase = 'MUY BAJO';
    ELSEIF ratio >= 0.5 AND ratio < 1 THEN
         SET clase = 'BAJO';
    ELSEIF ratio = 1 THEN
         SET clase = 'NORMAL';
    ELSEIF ratio > 1 AND ratio < 1.5 THEN
         SET clase = 'BUENO';
    ELSE
         SET clase = 'MUY BUENO';
    END IF;
    # Se devuelve el resultado
    RETURN clase;
END//
DELIMITER ;
```

1.10.5.- Uso de funciones

Las funciones se utilizan incorporándolas en expresiones, de la misma forma en que se utilizan las funciones predefinidas.

Ejemplo:

Si se desea ver el ranking actual de los empleados con su calificación, se podría utilizar la sentencia:

```
SELECT nombre, rating(cuota, ventas) AS calificacion FROM empleado;
```

que devolvería el resultado:

nombre	calificacion
Antonio Viguer	BUENO
Alvaro Jaumes	BUENO
Juan Rovira	BUENO
Jose Gonzalez	BAJO
Vicente Pantall	BUENO
Luis Antonio	BUENO
Jorge Gutierrez	BAJO
Ana Bustamante	BUENO
Maria Sunta	BUENO
Juan Victor	MUY BUENO

1.11.- Tratamiento de excepciones

El tratamiento de excepciones en MariaDB se realiza mediante la sintaxis:

```
DECLARE accion HANDLER FOR lista_errores sentencia;
```

donde:

- `accion` es la acción que MariaDB realizará *despues* de procesar el error. Puede ser una de:

    - `CONTINUE`. La ejecución continuará por la sentencia siguiente a la que produjo el error.
    - `EXIT`. Se abortará la ejecución del bloque `BEGIN...END` en que se encuentre la declaración. Usualmente esto significa salir del procedimiento o función.
- `lista_errores` es una lista de los errores que el manejador que se está declarando debe procesar. Es una lista separada por comas de las especificaciones de los errores. Estas especificaciones pueden ser:
    - Un número entero, equivalente a un código de error interno de MariaDB. Para ver los códigos de error y su significado hay que consultar el manual de MariaDB. No utilizar el código 0 nunca.
    - Un codigo SQLSTATE. Estos códigos están definidos en el estándar SQL y son más interoperables entre servidores. Es la opción recomendada siempre que se pueda en lugar de los códigos internos de MariaDB. El código se da en la forma `SQLSTATE 'codigo'`, donde `codigo` es un número de 5 cifras siempre. No hay que utilizar los SQLSTATEs que comienzan por 00 ya que indican éxito, no error.
    - Un identificador. Este identificador debe corresponder con una condición de error definida por el usuario. En este documento no se tratarán las condiciones definidas por el usuario.
    - `SQLWARNING`. Indica que se capturarán todos los errores de tipo Advertencia (WARNING). Estos errores se corresponden con SQLSTATEs que comienzan por 01 y suelen corresponder con condiciones de error que no son críticas y que permiten continuar con la ejecución del programa, aunque avisa de que algo merece revisarse porque puede ser fuente de problemas.
    - `NOT FOUND` (dos palabras). Indica que se capturarán los errores de cursor (ver cursores más adelante en este documento). Se corresponde con los SQLSTATEs que comienzan por 02.
    - `SQLEXCEPTION`. Indican que se capturaran los errores graves, cuyo SQLSTATE no comienza por 00, 01 ó 02. Suelen indicar una condición de error grave y que impide la continuación de la ejecución.

Para ver una lista de los códigos de error o SQLSTATEs para la versión 10.x del servidor, se puede consultar la página:

https://mariadb.com/kb/en/mariadb-error-codes/

Ejemplo:

Una forma clásica de detectar cuando se utilizan cursores que se ha llegado al final de los resultados es utilizando una variable booleana (`VERDADERO`/`FALSO`) que se indica al programa que se ha llegado al final de los resultados. Esta variable inicialmente vale FALSO y se cambia a VERDADERO desde un manejador de excepciones de la forma siguiente:

```
# Se crea la variable y se inicializa a FALSO directamente
DECLARE hecho INTEGER DEFAULT FALSE;
/* Declaración del manejador de errores. Se declara de tipo CONTINUE
porque no queremos detener la ejecución y la condición de error es
NOT FOUND. La única sentencia que se ejecuta es SET hecho = TRUE que
modifica el valor de la variable */
DECLARE CONTINUE HANDLER FOR NOT FOUND SET hecho = TRUE;
# Codigo que abre y usa el cursor
...
# Se comprueba que no se ha llegado al final. Si es asi, termina
IF hecho THEN .....
```

Relación 01 de ejercicios

2.- Cursores

Para poder utilizar un cursor en MariaDB lo primero que hay que hacer es declararlo. Para ello se utiliza la siguiente sintaxis:

```
DECLARE nombre CURSOR FOR sentencia_select;
```

donde

- `nombre` es el nombre que se va a dar al cursor. Este nombre se utilizará para operar sobre el cursor.
- s`entencia_select`. Sentencia `SELECT` completa. Cuando se active el cursor se utilizará para recorrer los resultados de esta consulta fila a fila.

Una vez declarado ya se puede utilizar. Hay que tener en cuenta que en MariaDB los cursores no se pueden actualizar, esto es, son sólo lectura y que no tienen vuelta atrás, esto es, el cursor se mueve de una fila a otra en una sola dirección, no pudiéndose volver a una fila ya visitada.

Una vez declarado el cursor hay que *abrirlo* para comenzar a utilizarlo. El proceso de apertura realiza efectivamente la consulta asociada al cursor y coloca éste en la primera fila, si es que la consulta devuelve alguna fila. Para abrir un cursor hay que utilizar la sentencia `OPEN`, con la sintaxis:

```
OPEN nombre;
```

donde `nombre` es el nombre de un cursor previamente declarado. Si el cursor no está declarado o la sentencia no es válida se producirá un error.

Una vez abierto, ya se puede acceder a información utilizando el cursor. Para ello se utiliza la sentencia `FETCH`, con la siguiente sintaxis:

```
FETCH nombre INTO var1, var2, ..., varN;
```

donde:

- `nombre` es el nombre de un cursor declarado y abierto.
- `var1`, `var2`, `varN` son variables donde se almacenarán los contenidos de la columna 1, 2, etc de la fila actual del cursor. Se pueden especificar tantas variables como columnas estén especificadas en la sentencia `SELECT` asociada al cursor.

Un efecto adicional de la ejecución de `FETCH` es que el cursor se desplaza a la siguiente fila de la tabla después de acceder a los campos de la actual, con lo que el cursor queda listo para otra lectura.

Por último, cuando se finaliza el trabajo con un cursor hay que *cerrarlo* para liberar los recursos empleados en el mismo. Si no se realiza esta operación, el cursor se libera automáticamente al finalizar el bloque `BEGIN`..`END` en el que está declarado. De todas formas se recomienda liberarlo explícitamente utilizando la siguiente sintaxis:

```
CLOSE nombre;
```

donde `nombre` es el nombre de un cursor ya declarado y abierto. Si no se dan estas dos condiciones, se producirá un error.

Con estas sentencias se manipulan los cursores pero aún queda pendiente el detalle de finalizar el proceso del cursor, esto es, ¿cuando se sabe que se han procesado todas las filas accedidas a través del cursor?

La respuesta ya se dio en capítulos anteriores: A través de excepciones. Mas concretamente, el sistema lanza una excepción con el `SQLSTATE '02000'` para indicar que se llegó al final del resultado accedido por el cursor. Esto puede ocurrir desde la primera ejecución de la sentencia `FETCH` si el resultado obtenido no contiene ninguna fila.

Por lo tanto hay que declarar un manejador de excepciones que capture esta excepción en concreto y utilizarlo para terminar el proceso. La técnica más usual, ya medio descrita anteriormente es utilizar una variable que se utiliza como *bandera*. La variable está en un estado inicialmente que indica que el proceso puede seguir y se modifica su valor desde el manejador de excepciones, indicando al procedimiento que el proceso debe finalizar. En el siguiente ejemplo completo de uso de cursores se describe un procedimiento típico que utiliza todas estas técnicas.

Ejemplo:

Supongamos que se tiene la tabla de ejemplo:

empleado					
codigo	**nombre**	**apellido1**	**apellido2**	**email**	**puesto**
1	Marcos	Magaña	Perez	marcos@jardineria.es	Director General
3	Alberto	Soria	Carrasco	asoria@jardineria.es	Subdirector Ventas
4	Maria	Solís	Jerez	msolis@jardineria.es	Secretaria
6	Juan Carlos	Ortiz	Serrano	cortiz@jardineria.es	Representante Ventas
8	Mariano	López	Murcia	mlopez@jardineria.es	Representante Ventas
9	Lucio	Campoamor	Martín	lcampoamor@jardineria.es	Representante Ventas
11	Emmanuel	Magaña	Perez	manu@jardineria.es	Director Oficina
12	José Manuel	Martinez	De la Osa	jmmart@hotmail.es	Representante Ventas
14	Oscar	Palma	Aceituno	opalma@jardineria.es	Representante Ventas
16	Lionel	Narvaez		lnarvaez@gardening.com	Representante Ventas
19	Walter Santiago	Sanchez	Lopez	wssanchez@gardening.com	Representante Ventas
21	Marcus	Paxton		mpaxton@gardening.com	Representante Ventas
23	Nei	Nishikori		nnishikori@gardening.com	Director Oficina
25	Takuma	Nomura		tnomura@gardening.com	Representante Ventas
28	John	Walton		jwalton@gardening.com	Representante Ventas
29	Kevin	Fallmer		kfalmer@gardening.com	Director Oficina
30	Julian	Bellinelli		jbellinelli@gardening.com	Representante Ventas
31	Mariko	Kishi		mkishi@gardening.com	Representante Ventas

y se desea crear un procedimiento que obtenga una lista de correo a partir de los correos de los empleados. Una lista de correo consta de los correos separados por punto y coma ";". La función quedaría

```
DELIMITER //
CREATE PROCEDURE listaCorreo(OUT lista VARCHAR(5000))
BEGIN
     # Variable bandera para marcar el fin del resultado
     DECLARE fin INTEGER DEFAULT FALSE;
     # Variable para leer el correo de cada fila
     DECLARE correo VARCHAR(100);
     # Variable para saber si la lista tiene más de un elemento o no
(para el proceso de separador)
     DECLARE masdeuno INTEGER DEFAULT FALSE;
     # Cursor
     DECLARE empleados CURSOR FOR SELECT DISTINCT email FROM
empleado;
     # Manejador de errores para detectar el fin de resultad
     DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = TRUE;

     # Comienza el proceso. Se abre el cursor
     OPEN empleados;
     # Y se inicializa la lista
     SET lista = '';
     # Ciclo que se repite hasta el fin de los resultados
```

```
    REPEAT
         # Se lee la fila
         FETCH empleados INTO correo;
         # Si no se llego al final del resultad
         IF NOT fin THEN
              # Si hay mas de uno
              IF masdeuno THEN
                   # Añade el separador ";"
                   SET lista = CONCAT(lista, ';');
              ELSE
                   # Ahora si hay mas de uno
                   SET masdeuno = TRUE;
              END IF;
              # Añade el elemento
              SET lista = CONCAT(lista, correo);
         END IF;
    UNTIL fin
    END REPEAT;

    # Cierra el cursor
    CLOSE empleados;
END//
DELIMITER ;
```

Relación 02 - Cursores

3.- Disparadores

En MariaDB, los disparadores o triggers se crean utilizando la siguiente sintaxis:

```
CREATE TRIGGER nombre momento operacion ON tabla FOR EACH ROW
[orden] sentencia;
```

donde

- `nombre` es el nombre del disparador. Debe ser único entre los disparadores de una misma base de datos, aunque puede haber disparadores con el mismo nombre en distintas bases de datos.
- `momento` es el momento en que se dispara relativo a la operación que provoca el disparo. Puede ser uno de:
  - `BEFORE`. El disparador se dispara antes de que se realice la operación.
  - `AFTER`. El disparador se dispara después de que se realice la operación.
- `operacion` es la operación que provoca el disparo. Puede ser una de:

    - `INSERT`. Se inserta un registro en la tabla.
    - `UPDATE`. Se modifica un registro de la tabla.
    - `DELETE`. Se elimina un registro de la tabla.
- `tabla` es la tabla sobre la que se crea el disparador. No se pueden crear disparadores sobre varias tablas. Hay que crear uno por cada tabla que se quiera monitorizar.
- `orden` es opcional y especifica, si hay más de un disparador para la misma operación y momento, cual de ellos debe ejecutarse primero. Si no se especifica, los disparadores se ejecutan en el mismo orden en que se definieron. La sintaxis es:
    - `FOLLOWS otro_disparador`. Indica que este disparador se debe lanzar después del disparador con nombre `otro_disparador`.
    - `PRECEDES otro_disparador`. Indica que este disparador se debe lanzar antes del disparador con nombre `otro_disparador`.
- `sentencia` es la sentencia que se ejecuta cuando se lanza el disparador. Si se necesita más de una sentencia hay que utilizar un bloque `BEGIN...END` como en los cuerpos de los procedimientos y funciones. Usualmente se utiliza una sola sentencia o se llama a un procedimiento.

Dado que un disparador se invoca en el momento que se va a producir un cambio o este ya se ha producido, es importante que la sentencia que procesa el disparador pueda acceder a los valores de los campos antes y después de que se modifiquen, a fin de saber la naturaleza de los cambios y actuar en consecuencia. Para ello se deben preceder los nombres de los campos con los alias `OLD.campo` y `NEW.campo` para indicar el valor del campo antes y después de la modificación, respectivamente. Si el campo no está afectado por la actualización, `OLD.campo` y `NEW.campo` tendrán el mismo valor.

Ejemplo:

Supongamos una tabla con la siguiente definición:

```
CREATE TABLE usuario(id INTEGER PRIMARY KEY, nombre VARCHAR(50) NOT
NULL, puntos INTEGER DEFAULT 0)
```

que contiene información sobre un usuario de una plataforma on-line. El usuario puede obtener puntos participando y los puede gastar. Se decide que el saldo de puntos debe estar siempre entre 0 y 100. Para asegurar esta regla se puede realizar un disparador que asegure su cumplimiento. El disparador se crearía de la siguiente forma:

```
# Para poder hacer el bloque BEGIN..END
DELIMITER //
CREATE TRIGGER comp_saldo BEFORE UPDATE ON usuario FOR EACH ROW
BEGIN
```

```
    # Si el nuevo saldo es negativo
    IF NEW.saldo < 0 THEN
         # Lo ajusta a cero
         SET NEW.saldo = 0;
    # Si no es negativo comprueba que no sea mayor que 100
    ELSEIF saldo > 100 THEN
         # Si lo es, lo recorta a 100
         SET NEW.saldo = 100;
    END IF;
    # Si el saldo no es ni negativo ni mayor que 100 no se modifica
END;//
DELIMITER ;
```

Para eliminar un disparador, la sintaxis es muy simple:

```
DROP TRIGGER nombre;
```

donde `nombre` es el nombre del disparador a eliminar.

Ejemplo:

Para eliminar el disparador creado en el ejemplo anterior se utilizaría:

```
DROP TRIGGER comp_saldo;
```

4.- Temporizadores

En MariaDB, los temporizadores se denominan *eventos*. Para crear un evento se emplea la siguiente sintaxis:

```
CREATE EVENT nombre ON SCHEDULE tiempo [ON COMPLETION PRESERVE]
[activacion] DO sentencia;
```

donde:

- `nombre` es el nombre del evento. Debe ser único entre los eventos de la base de datos, esto es, no puede haber dos eventos con el mismo nombre en la misma base de datos.
- `tiempo` es la definición de cuando se va a activar el temporizador. Se puede indicar de dos formas:
    - `AT timestamp`. Donde `timestamp` es una fecha y hora concretos en la que se realizará el disparo.
    - `EVERY intervalo`. Donde `intervalo` es un intervalo de tiempo que se puede expresar en unidades de tiempo desde microsegundos a años. En este caso es necesario añadir la clausula `ON COMPLETION PRESERVE` para que el temporizador no se elimine después del primer disparo.

- `activacion`. Indica si el temporizador se activa inmediatamente cuando es creado (el comportamiento por defecto) o se crea desactivado (se puede activar más tarde). Puede valer:
  - `ENABLE`. Para que el temporizador se cree activado (la opción por defecto).
  - `DISABLE`. Para que el temporizador se cree desactivado.
- `sentencia`. Es la sentencia que se ejecutará cuando se dispare el evento. Del mismo modo que ocurre con los disparadores, si se desea realizar algo que necesite más de una sentencia, hay que utilizar un bloque `BEGIN..END` ó utilizar llamadas a procedimientos almacenados.

Para que funcionen los eventos en MariaDB debe estar activado un módulo del SGBD denominado planificador de eventos (event scheduler). El estado del planificador se controla mediante la variable `event_scheduler` Esto se puede realizar de dos maneras:

- Por configuración. Mediante un ajuste en el archivo `my.cnf`, que controla la inicialización del SGBD (introduciendo una línea con la sintaxis: `event_scheduler=ON`)
- Desde comandos: Modificando la variable utilizando la sentencia `SET` (`SET GLOBAL event_scheduler="ON"`)

Ejemplo:

Supongamos que la tabla `usuarios` dispone de varios campos para el nombre de usuario, contraseña, etc. y que además, para tareas de control de cuentas no usadas, se dispone de un campo `ultimoAcceso`, de tipo `TIMESTAMP` que contiene la fecha y hora del último acceso del usuario. Para crear un evento que se dispare el día 1 de Enero a medianoche y que elimine las cuentas que no se han accedido en 6 meses, la sentencia sería:

```
CREATE EVENT limpia_cuentas_2018 ON SCHEDULE AT '2018-01-01
00:00:00' DO DELETE FROM usuarios WHERE ultimoAcceso < '2017-06-30
23:59:59';
```

Si se deseara realizar esta misma tarea pero cada mes, habría que utilizar:

```
CREATE EVENT limpia_cuentas_mensual ON SCHEDULE EVERY 1 MONTH ON
COMPLETION PRESERVE DISABLE DO DELETE FROM usuarios WHERE
DATEDIFF(NOW(), ultimoAcceso) > 180;
```

En este último cambia la condición del borrado porque debe funcionar correctamente en todas las ejecuciones. Se comprueba que no hayan pasado más de 180 días desde el último acceso. Asimismo se ha creado el evento sin activar a fin de activarlo justo a comienzos de mes.

Para eliminar un temporizador de forma manual se utiliza la sentencia `DROP EVENT`, con la sintaxis:

```
DROP EVENT [IF EXISTS] nombre;
```

donde nombre es el nombre de un evento. La clausula `IF EXISTS` previene que ocurra un error si el evento no existe.

Ejemplo:

Para eliminar el evento `limpia_cuentas_2018` se utilizaría la sentencia:

```
DROP EVENT IF EXISTS limpia_cuentas_2018;
```

Por último, para modificar un evento existente, la sintaxis es:

```
ALTER EVENT nombre [ON SCHEDULE tiempo] [RENAME TO nuevo_nombre]
[activacion] [DO sentencia];
```

donde:

- `nombre` es el nombre del evento existente que se desea modificar.
- `tiempo`. Nuevo momento o periodo de activación.
- `nuevo_nombre`. El nuevo nombre del evento. No debe existir un evento con ese nombre en la base de datos.
- `activacion`. Nuevo estado de la activación del evento.
- `sentencia`. Nuevo comando del evento.

Todos los apartados son opcionales. Si alguno no se especifica, su valor no se modifica. Los parámetros `tiempo`, `activacion` y `sentencia` se especifican como se describió en la sentencia `CREATE EVENT`.

Ejemplo:

Supongamos que se desea activar ya el evento `limpia_cuentas_mensual`. Se haría con la sentencia:

```
ALTER EVENT limpia_cuentas_mensual ENABLE;
```

5.- Resumen

En este documento se ha intentado describir los mecanismos proporcionados por MariaDB para implementar la programación de guiones de forma básica. Aun así, los mecanismos propuestos permiten un nivel de manipulación bastante grande que permitirán mejorar el rendimiento de las bases de datos.