Diseño de Bases de Datos (Unidades 3 y 2)

Adaptado para MySQL Server

Contenido

Diseño de Bases de Datos

Los pasos del diseño de una Base de Datos, representados en la siguiente figura, se pueden resumir en

- **Recolección y análisis de requerimientos**: En este paso recogemos información del sistema para el que debemos diseñar la Base de Datos.

- **Diseño conceptual**: Una vez recogidos todos los requisitos y conocido el problema, realizamos un primer esquema conceptual en algún lenguaje de alto nivel como es el *Modelo Entidad-Relación*

- **Diseño lógico**: El diseño conceptual debe ser ahora transformado en un diseño lógico, que es la transformación de un modelo conceptual a un modelo de datos concreto con el fin de poder representar el problema, más adelante, en algún software concreto. En nuestro caso usaremos el *Modelo Relacional*.

- **Diseño físico**: En este punto debemos aplicar el modelo lógico de datos del punto anterior sobre un SGBD concreto. Dependiendo del diseño físico escogido, tendremos un abanico de posibilidades en cuanto al software disponible. En nuestro caso hemos optado por un modelo relacional por lo que tendremos que escoger entre los SGBD relacionales disponibles.

Modelo Entidad/Relación

Es un modelo de datos que representa la realidad a través de *entidades* , que son objetos que existen y se distinguen de otros por sus características, que llamamos *atributos*. Además, estas entidades podrán o no estar relacionadas unas con otras a través de lo que se conoce como *relación*. Hay que tener en cuenta que se trata solamente de un modelo de representación, por lo que no tiene correspondencia real con ningún sistema de almacenamiento. Se utiliza en la etapa de Análisis y Diseño de una Base de Datos, por lo que habrá que convertirla a otro modelo antes de poder empezar a trabajar con ella.

Elementos principales

El modelo Entidad-Relación es un lenguaje de modelado y como todo lenguaje tiene sus propios términos y sintaxis. A diferencia de los lenguajes textuales, el modelo Entidad-Relación utiliza un lenguaje de símbolos:

Entidad

Una *entidad* es un objeto que existe en una realidad que queremos representar, por ejemplo, un alumno, que se distingue de otro por sus características como pueden ser: el nombre, los apellidos, el número de expediente, . . .

Las entidades se representan por el siguiente símbolo:

Entidad

Esas características que hacen que unas entidades se distingan de otras, son los *atributos*. El nombre, los apellidos y el número de expediente serían atributos de la entidad alumno. Los atributos se representan por el siguiente símbolo:

Relación

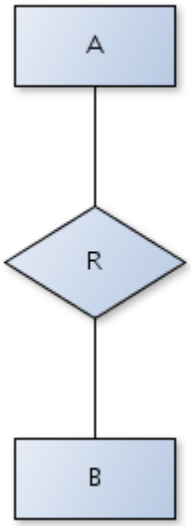
A su vez, podemos relacionar unas entidades con otras a través de lo que se conoce como *relación*. Por ejemplo, dos entidades alumno y asignatura podrían estar relacionadas entre sí puesto que un alumno *cursa* una asignatura (o varias). Conviene resaltar que una relación entre dos entidades no expresa obligatoriedad de relación sino **posibilidad de relacionarse**.

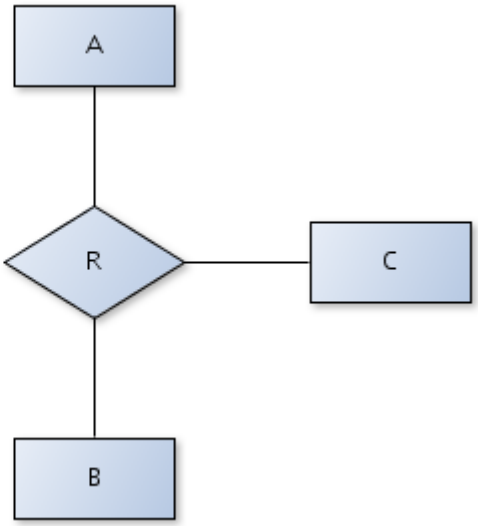
En este caso, no será necesario que todos los alumnos cursen una asignatura o que una asignatura sea cursada por todos los alumnos para que la relación se establezca. Por tanto, en este caso se establece que entre esas dos entidades existe una relación a la que podríamos llamar *cursa*. Las relaciones se representan por el siguiente símbolo:

Relaciones binarias

Existirán situaciones en las que a la hora de analizar los requisitos interpretemos que en una misma relación hay más de dos entidades involucradas. A este tipo de relaciones se les llama relaciones ternarias, cuaternarias, etc.

Hay que tener en cuenta una cosa: todo modelo E/R se transformará en el modelo Relacional solamente a tablas, independientemente del tipo de relación. Sin embargo, no tendrá la misma facilidad transformar a tablas una relación ternaria que una binaria; probablemente debamos volver a hacer un estudio a la hora de transformar el modelo E/R. Para evitar esto, trataremos de plantear siempre relaciones binarias frente a relaciones ternarias.

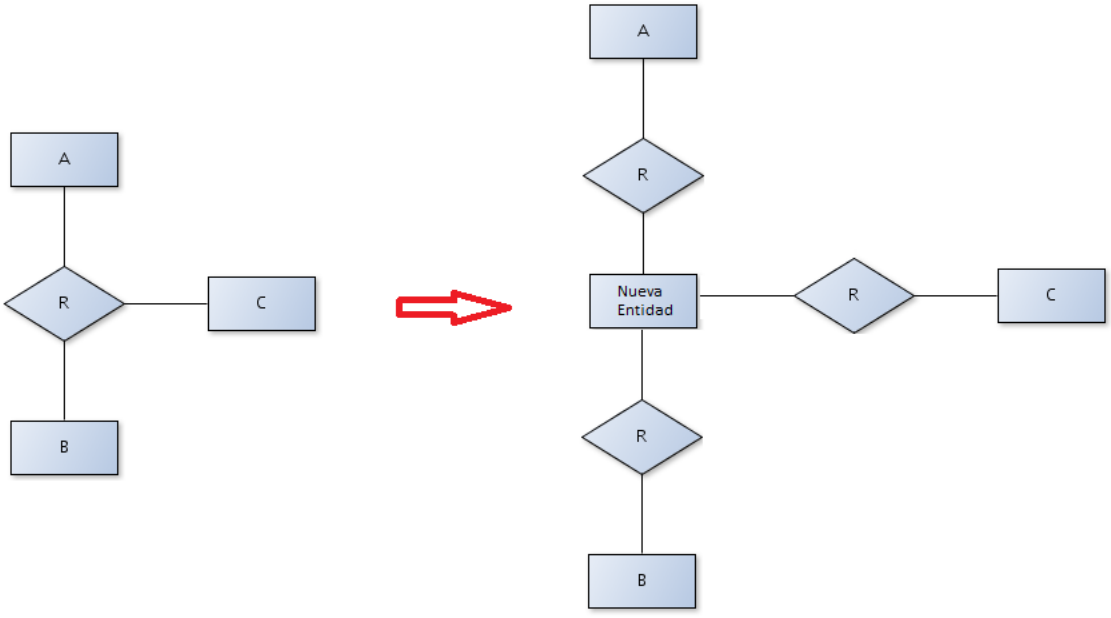




Relaciones binarias y ternarias entre entidades

En las situaciones en las que nuestra primera interpretación es una relación ternaria, trataremos de darle sentido a la relación a través de una nueva entidad relacionada con las entidades previas.

Por ejemplo: en una base de datos para una tienda, en la que necesitamos saber qué productos ha comprado cada cliente y qué trabajador los ha vendido, podemos plantear una relación terminaria entre comprador, vendedor y producto, obteniendo una relación ternaria. Sin embargo, también podemos crear una nueva entidad llamada venta con ciertas propiedades (fecha, precio, modo de pago) y relacionarla con las tres entidades previas:



Siempre intentaremos plantear relaciones binarias entre las entidades, para facilitar la transformación a modelo relacional.

Cardinalidad de una relación

Si consideramos que dos entidades A y B están relacionadas a través de una relación R, deberemos determinar lo que se conoce como *cardinalidad de la relación*, que determina cuantas entidades de tipo A se relacionan, como máximo, con cuantas entidades de tipo B. Además, resulta conveniente, en cada caso, calcular cuántas entidades de tipo A se relacionan, cómo mínimo, con cuantas entidades de tipo B (que normalmente será 0 ó 1). De esa manera podremos indicar la obligatoriedad, o no, de relación entre elementos de las entidades A y B.

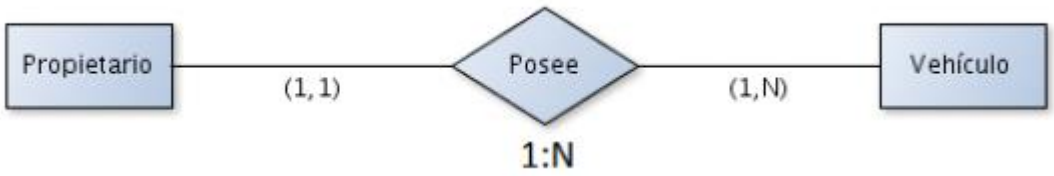
Cardinalidades parciales de una relación

Cardinalidades parciales de una relación

Son las cardinalidades que nos indican con cuantos elementos de una entidad participan los elementos de la otra entidad, cómo mínimo y como máximo. Cada entidad tiene su cardinalidad parcial, y a partir de ambas se obtiene la cardinalidad completa de la relación. Se separan mediante **,** y se encierran entre parentesis.

Cardinalidad total o completa de una relación

Es una cardinalidad que nos hace ver rápidamente la participación de dos entidades en una relación. La cardinalidad de la relación se indica dentro del símbolo de la relación o debajo de la relación. Se obtiene a partir de los máximos de las cardinalidades parciales de la relación:



Se indica en mayúsculas y separada por **:**

En los casos en los que el nombre de la relación entre dos entidades sea obvio, podemos omitir el nombre de la relación e indicar la cardinalidad completa o total de la relación en su interior.

Tipos de cardinalidad de una relación

Relación uno a uno

En esta relación una entidad de tipo A sólo se puede relacionar con una entidad de tipo B, y viceversa. Por ejemplo, si suponemos dos entidades *Curso* y *Aula*, relacionadas a través de una relación *Se Imparte*, podremos suponer que un *Curso* se imparte en una *Aula* y en una *Aula* sólo se

puede impartir un *Curso*. Se representaría como sigue:

Relación uno a muchos

Indica que una entidad de tipo A se puede relacionar con un número indeterminado de entidades de tipo B, pero a su vez una entidad de tipo B sólo puede relacionarse con una entidad de tipo A. Si suponemos una entidad *Propietario* y otra entidad *Vehículo* relacionadas a través de una relación *Posee*, podremos suponer que un *Propietario* puede poseer varios *Vehículos*, mientras que cada Vehículo sólo puede pertenecer a un *Propietario*.

Quedaría representado de la siguiente manera:

Relación muchos a uno

Significa que una entidad de tipo A sólo puede relacionarse con una entidad de tipo B, pero una entidad de tipo B puede relacionarse con un número indeterminado de entidades de tipo A. En realidad se trata como una relación uno a muchos pero el sentido de la relación es el inverso.

Relación muchos a muchos

En este caso, tanto las entidades de tipo A y B, pueden relacionarse con un número indeterminado de entidades del otro tipo. Por ejemplo, si suponemos las entidades *Alumno* y *Asignatura* y una relación *Cursa*, podremos suponer que un *Alumno* cursa varias asignaturas mientras que una *Asignatura* la cursan varios *Alumnos*. Quedaría representado de la siguiente manera:

Atributos de una relación

Existen situaciones en las que a la hora de modelar ciertos requisitos, nos encontramos con que hay ciertos datos que debe almacenar la base de datos que no son características **realmente propias** de ninguna de las entidades que conforman una relación. En esos caso es probable que dichas propiedades pertenezcan a la relación en sí.

Ejemplo: Un alumno puede cursar varias veces la misma asignatura por lo que debemos almacenar el año en que la cursa.

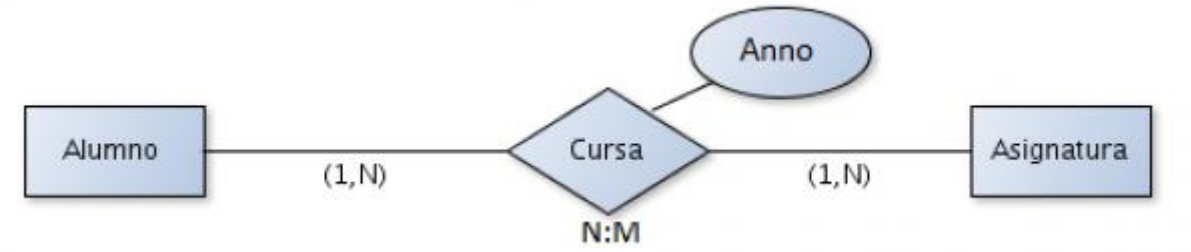
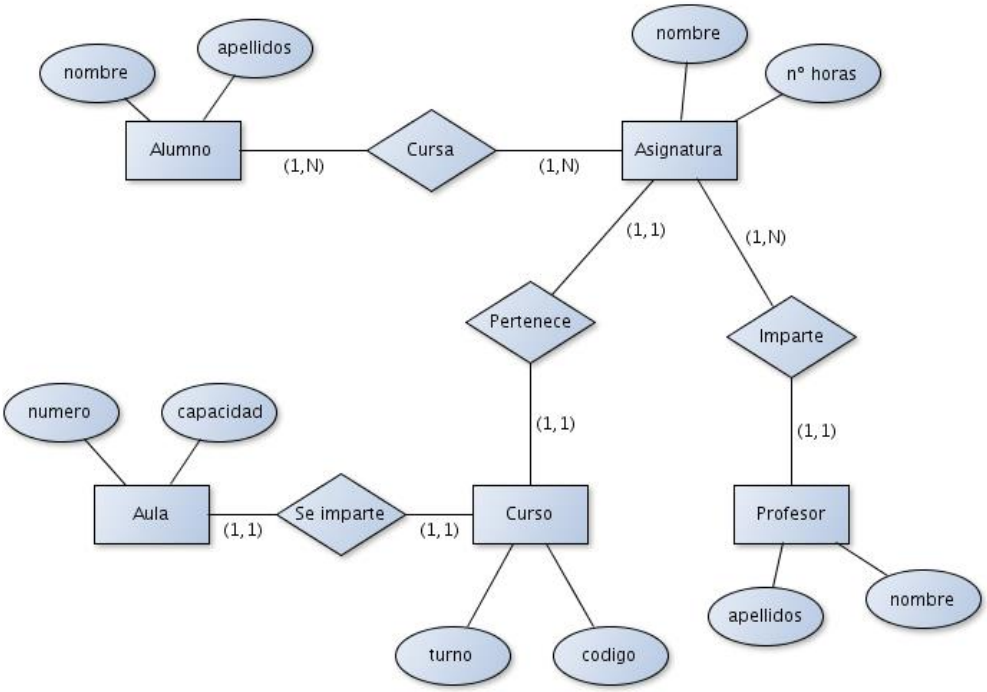


Diagrama Entidad/Relación

Se conoce como Diagrama Entidad/Relación (E/R) al diagrama resultante de modelar los requisitos de un caso concreto del mundo real siguiendo el modelo Entidad/Relación. Como resultado, se modelan todas las entidades con sus atributos, así como todas las relaciones existentes entre ellas, junto con su cardinalidad.

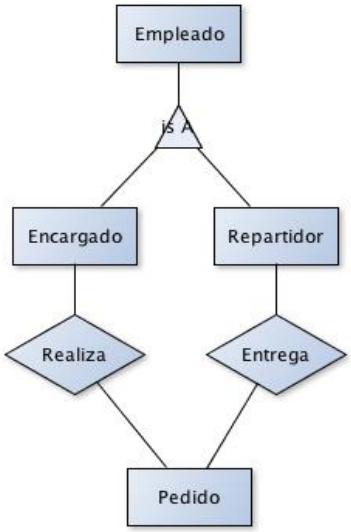


Otros tipos de relaciones

Hay algunos tipos concretos de relaciones que es interesante comentar:

Relación de Herencia

También conocida como **Generalización / Especialización**. La relación de herencia, representada como un triángulo, expresa que un objeto es un subtipo de otro objeto. También se suele considerar al subtipo como una especialización del primero o al primero como una generalización del segundo.



En el caso del ejemplo, existen dos tipos de empleados que se relacionan de forma diferente con otros objetos del sistema, pero que a su vez pueden tener gran parte en común. Por ejemplo, trabajan de forma diferente pero muchos de los datos personales que almacenaremos de ambos son comunes. Es por eso que el objeto *Empleado* se

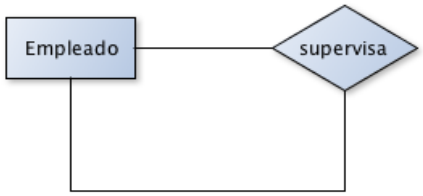
puede considerar una generalización de los dos tipos de trabajadores que hay en el sistema. Todos aquellos atributos y relaciones que tengan en común serán atributos y funcionalidades del objeto *Empleado* y los atributos y relaciones que tengan como trabajadores especializados(Encargados o repartidores) serán representados en la correspondiente entidad.

¿Cuándo nos interesa plantear una relación de herencia?

- Si las entidades padres e hijas tienen relaciones diferentes con otras entidades.
- Si las entidades hijas tienen atributos diferentes a la entidad padre.

Relaciones Reflexivas

Es posible que la misma entidad ocupe ambos lados de una relación. En ese caso estamos frente a lo que se conoce como relaciones reflexivas. La cardinalidad de la relación indicará si todos los elementos de la relación están relacionados reflexivamente o bien sólo algunos están relacionados entre sí. En el caso de la figura podríamos suponer una empresa en la que algunos empleados hacen de supervisor de otros empleados.



Las relaciones reflexivas indican todas sus cardinalidades del mismo modo que cualquier otra relación.

Otros tipos de Atributos

Ademas de los atributos simples que hemos visto en la primera parte, también existen otros tres tipos de atributos que podemos emplear:

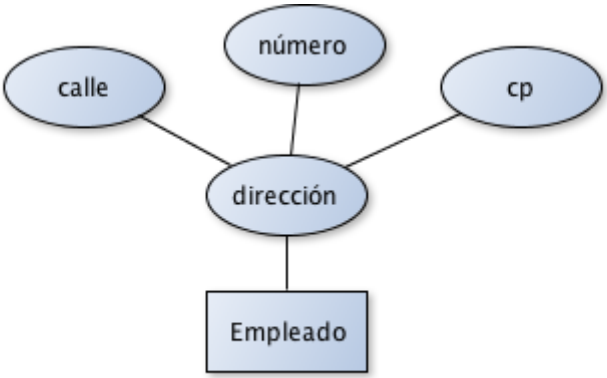
Atributos multivaluados

Los atributos multivaluados son aquellos atributos que pueden contener una cantidad indeterminada de valores. Por ejemplo, el teléfono de un empleado podría tomar varios valores para alguien que posea varios teléfonos.

Se representan del siguiente modo:

Atributos estructurados o compuestos

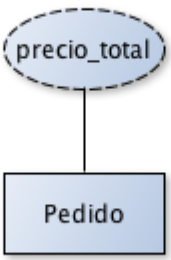
Son atributos que pueden ser divididos en subpartes; éstas constituirán otros atributos con significado propio. Por ejemplo, la dirección del empleado podría considerarse como un atributo compuesto por la calle, el número y la localidad.



Normalmente son atributos que pueden descomponerse aunque dependiendo del contexto de la aplicación puede no interesar hacer esa descomposición y tratarlo como un atributo simple.

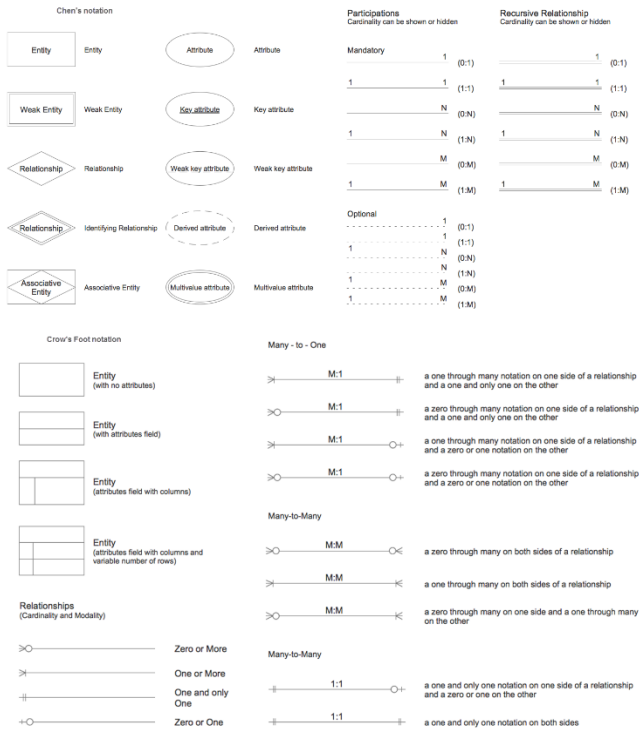
Atributos derivados o calculados

Los atributos derivados (o calculados) son aquellos atributos cuyo valor puede ser deducido realizando algunas operaciones con otros atributos de la misma entidad o de otras entidades. En algunas situaciones se podría considerar redundante (puesto que su valor se puede deducir) pero en otras puede resultar cómodo almacenarlo ya calculado puesto que se puede ahorrar mucho tiempo de cómputo si se trata de un valor de díficil y/o recurrente cálculo.



Otras simbologías del modelo E/R

El modelo Entidad Relación es el lenguaje más común para hacer diseños comceptuales de bases de datos. Es un modelo creado por Peter Chen en el año 1976, y ha pasado por diferentes ampliaciones, y han contribuido diferentes personalidades. Es por eso, que además de los elementos vistos en esta parte del tema, podemos encontrar otros elementos u otras formas de representación.



La elección de los elementos expuestos en este bloque se basa en dos cuestiones:

- Elementos del lenguaje fáciles de comprender y aplicar en el diseño de una base de datos.
- Conjunto de elementos suficiente y necesario para plantear cualquier situación de diseño de una bbdd.

Comprobaciones sobre el Diagrama Entidad-Relación

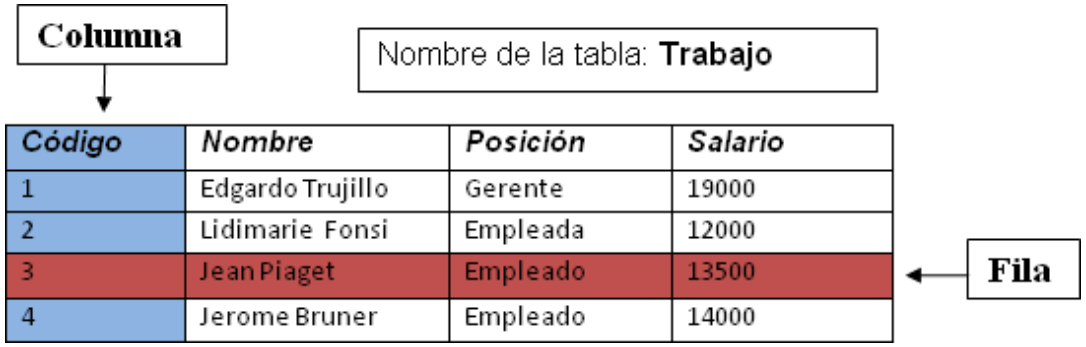
1. Resulta cómodo que las entidades estén escritas en minúscula para hacer todas estas comprobaciones
2. Comprobar que nuestro diagrama no se ha convertido en un diagrama de flujo y no describe procesos, sino almacenes de datos
3. Comprobar que las Entidades son nombres de cosas y las relaciones son verbos. En los casos en que las relaciones sean muy obvias, se puede omitir el nombre.
4. Comprobar que ninguna Entidad tiene como atributo algo que existe como Entidad (si ocurre, se deberían relacionar ambas Entidades)
5. Comprobar que varias entidades no comparten un mismo atributo estructurado que pueda ser considerado realmente como una Entidad
6. Evitar los ciclos (si aparece alguno, que puede ocurrir, comprobar que es necesario)

7. Si una relación tiene varios atributos, valorar si es posible que realmente deba ser una nueva Entidad (Comprar -> Pedido, Alquilar -> Alquiler, Reservar -> Reserva, Enviar -> Envío, . . .)
8. Comprobar que no hay colocada ninguna cardinalidad al revés: Se tiene que poder leer: un **Usuario Realiza** de 0 a N **Pedidos**. **Usuario** y **Pedido** son entidades y **Realizar** la relación entre ambas. En este caso, (0, N) debería estar escrito en el lado **Pedido** para que pudiera leerse correctamente

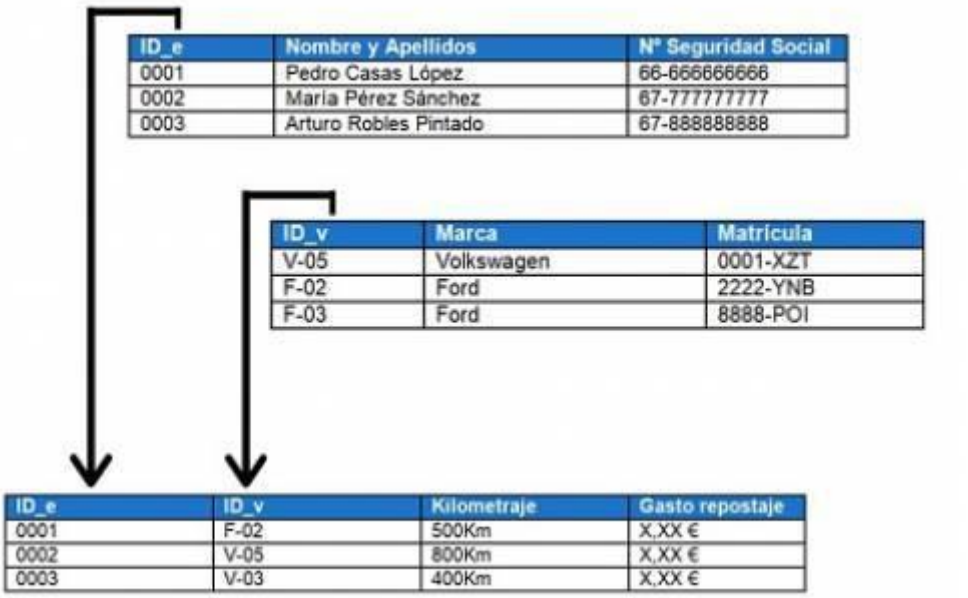
Modelo relacional

El modelo relacional es otro modelo de representación en el que los datos y sus relaciones se representan a través de tablas, y en el que los atributos se traducen en campos de esas tablas. Es el modelo de representación que siguen la gran mayoría de los SGBD relacionales (MySQL, SQL Server, Oracle, Ms Access, entre otros) en la actualidad, puesto que es el modelo de datos más extendido.

Así, es necesario transformar nuestro modelo Entidad/Relación a un modelo relacional si queremos crear nuestra Base de Datos en algún SGBD relacional.



Código	Nombre	Posición	Salario
1	Edgardo Trujillo	Gerente	19000
2	Lidimarie Fonsi	Empleada	12000
3	Jean Piaget	Empleado	13500
4	Jerome Bruner	Empleado	14000



ID_e	Nombre y Apellidos	Nº Seguridad Social
0001	Pedro Casas López	66-666666666
0002	María Pérez Sánchez	67-777777777
0003	Arturo Robles Pintado	67-888888888

ID_v	Marca	Matrícula
V-05	Volkswagen	0001-XZT
F-02	Ford	2222-YNB
F-03	Ford	8888-POI

ID_e	ID_v	Kilometraje	Gasto repostaje
0001	F-02	500Km	X,XX €
0002	V-05	800Km	X,XX €
0003	V-03	400Km	X,XX €

En la práctica no es necesario dibujar una tabla completa, sino que se suele indicar el nombre de los campos, y los campos claves que conforman las relaciones:

```
Trabajo (#codigo, nombre, posicion, salario)

Empleado(#dni, nombre, apellidos, -codigo_trabajo)



Profesor (#id, dni, nombre, apellidos, departamento)
Asignatura (#id, nombre, n_horas, -id_profesor)
```

Clave Primaria y Clave Ajena

Cuando tenemos un diseño lógico organizado en tablas y campos, las relaciones se realizan estableciendo referencias entre columnas de dos tablas. Los sistemas SGBD Relacionales hacen hincapie en el concepto de **Integridad Referencial** de los datos almacenados.

La *Integridad Referencial* consiste en mantener las relaciones entre los registros de diferentes tablas de forma consistente e inequívoca, de forma que siempre podamos conocer qué datos de una tabla están relacionados con otro dato de otra tabla.

Para mantener esta integridad de los datos es necesario es uso de dos tipos de campos clave dentro de las tablas: *Claves primarias y Claves ajenas o foráneas*.

Antes de explicar estos dos tipos de claves vamos a ver los distintos tipos de claves:

Superclave

Una superclave es una combinación de atributos o columnas cuya combinación de valores permite identificar inequívocamente a cada fila o registro de la tabla.

Clave candidata

Una clave candidata es una superclave que no se puede reducir. Cada clave candidata puede ser un solo atributo o una combinación de varios atributos. Una tabla puede tener varias claves candidatas.

Clave Primaria

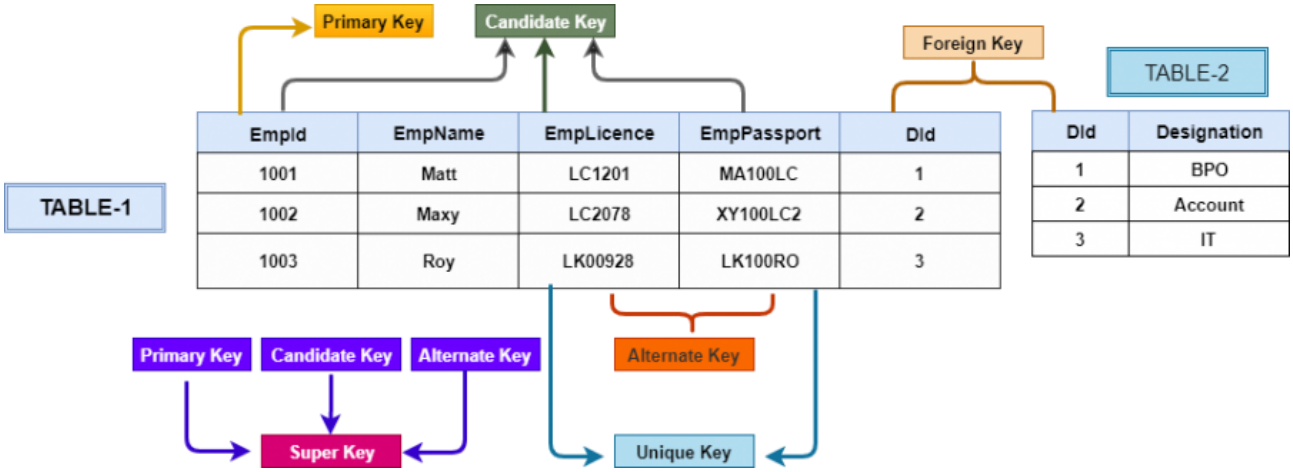
Denominamos *Clave Primaria*, en inglés Primary Key (PK), al campo identificador principal de una tabla. Es la clave candidata elegida por el usuario para identificar a cada registro de la tabla.

La elección del campo que hará de clave primaria corre a nuestra cuenta, pero es habitual crear un campo *id* autonumérico para tal fin por motivos de simplicidad.

```
-- Representamos las claves primarias con la almohadilla (#)

Trabajo (#codigo, nombre, posicion, salario)
Profesor (#id, dni, nombre, apellidos, departamento)
Asignatura (#id, nombre, n_horas, -id_profesor)
```

Claves Alternativas o Secundarias

Se suele llamar de este modo a todas las *claves candidatas* que no se han designado como clave primaria.



Clave Ajena

Toda tabla cuyos registros tengan relación con los registros de otra tabla, debe tener un campo cuyos valores identifiquen a los registros de otra tabla. A este campo se le conoce como *Clave Ajena o Foránea*, en inglés Foreign Key (FK). Contiene los valores de la clave primaria de los registros de la tabla con la que se relaciona.

Aparte, es habitual que esta tabla también tengo un campo Clave Primaria.

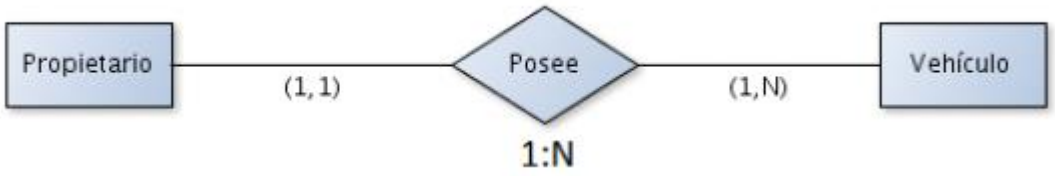
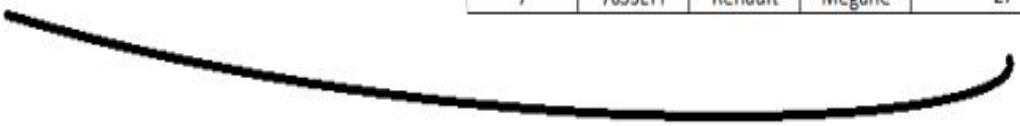


Tabla Propietario

id (PK)	nombre	apellidos	...
13	Fernando	Valdeón	
15	Ana	López	
27	Javier	Alonso	

Tabla Vehiculo

id (PK)	matricula	marca	modelo	id_propietario (FK)
2	5422AWB	Seat	León	13
3	7543KGT	Ford	Focus	15
5	4764AAB	Citroen	C3	13
7	7653LTY	Renault	Megane	27



La tabla *Propietario* tiene un campo (id) que actúa como *clave primaria*(PK) de la tabla. No puede haber dos registros con el mismo valor (id). La tabla *Vehiculo* tiene un campo(id) que actúa como *clave primaria* de la tabla. Además la tabla *Vehículo* tiene un campo (id_propietario) que actúa como *clave ajena*(FK), y referencia cada uno de sus registros con un registro concreto de la tabla Propietario.

De este modo podemos saber quién es el propietario de cada vehículo. Como se trata de una relación 1:N un propietario puede tener varios Vehículos, pero cada vehículo solo puede tener un propietario.

```
-- Representamos las claves ajenas mediante un guión (-)
```

```
-- Además indicamos añadimos a la columna el nombre de la tabla de la que procede.
Profesor (#id, dni, nombre, apellidos, departamento)

-- Una asignatura es impartida por un profesor
Asignatura (#id, nombre, n_horas, -id_profesor)

-- Vehiculos tiene una clave ajena que referencia a peopietarios
Propietarios (#id, nombre, apellidos, . . .)
Vehiculos (#id, matricula, marca, modelo, -id_propietario)
```

Elección de Claves Primarias

A la hora de seleccionar la columna que ejercerá como *clave primaria* podemos utilizar dos enfoques:

- **Clave Primaria Natural**: Son claves primarias que ya existen en nuestra tabla y tienen un significado real; representan datos reales. Ejemplos de claves primarias naturales son: dni, nss, matricula, expediente, o combinaciones de columnas que tengamos en nuestra tabla.
- **Clave Primaria Subrogada**: Son columnas *clave primaria* creadas por el programador exclusivamente para el diseño de la base de datos, sin que sean campos existentes en los requisitos de la base de datos. Consiste en añadir una columna autonumérica cuyos valores son generados por el motor de la base de datos de forma automática siendo siempre únicos. El ejemplo que nosotros usamos en estos apuntes es crear una columna **id** autonumérica.

Ventajas e inconvenientes:

- Para poder designar claves primarias naturales necesito que mi tabla tenga campos que sean claves candidatas. Esta situación no siempre se da, o requiere de la unión de varias columnas.
- A la hora de relacionar tablas, simplifica mucho la tarea que las claves ajenas sean lo más sencillas posibles; mejor si la clave ajena es un solo campo de la tabla a que sea una clave ajena compuesta. Para ello necesito que la clave primaria sea simple, y no se componga de varios campos.
- Las claves primarias subrogadas se pueden añadir a cualquier tabla aunque no tenga claves candidatas, tanto en el momento del diseño, como en fases posteriores que requieren modificar la estructura de la bbdd.

Existen numerosos debates sobre las ventajas e inconvenientes de cada tipo de enfoque.

En los ejemplos de esta guía se opta por uso de **claves primarias subrogadas** ya que permite plantear un enfoque más sencillo y escalable. Eso no quiere decir que el diseño sea mejor o peor que utilizando *claves naturales*.

Transformación del modelo E/R al modelo relacional

El paso de un modelo E/R a un modelo relacional se puede llevar a cabo, en gran parte, siguiendo una serie de reglas o pautas, que se enumeran a continuación:

Entidades y atributos:

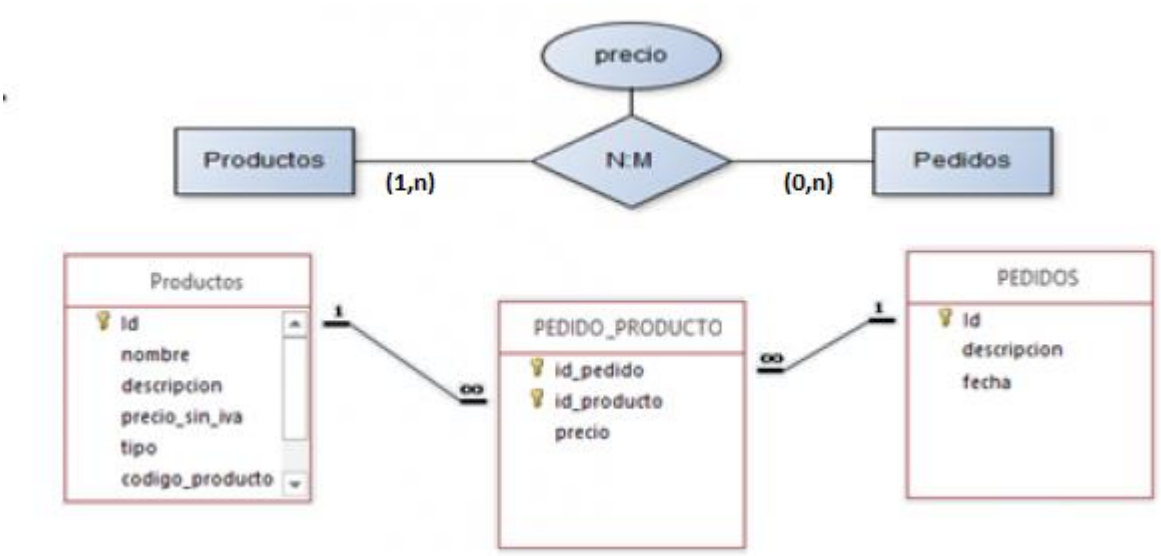
- Toda **entidad** se transforma en una tabla.

- Todo **atributo simple** o **derivado** se transforma en columna de una tabla.
- Los **atributos estructurados** desaparecen de la tabla y solamente setransforman los campos de los que se componen en nuevas columnas.
- El **identificador único** de la entidad se convierte en clave primaria. Una posibilidad es utilizer una clave primaria subrogada, añadiendo un campo *id* como clave primaria.
  Los **atributos multivaluados** hacen desaparecer el atributo de la entidad origen generando una nueva tabla con tres columnas: un *id*, el *id* de la tabla de la que surgen propagado como *clave ajena* y el valor del campo multivaluado:
  (#id, -id_tabla_origen, atributo_multivaluado)

Relaciones:

- Toda **relación N:M** se transforma en una tabla que tendrá como clave primaria la concatenación de los atributos clave de las entidades que relaciona. Cada uno de los dos campos será además clave ajena.

Ejemplo: Un producto puede estar en varios pedidos, y un pedido puede tener varios productos.

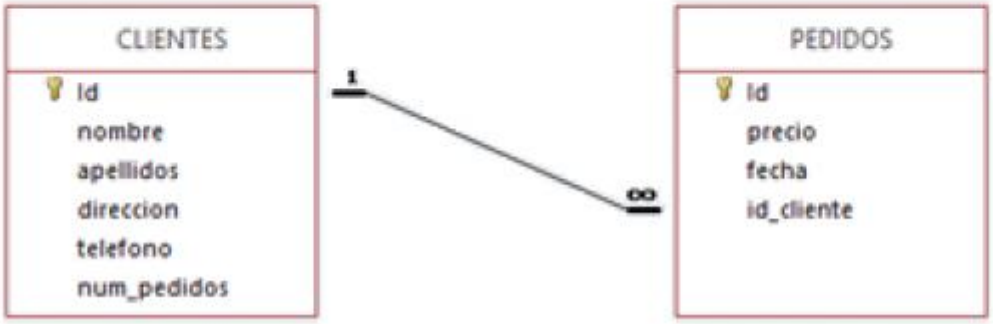


Tablas pedidos, productos y pedido_producto

pedidos(#id, descripcion, fecha)
productos(#id, nombre, descripcion, precio_sin_iva, tipo, codigo_producto)
pedido_producto(#(-id_pedido, -id_producto), precio)

- En la transformación de **relaciones 1:N** se propaga el atributo clave (habitualmente el campo id) de la entidad que tiene de cardinalidad máxima 1 a la que tiene cardinalidad máxima N, haciendo desaparecer a la relación.

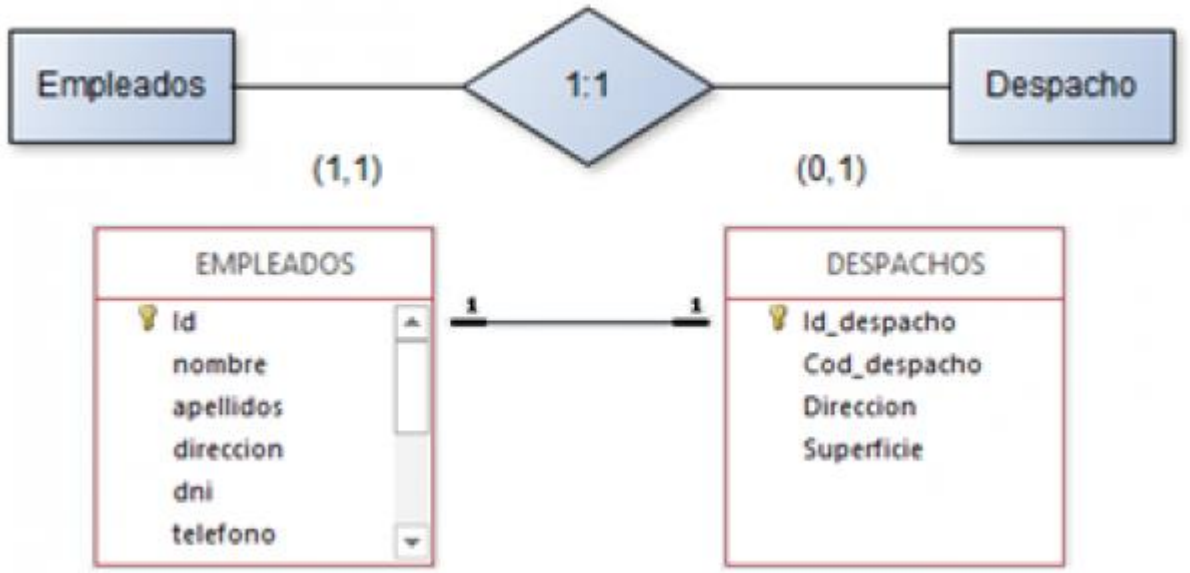
Ejemplo: Un pedido es realizado por un solo cliente, y un cliente puede realizar multiples pedidos.



pedidos(#id, precio, fecha, -id_cliente)
clientes(#id, nombre, apellidos, direccion, telefono, num_pedidos)

- En la transformación de **relaciones 1:1** se tienen en cuenta las cardinalidades de las entidades que participan en ellas. Existen diferentes soluciones:
    1. La transfomación tradicional consiste en que una de las tablas tenga una clave primaria que al mismo tiempo es clave ajena de la otra tabla. Solo se puede plantear si alguna de las cardinalidades parciales es (1,1). En ese caso se propaga la clave primaria de la entidad con cardinalidad (1,1) a la tabla resultante de la entidad de cardinalidad (0,1), en la que será clave primaria y ajena al mismo tiempo. Si ambas cardinalidades parciales son (1,1), no importa hacia qué lado se propaga. (Opción 1)
    2. Otra solución más sencilla es transformarla como si fuera una relación 1:N. Si existe una entidad con cardinalidad parcial (0,1), su tabla tendrá una columna nueva que sería la clave primaria de la otra entidad propagada como clave ajena. Si las cardinalidades parciales son iguales da igual hacia qué tabla se propague la clave primaria. *Si usamos esta opción, al crear las tablas en el SGBD, la columna clave ajena debe tener restricción* UNIQUE. (Opción 2)

Ejemplo: Un empleado solo puede tener un despacho, y un despacho pertenece a un solo empleado.

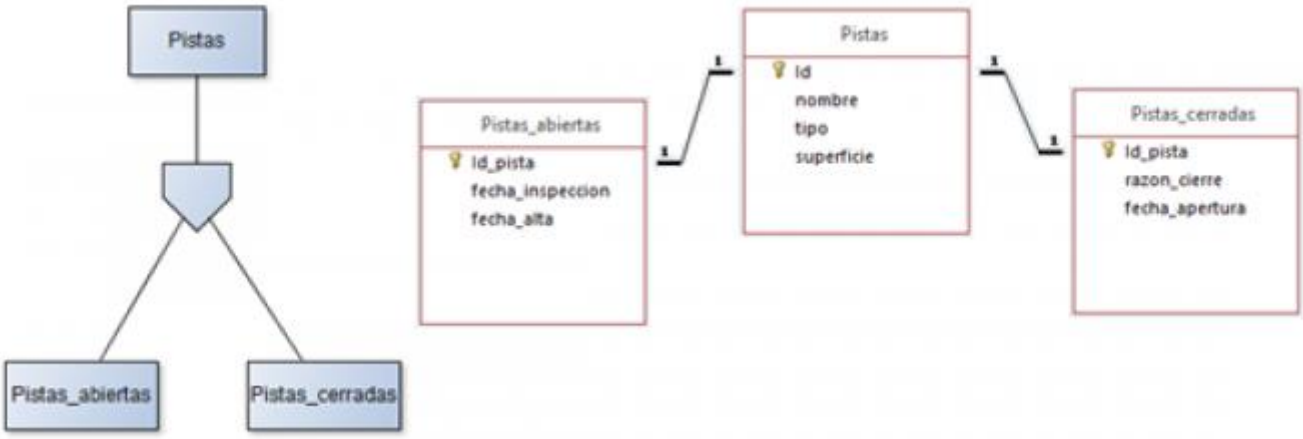


Tablas empleados y despachos (Opción 1)

```
-- Opción 1: Clave primaria y ajena al mismo tiempo

empleados(#id, nombre, apellidos, direccion, dni, telefono)
despachos(#(-id_despacho), cod_despacho, direccion, superficie)

-- Opción 2: Clave primaria y ajena diferente (como 1:N)

empleados(#id, nombre, apellidos, direccion, dni, telefono)
despachos(#id, cod_despacho, direccion, superficie, -id_empleado)
```

- Las relaciones de **herencia** se transforman como relaciones 1:1 entre tablas padres e hijas. Aplicando las reglas anteriores, se ha de crear una tabla por cada entidad hija con sus propias columnas.
    1. Para relacionarlas, la forma más directa es propagar la clave primaria de la tabla padre como claves ajenas en las tablas hijas; Las tablas hijas tienen como clave primaria el campo clave de la tabla padre (Ejemplo 1).
    2. La otra forma, menos común, es transfomarla como si fueran dos relaciones **1:N**: las tablas hijas tienen su propia clave primaria (id), y añadirán una columna nueva que es la clave primaria de la entidad padre, propagada como clave ajena. En este caso, al crear las tablas hijas en el SGBD, las claves ajenas deben tener las restricciones UNIQUE y NOT NULL) (Ejemplo 2)

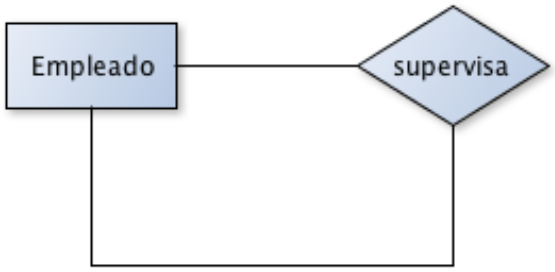
Ejemplo: Una pista puede ser una pista_abierta o una pista_cerrada.



Tablas pistas, pistas_abiertas y pistas_cerradas

```
-- Ejemplo 1: Claves primarias y ajenas al mismo tiempo

pistas(#id, nombre, tipo, superficie)
pistas_abiertas(#(-id_pista), fecha_inscripcion, fecha_alta)
pistas_cerradas(#(-id_pista), razon_cierre, fecha_apertura)

-- Ejemplo 2: Claves primarias y ajenas diferentes (-id_pista es UNIQUE y NOT
NULL)

pistas(#id, nombre, tipo, superficie)
pistas_abiertas(#id, fecha_inscripcion, fecha_alta, -id_pista)
pistas_cerradas(#id, razon_cierre, fecha_apertura, -id_pista)
```

- Las relaciones **reflexivas**, se transforman antendiendo a su cardinalidad del mismo modo que hemos planteado en los casos anteriores:
    - Si la cardinalidad es **N:M**, se crea una tabla nueva como el caso general de relación N:M. La clave primaria se compone de dos columnas que referencian al mismo id de la propia tabla.
    - Si la cardinalidad es **1:N**, se transforma como si se tratara de una relación 1:N, se propaga la clave primaria a la misma tabla como clave ajena.
    - Si la cardinalidad es **1:1**, al tratarse de la misma tabla, se transforma como si se tratara de una relación 1:N. *Debemos indicar en el SGBD que la columna clave ajena no permitirá valores repetidos (restricción* UNIQUE*).*



> Un empleado puede supervisar a varios empleados

```
-- Relacion 1:N y 1:1
empleados(#id, nombre, apellidos, dni, -id_empleado_supervisor)

-- Relación N:M
empleados(#id, nombre, apellidos, dni)
empleados_supervisores(#(-id_empleado_supervisor, -id_empleado_supervisado))
```

- Para los **atributos de las relaciones** existen dos casos:
    - Si la relación es **1:N**, sus atributos se propagan a la tabla de lado N, junto con la clave del lado 1
    - Si la relación es **N:M**, sus atributos se transforman en columnas de la tabla generada por dicha relación
    - Las relaciones **1:1** no deberían de tener atributos, ya que estos deben pertenecer a alguna de las dos entidades.

Las situaciones más particulares habrá que estudiarlas y aplicar algún *patrón de diseño* conocido, si lo hay, para generar el correspondiente modelo relacional. Estos casos no siempre se podrán reflejar en el correspondiente modelo Entidad-Relación puesto que algunos tienen que ver con exigencias técnicas o de tiempo, más que con el propio modelo de datos. Algunos casos particulares pueden ser:

- **Datos temporales**: Cómo representar en un modelo relacional datos que tienen *fecha de caducidad* (los precios de un producto del que queremos tener un histórico de precios, . . .)
- **Datos eliminados**: Cómo representar en un modelo relacional datos que se desean eliminar, pero que por alguna razón necesitamos que sigan estando almacenados en la base de datos (productos descatalogados, alumnos que terminan sus estudios, . . .)
- **Registro o auditoría**: Cómo podemos registrar las acciones de los usuarios o de la

aplicación durante el ciclo de vida de la Base de Datos.

- **Bloqueo de registros**: Cómo podemos bloquear el acceso a un registro para evitar la modificación simultánea del mismo dato por más de un usuario desde la aplicación que conecta con la Base de Datos.

En cualquier caso, aplicar correctamente al modelo relacional resultante las reglas de normalización eliminará todas las anomalías que nuestro modelo pueda contener. Así, hay que tener en cuenta que el modelo relacional que hemos obtenido en este momento todavía puede no ser el definitivo y puede sufrir transformaciones (e incluso se pueden añadir nuevas tablas) como resultado de aplicar las reglas de normalización que se pasan a explicar en el siguiente punto.

Normalización de modelos relacionales

Uno de los retos en el diseño de toda Base de Datos es el obtener una estructura estable tal que:

- El sistema no sufra de anomalías de almacenamiento
- El modelo lógico pueda modificarse si aparecen nuevos requisitos

Una Base de Datos bien diseñada tiene mayor esperanza de vida, incluso en un ambiente dinámico donde puedan aparecer nuevos requisitos, que una Base de Datos con un diseño pobre. Como media, una Base de Datos puede sufrir una reorganización cada seis años, dependiendo de lo dinámico que sean sus requisitos. Si la Base de Datos se diseño bien seguirán teniendo un buen rendimiento aunque aumente el tamaño, y será lo suficientemente flexible para soportar los nuevos requisitos y/o características adicionales.

Actualmente existen diversos riegos en el diseño de Bases de Datos relacionales. Los más habituales son la redundancia de información, la inconsistencia de datos y el no aprovechamiento de espacio en disco.

La normalización es el proceso de simplificar la relación entre los campos de un registro de forma que éste se reemplaza por varios registros más simples y predecibles y, por tanto, más manejables. En definitiva, la normalización busca simplificar el diseño para que éste sea más fácil incorporar nuevas funcionalidades con el paso del tiempo y no baje su rendimiento cuando la cantidad de datos almacenados en ella aumenten considerablemente.

La teoría de la normalización se basa en lo que se conoce como *Formas Normales*. Cada una de estas Formas Normales establece una serie de restricciones que el diseño deberá cumplir para satisfacer dicha Forma. Se considera que una base de datos que cumple las tres primeras FN (Formas Normales) tiene un nivel suficiente de normalización.

Primera forma normal 1FN

Se dice que una tabla está en *primera forma normal* si una tabla posee las siguientes propiedades:

- Cada columna tiene un solo valor y un solo tipo de datos
- El orden de las filas y las columnas no importa
- Dos filas no contienen valores idénticos
- Las columnas no pueden contener valores repetidos o que representan lo mismo

Supongamos el caso más común, que un campo pueda tener más de un valor:

id	nombre	apellidos	telefono

123	Alfonso	Garcia	123-345-456
456	Sara	Casas	555-666-777, 555-234,876, 234-654-345
789	Lorena	González	555-666-777

Es similar al caso de tablas que poseen columnas con valores repetidos (una tabla de clientes con los campos teléfono 1, teléfono 2, teléfono 3, . . .):

id	**nombre**	**apellidos**	**telefono1**	**telefono2**	**telefono3**
123	Alfonso	Garcia	123-345-456		
456	Sara	Casas	555-666-777	555-234-876	234-654-345
789	Lorena	González	555-666-777		

En el caso anterior, hay columnas con valores nulos (vacias) y si hay una gran cantidad de registros la definición de la tabla ocupará mucho espacio en desuso. Del mismo modo tendremos problemas a la hora de realizar consultas del tipo: *¿Cual es el teléfono2 de los clientes?*, *¿Cuantos números de teléfono tiene el usuario 123?*, ó *¿Que usuarios tienen el mismo número de teléfono?*.

De cualquier modo estás tablas no cumplen la **1FN**. Para normalizar debemos crear una tabla nueva para los teléfonos:

- Se localizan los atributos correspondientes a la clave principal.
- Cada una de las columnas o valores repetidos se separan en una nueva tabla, de manera que se hace la proyección de la clave primaria de la tabla de la que proceden sobre cada uno de los valores del atributo que no es atómico.

tabla clientes		
id	**nombre**	**apellidos**
123	Alfonso	Garcia
456	Sara	Casas
789	Lorena	González

tabla teléfonos		
id	**numero**	**id_cliente**
1	123-345-456	123
2	555-666-777	456
3	555-234-876	456
4	234-654-345	456
5	555-666-777	789

Segunda forma normal 2FN

Esta formal normal sólo debe ser considerada para aquellas tablas en las que la clave principal sea compuesta. Si no fuera así, la tabla estaría, de forma directa, en segunda forma normal.

Decimos que una tabla está en *segunda forma normal* si se cumplen las siguientes condiciones:

- Está en 1FN

- Todo atributo secundario (que no pertenezca a la clave principal) tiene una dependencia funcional total de la clave principal, y no de una parte de ella

*Se dice un atributo B depende funcionalmente de A (*A->B*) si cada valor de A se corresponde con un único valor de B. Visto de otra manera, si dado A puedo obtener B. Un caso típico podría ser* DNI -> Nombre*, puesto que dado un DNI puedo obtener, de forma unívoca, el nombre de la persona*

Para convertir una tabla que no está en 2FN se creará una tabla con la clave y todas sus dependencias funcionales totales y otra tabla con la parte de la clave que tiene dependencias con los atributos secundarios.

En el ejemplo podemos ver como el campo *TelefonoProveedor* no depende totalmente de la clave *(NombreProducto, NombreProveedor)*, sino únicamente del campo *NombreProveedor*.

NombreProducto	NombreProveedor	Categoria	TeléfonoProveedor
Diarios	Exotic Kiosk	Prensa	968582222
Revistas	Exotic Kiosk	Prensa	968582222
Habas	Tnj export	Alimentacion	975869999

Clave primaria (NombreProducto, NombreProveedor). Existen dependencias funcionales

IdProducto	NombreProduct	Categoria	IdProveedor
1	Diarios	Prensa	1
2	Revistas	Prensa	1

Elimino los atributos que depende de una parte de la clave primaria

IdProveedor	NombreProveedor	TeléfonoProveedor
1	Exotic Kiosk	968582222
2	Tnj Export	975869999

Aplicando la 2ª FN. Tabla Proveedores

Los junto en una tabla propia

Tercera forma normal 3FN

Se dice que una tabla está en *tercera forma normal* si:

- Está en 2FN
- No existen atributos no primarios (que no pertenezcan a la clave) que son transitivamente dependientes de cada posible clave de la tabla. Es decir, un atributo secundario sólo puede ser conocido a través de la clave principal y no por medio de un atributo no primario.

Para convertir una tabla que no está en 3FN se realizará una proyección de la clave a los elementos que no tengan dependencia funcional transitiva y otra tabla con una nueva clave a los elementos que anteriormente tenían esta dependencia.

En el ejemplo, es posible conocer la *edad* del inscrito a través del *número de licencia*, y dada la *edad* podemos conocer su *categoría*, tenemos una dependencia funcional transitiva entre

categoría y *número de licencia*. Lo importante es reconocer que la *categoría* depende de un atributo que no forma parte de la clave. Para normalizar, debemos descomponer esa tabla en las tablas *Atletas* y *Categorías*

Licencia	Nombre	Club	Edad	Categoria
189585	Pepe	Alcoy	12	Junior
205888	Tomas	Jaca	10	Cadete
748523	Andres	Almansa	9	Alevín

> Tabla con dependencias transitivas (categoría -> edad)

Licencia	Nombre	Club	Edad
189585	Pepe	Alcoy	12
205888	Tomas	Jaca	10
748523	Andres	Almansa	9

> Tabla Atletas

Edad	Categoria
9	Alevín
10	Cadete

> Tabla categorías

Creación de BBDD en lenguaje SQL (MySQL)

El lenguaje SQL (Structured Query Language) permite la comunicación con el SGBD. Actualmente es el lenguaje estándar para la gestión de Bases de Datos relacionales para ANSI (American National Standard Institute) e ISO (International Standarization Organization). Entre las principales características de este lenguaje destaca que es un lenguaje para todo tipo de usuarios ya que quién lo utiliza especifica qué quiere, pero no dónde ni cómo, de manera que permite realizar cualquier consulta de datos por muy complicada que parezca.

En el siguiente enlace hay un video tutorial sobre cómo configurar MySql y MySql Workbench y exportar scripts: https://vimeo.com/764773231

Definición de una Base de Datos (Sentencias DDL)

En el lenguaje SQL, dependiendo de las tareas que se quieran realizar, se distinguen dos tipos de sentencias. Sentencias DDL (Data Definition Language) que sirven para especificar y gestionar estructuras de datos, y las sentencias DML (Data Manipulation Language) que sirven para trabajar con los datos almacenados en dichas estructuras.

Puesto que por ahora abordaremos aquellas sentencias que nos van a permitir crear nuestras Bases de Datos en un SGBD relacional, comenzaremos por ver el grupo de sentencias DDL, que son las que se citan a continuación:

Crear un objeto: CREATE

Es la sentencia utilizada para la creación de un objeto (base de datos, tabla, usuario, vista, procedimiento, . . .) en una Base de Datos.

Para crear una Base de Datos:

```
CREATE DATABASE <nombre_base_de_datos>
```

Según los estándares, **una base de datos es un conjunto de objetos que nos servirán para gestionar los datos**. Estos objetos están contenidos en esquemas y éstos a su vez suelen estar asociados a un usuario. De ahí que antes dijéramos que cada base de datos tiene un esquema que está asociado a un usuario.

Para crear una tabla:

```
CREATE TABLE <nombre_tabla>
(
  <nombre_columna1> <tipo_dato> <restricciones>,
  <nombre_columna2> <tipo_dato> <restricciones>,
  ..............................
)
```

Conectar con una Base de Datos

Permite realizar la conexión con una Base de Datos de MySQL. Es necesario conectarse a una base de datos, para crear la estructura de esa base de datos.

```
  USE <nombre_base_de_datos>
```

Ejemplo completo:

```
-- Crear una base de datos
CREATE DATABASE colegio;

-- Conectarse a la base de datos
USE colegio;

-- Crear una tabla
CREATE TABLE asignaturas(
  id INT PRIMARY KEY,
  nombre VARCHAR(20),
  departamento VARCHAR(20),
  id_curso INT,
  FOREIGN KEY (id_curso) REFERENCES cursos(id)
);
```

Eliminar un objeto: DROP

Es la sentencia utilizada para eliminar objetos (tabla, usuario, vista, procedimiento, . . .) en una Base de Datos.

La sintaxis para la eliminación de tablas es la siguiente:

```
DROP TABLE <nombre_tabla>
```

Y para eliminar una Base de Datos:

```
DROP DATABASE [IF EXISTS] <nombre_base_de_datos>
```

Modificar un objeto: ALTER

Es la sentencia utilizada para modificar objetos (tabla, usuario, vista, procedimiento, . . .) en una Base de Datos.

La sintaxis para modificar una tabla es la siguiente:

```
ALTER TABLE <nombre_tabla>
  [ ADD <definicion_columna> ]
  [ MODIFY <nombre_columna> <definicion_columna> ]
  [ DROP COLUMN <nombre_columna> ]
  [ ADD CONSTRAINT <restriccion> ]
  [ CHANGE <nombre_columna> <nuevo_nombre> <definicion_nueva_columna>]
  [ AUTO_INCREMENT = <valor> ]
```

Ejemplos:

```
-- Añadir o eliminar columna
ALTER TABLE alumnos ADD COLUMN edad INT DEFAULT 18;
ALTER TABLE alumnos DROP COLUMN edad;

-- Modifica la definición de la columna
ALTER TABLE alumnos MODIFY nombre VARCHAR(30) NOT NULL;
-- Modifica el nombre y la definición de la columna
ALTER TABLE alumnos CHANGE nombre nick VARCHAR(30) NOT NULL;

-- Añadir o quitar claves primarias o ajenas
ALTER TABLE alumnos ADD PRIMARY KEY(id);
ALTER TABLE alumnos DROP PRIMARY KEY;

ALTER TABLE alumnos ADD FOREIGN KEY (id_curso) REFERENCES cursos(id);
-- Igual a la anterior pero indicando el nombre de la restricción
ALTER TABLE alumnos ADD CONSTRAINT nombre_fk_1 FOREIGN KEY (id_curso) REFERENCES
cursos(id);
ALTER TABLE alumnos DROP FOREIGN KEY nombre_fk;

-- Renombar tabla
ALTER TABLE alumnos RENAME estudiantes;
```

En la documentación oficial tenemos más ejemplos de uso.

Tipos de datos

Cadenas de caracteres

Tipo CHAR, VARCHAR

Este tipo de datos permite almacenar cadenas de texto fijas (CHAR) o variables (VARCHAR).

El tipo CHAR permite almacenar cadenas de caracteres de longitud fija entre 1 y 255 caracteres. La longitud de la cadena se debe especificar entre paréntesis en el momento de la declaración

(cadena CHAR(25)).

Por otro lado, el tipo VARCHAR permite almacenar cadenas de caracteres variables. La declaración del tipo VARCHAR es similar a la de un tipo CHAR (cadena VARCHAR(25)). La principal y única diferencia entre estos dos tipos, es que el tipo CHAR declara una cadena fija de la longitud que se especifica mientras que en la declaración de un tipo VARCHAR lo que se especifica es un tamaño máximo, la cadena sólo ocupará el tamaño necesario para almacenar el dato que contenga (hasta llegar al máximo). En cualquier caso, no es posible almacenar cadenas de mayor tamaño al especificado en su declaración, puesto que el SGBD truncará el valor almacenándose sólo hasta la longitud establecida.

Tipo TEXT

El tipo TEXT permite almacenar cadenas de caracteres de hasta varios GB de longitud. Sólo se recomienda su uso para almacenar textos realmente grandes, puesto que presenta ciertas restricciones, aunque algunas pueden variar dependiendo del SGBD que se utiliza:

- Sólo se puede definir una columna TEXT por tabla
- No se pueden establecer restricciones en columnas de este tipo
- No se permite su utilización en ciertas cláusulas

Tipos numéricos

Para la representación de tipos de datos numéricos. Los tipos más utilizados son BIT, TINYINT, INT, BIGINT, para la representación de números enteros de menor o mayor tamaño.

Para número decimales tenemos los tipos FLOAT y DOUBLE, números en coma flotante de menor o mayor precisión, respectivamente. En este caso conviene tener en cuenta los problemas de precisión [1)] que existen con estos tipos de datos. Si necesitamos almacenar valores en los que es muy importante la precisión decimal, podemos utilizar DECIMAL(d,n), *d* representa la cantidad de digitos del valor, y *n* la cantidad de decimales después de la coma.

En ocasiones el rango de los valores negativos resultará prescindible (claves numéricas, valores de dinero, cantidades, . . .) por lo que será posible ampliar el rango positivo de un tipo numérico añadiendo la restricción UNSIGNED tras definir el tipo de éste.

```
id INT UNSIGNED
```

Tipos para fechas

Los tipos más utilizado para almacenar valores de fechas (DATE) o fechas con hora (DATETIME). Por defecto el formato utilizado es YYYY-MM-DD y YYYY-MM-DD HH:MM:SS respectivamente.

También se puede usar el tipo TIMESTAMP para almacenar una marca de tiempo (fecha y hora YYYY-MM-DD HH:MM:SS). Además, permite el uso de la constante CURRENT_TIMESTAMP en la definición de la columna al definirle un valor por defecto cuando se crea la tabla.

La diferencia entre TIMESTAMP frente a DATETIME, es que a pesar de mostrar el mismo formato, TIMESTAMP almacena también la zona horaria, y devuelve el valor modificado si cambiamos la zona horaria del servidor.

[1)] https://dev.mysql.com/doc/refman/5.5/en/problems-with-float.html

Tipo booleano

Permite almacenar valores lógicos Verdadero/Falso o Sí/No. Cuando usamos el tipo BOOLEAN para definir un campo, MySQL define internamente la columna como del tipo TINYINT, utilizando los valores 0 y 1 para indicar los valores lógicos true y false respectivamente. Así, podremos utilizar los valores TRUE ó FALSE o directamente asignar 1 ó 0 para asignar valor.

Restricciones

Las restricciones se pueden establecer, o no, a las columnas de cada tabla para forzar a que los datos almacenados en ellas cumplan una serie de condiciones, con la finalidad de que la información sea más correcta. Por ejemplo, podemos obligar a que un campo donde almacenamos el DNI de una persona tenga una longitud mínima, un campo donde almacenamos la categoría de un equipo de fútbol, sólo pueda tomar unos determinados valores predefinidos (benjamín, juvenil, cadete, . . .) o bien podemos hacer que un campo no pueda repetirse, por tratarse de un valor único (DNI, NSS, teléfono, email, . . .).

Hay que tener en cuenta que, por lo general, las restricciones se definen en línea con la definición del campo (tal y como se muestra en la sintaxis de la sentencia de CREATE TABLE, pero de forma opcional también pueden ser definidas por separado justo debajo de la definición de todos los campos de la tabla.

Clave primaria

Una clave primaria dentro de una tabla, es una columna o conjunto de columnas cuyo valor identifica unívocamente a cada fila. Debe ser única, no nula y es obligatoria. Como máximo podremos definir una clave primaria por tabla y es muy recomendable definirla.

Para definir una clave primaria utilizamos la restricción PRIMARY KEY.

```
CREATE TABLE personas(
  dni VARCHAR(9) PRIMARY KEY,
  ...
);
```

Y si lo hacemos al final de la definición de las columnas, quedaría así:

```
CREATE TABLE personas(
  dni VARCHAR(9),
  nombre VARCHAR(10),
  PRIMARY KEY (dni)
);
```

Hay que tener en cuenta que a la hora de definir claves primarias compuestas (dos ó más columnas), ésta deberá ser definida forzosamente tras la definición de los campos involucrados, siguiendo esta sintaxis

```
CREATE TABLE personas(
  dni VARCHAR(9),
  nombre VARCHAR(10),
  apellidos VARCHAR(20),
  PRIMARY KEY (nombre, apellidos)
);
```

Autonumérico

Es una propiedad que solo se puede aplicar a claves primarias de tipo entero. Hace que en cada inserción de nuevos registros en esa tabla, la clave primaria se genere automáticamente de forma secuencial.

Es realmente útil ya que nos evita tener que dar valor a esa columna, siendo el motor de la base de datos el encargado de que sea siempre distinta. De esta forma podemos crear **claves primarias subrogadas**.

La forma de definirlo es añadiendo la restricción AUTO_INCREMENT en la definición de la columna que se ha definido como clave primaria:

```
id INT PRIMARY KEY AUTO_INCREMENT
```

Como detalle, cualquier inserción fallida en la tabla seguirá incrementando el índice autonumérico aunque no se inserten los datos. Podemos resetearlo:

```
ALTER TABLE mitabla AUTO_INCREMENT = 1;
```

Si la tabla no está vacía, debe ser mayor que el último valor de esa columna.

Clave ajena

Una clave ajena está formada por una o varias columnas cuya finalidad es almacenar valores que existen en la columna/s clave primaria de otra tabla (o la misma) a la que hace referencia. De este modo los sistemas relacionales aseguran que las relaciones entre tablas tienen significado.

Al definir una columna como clave ajena FOREIGN KEY el motor de la base de datos nos obliga a que todos los valores que contenga esa columna existen siempre en la columna de la clave primaria de la tabla con la que se relaciona. Este concepto se conoce como **Integridad Referencial**.

Las claves ajenas se deben definir después de la definición de los campos de la tabla:

```
CREATE TABLE asignaturas(
  id INT GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,
  nombre VARCHAR(20),
  departamento VARCHAR(20),
  id_curso INT,
  FOREIGN KEY (id_curso) REFERENCES cursos (id)

  -- También se puede crear indicandole un nombre concreto a la restricción:
  CONSTRAINT 'fk_cursos' FOREIGN KEY (id_curso) REFERENCES cursos (id)
 -- O simplemente indicando REFERENCES después de la definición del campo:
 id_curso INT REFERENCES cursos(id)
);
```

El ejemplo anterior haría referencia tabla *cursos*:

```
CREATE TABLE cursos(
  id INT GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,
  horario VARCHAR(20),
  . . .
);
```

Si una tabla tiene diferentes claves ajenas, se definen del mismo modo:

```
  id_curso INT,
  id_profesor INT,

  FOREIGN KEY (id_curso) REFERENCES cursos (id),
  FOREIGN KEY (id_profesor) REFERENCES profesores (id)
```

Si la clave ajena hace referencia a una clave primaria compuesta:

```
FOREIGN KEY (id_curso, id_aula) REFERENCES cursos(id_curso, id_aula)
```

Las definiciónes de la columna y de la restricción de clave ajena se hacen de forma independiente, pero debemos tener en cuenta que para crear la restricción de clave ajena sobre una columna, **su tipo de datos debe ser exactamente igual al de la clave primaria a la que hace referencia**.

Habrá que tener en cuenta que mientras que un campo definido como clave ajena haga referencia a un campo definido como clave primaria, la fila de la segunda tabla no podrá ser eliminada hasta que no lo haga la fila que le hace referencia (integridad referencial). Para evitar estos problemas (aunque no siempre es un problema) es posible definir la restricción de clave ajena añadiendo la cláusula ON DELETE o bien ON UPDATE para el caso de una actualización. De esa manera, cuando se vaya a eliminar o actualizar una fila a cuya clave primaria se haga referencia, podremos indicar a MySQL que operación queremos realizar con las filas que le hacen referencia:

- **RESTRICT**: Se rechaza la operación de eliminación/actualización
- **CASCADE**: Realiza la operación y se elimina o actualiza en cascada en las filas que hacen referencia
- **SET NULL**: Realiza la operación y fija a NULL el valor en las filas que hacen referencia
- **NO ACTION**: Se rechaza la operación de eliminación/actualización, como ocurre con la opción RESTRICT

Si no especifico ningun tipo de acción, se tomará NO ACTION por defecto para ambas operaciones (UPDATE y DELETE); se restringe el borrado o actualización de valores de claves primarias si tiene claves ajenas referenciando.

```
-- No podré eliminar cursos si hay asignaturas referenciandolos:
FOREIGN KEY (id_curso) REFERENCES cursos (id) ON DELETE RESTRICT

-- Si modifico el valor de la clave primaria de un curso, se actualiza la clave
ajena
-- Si elimino un curso, se pondrá a NULL el valor de la clave ajena de las
asignaturas
FOREIGN KEY (id_curso) REFERENCES cursos (id) ON UPDATE CASCADE ON DELETE SET NULL
```

Consideraciones

Para definir claves ajenas en MySQL habrá que tener en cuenta algunas consideraciones:

- Una columna clave ajena nunca podrá ser AUTO_INCREMENT, ya que rompería el propósito de la relación.
- La columna/s clave ajena debe ser del mismo tipo de datos que la columna clave primaria a

las que se referencia.

- La columna deberá ser un índice. A partir de Mysql 8 y versiones semejantes de MariaDB, toda clave ajena define un índice automáticamente.
- Si la columna se define como obligatoria (NOT NULL) no podrá contener la claúsula (SET NULL) para los casos de borrado (ON DELETE) o actualización (ON UPDATE).
- Toda restricción de clave ajena tiene un nombre. Para ver el nombre de las restricciones de una tabla: SHOW CREATE TABLE <nombre_tabla>

Campos obligatorios

Esta restricción obliga a que se le tenga que dar valor obligatoriamente a una columna. Por tanto, no podrá tener el valor NULL. Se utiliza la palabra reservada NOT NULL.

```
  apellidos VARCHAR(250) NOT NULL
```

Valores por defecto

Se puede definir el valor que una columna tomará por defecto, es decir, si al introducir una fila no se especifica valor para dicha columna. Se utiliza la palabra reservada DEFAULT.

```
fecha  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
nombre VARCHAR(250) DEFAULT 'Sin nombre'
```

Condiciones

De forma más genérica, podemos forzar a que los valores de determinados campos de la tabla cumplan una ciertas condiciones.

Lo habitual es definir una columna como de tipo enumeración (ENUM en MySQL) si queremos indicar que solamente una serie de valores (definidos) son válidos:

```
curso   ENUM ('0', '1', '2'),
horario ENUM ('mañana', 'tarde', 'noche'),
```

Valores únicos

La restricción UNIQUE evita valores repetidos en una misma columna. Al contrario que ocurre con la restricción PRIMARY KEY, UNIQUE sí admite el valor NULL. Con respecto a esta última consideración, conviene saber que si una columna se define como UNIQUE, sólo una de sus filas podrá contener el valor NULL. Del mismo modo que la clave primaria, esta restricción se puede aplicar a una columna o a un conjunto de columnas.

```
email VARCHAR(100) UNIQUE

// o para varias columnas
. . .
email VARCHAR(100),
login VARCHAR(50),
UNIQUE (email, login)
```

Índices

Los índices se utilizan para obtener datos de las tablas de una forma más rápida. En definitiva, lo

que el SGBD hace es asociar el valor de una columna (sobre la que definimos el índice) con su posición en la tabla. De esa manera será más rápido buscar sobre esa columna puesto que al encontrar el valor, el SGBD conocerá su posición en la tabla.

Se recomienda su uso en aquellas columnas sobre las que se vayan a realizar búsquedas en una tabla. Por ejemplo, si tenemos una tabla donde almacenamos información sobre Libros, nos podría interesar crear un índice en el campo autor, puesto que puede ser muy común buscar qué libros ha escrito un autor determinado. Además, será un valor que contendrá pocos valores repetidos, por lo que maximizará el beneficio de usar un índice.

```
CREATE TABLE libro(
id INT ...,
titulo ...,
autor VARCHAR(20),
INDEX autor_index (autor)
);

-- También puedo añadirlo cuando la tabla ya se ha creado
CREATE INDEX indice_autor ON libro(autor);
```

Por otra parte, los índices presentan algún inconveniente como puede ser el hecho de que ocupan espacio en la tabla, y dependiendo del caso podría llegar a ocupar más espacio que la propia tabla, por lo que hay que tener cuidado a la hora de escoger una columna como índice. También hay que tener en cuenta que hay que actualizar el índice cada vez que se modifica la columna en la tabla por lo que no resulta conveniente elegir como índices aquellas columnas que creamos que van a escribirse con mucha frecuencia.

Usuarios y privilegios

Como se vió en el tema anterior, una de las funciones del SGBD es la de proporcionar seguridad en el acceso a los datos a través de mecanismos de control de acceso.

En SQL, y así lo hacen todos los SGBD relacionales, se sigue un modelo Usuario-Privilegio para otorgar acceso a los objetos de la Base de Datos. Existen una serie de privilegios predefinidos y es el administrador del SGBD el encargado de asignar o no los privilegios [2)] a los usuarios sobre determinados objetos (tablas, procedimientos, . . .).

Supongamos que somos los administradores de un SGBD y tenemos que proporcionar acceso a una Base de Datos para una aplicación *biblioteca* a un desarrollador de mi compañía:

```
-- Crea el usuario asignándole contraseña
CREATE USER 'desarrollador'@'localhost' IDENTIFIED BY 'micontraseña';
-- Asigna todos los privilegios al usuario
GRANT ALL PRIVILEGES TO desarrollador;
```

Así, hemos creado la Base de Datos y el usuario, y hemos concedido todos los privilegios a dicho usuario sobre esa Base de Datos.

Para comprobar los usuarios existentes en el sistema y sus permisos:

```
SELECT *
FROM mysql.user;
```

Conceder privilegios sobre un objeto: GRANT

Permite conceder privilegios sobre un objeto a un usuario de la Base de Datos.

```
GRANT <privilegio>
ON <objeto>
TO <usuario>
[WITH GRANT OPTIONS]
```

Revocar privilegios sobre un objeto: REVOKE

Permite eliminar el privilegio sobre un objeto a un usuario.

```
REVOKE <privilegio>
ON <objeto>
FROM <usuario>
```

Creación de scripts en MySQL

La forma más habitual de trabajo a la hora de lanzar órdenes en SQL sobre un SGBD relacional como MySQL es crear ficheros por lotes de órdenes SQL, lo que se conoce como *scripts SQL*, donde podemos escribir todas las sentencias SQL que queremos ejecutar una detrás de otra separadas por el carácter ;.

Existe la posibilidad de añadir comentarios al código según la siguiente sintaxis:

```
-- Esto es un comentario y MySQL no lo ejecuta
/* Esto también es un comentario y
     tampoco se ejecuta */
```

Por ejemplo, para la creación de una nueva Base de Datos y sus tablas podríamos preparar un script SQL como el siguiente:

```
CREATE DATABASE IF NOT EXISTS pagina_web;
USE pagina_web;

CREATE TABLE IF NOT EXISTS usuarios (
```

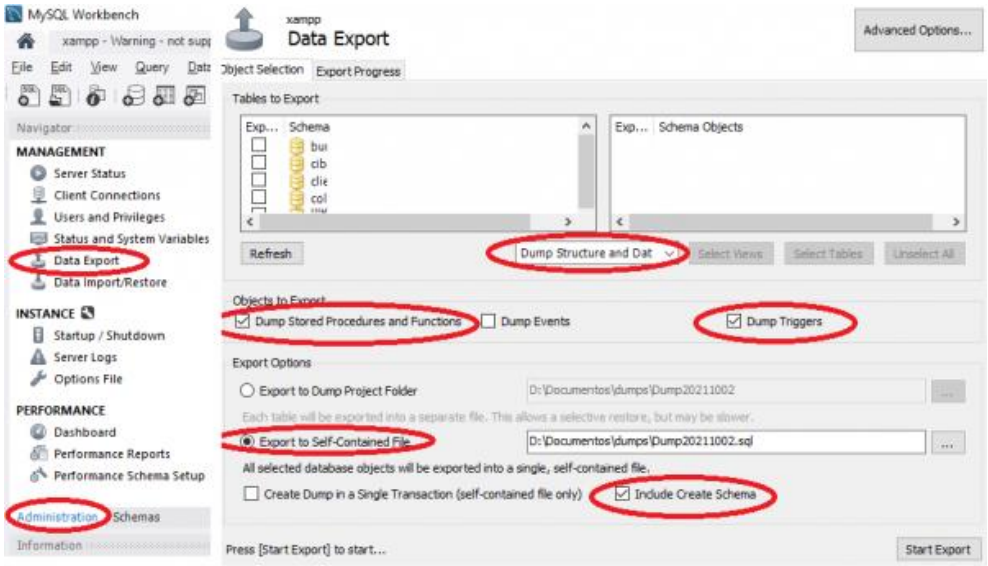
```
  id INT PRIMARY KEY AUTO_INCREMENT,
  . . .
  . . .
);
CREATE TABLE IF NOT EXISTS productos (
  . . .
  . . .
);
. . .
. . .
```

Una vez creado el script podremos lanzar su ejecución sobre MySQL y se ejecutarán todas las sentencias contenidas en él de forma secuencial. Es una forma muy útil de crear scripts para la creación de una Base de Datos y todas sus tablas y restricciones y también para crear scripts de actualización o parcheo de una Base de Datos existentes de forma que se incluyan todas las sentencias SQL que actualicen o arreglen los problemas que actualmente pueda haber (añadir una nueva tabla, eliminar un campo, añadir una nueva restricción, . . .).

Exportar script desde MySql Workbench

Desde la sección *Administration* del panel *Navigation* que está a la izquierda de la interfaz, haremos click sobre *Data Export*.

En la siguiente ventana debemos indicar varias cosas:



- *Dump Structure and Data*: exportar la estructura y los datos contenidos.
- *Dump Stores Procedures…*, *Dump Triggers*, etc: exportar procedimientos, funciones o triggers.
- *Export to Self-Contained File*: Para exportar toda la base de datos en un solo fichero (Opción recomendable, podemos dar nombre al fichero). En caso contrario creará un fichero por cada tabla.
- *Include Create Schema*: incluye las instrucciones CREATE DATABASE [nombre] y USE [nombre]

Comprobaciones sobre el script SQL

1. Utilizar notación snake_case para todos los identificadores (nombre de la base de datos, nombres de tablas, nombres de columnas, . . .). Y siempre en minúscula
2. No utilizar acentos, el caracter ñ ni otros caracteres extraños (|@#...) para nombres de bases de datos, tablas, columnas o cualquier otro elemento
3. Escribir las palabras reservadas del lenguaje SQL en mayúsculas
4. Las claves ajenas indicarán la tabla a la que hacen referencia (en singular) como parte de su nombre. Por ejemplo: id_usuario si es una clave ajena de una tabla usuarios. Si en una tabla hay dos claves ajenas que apuntan a la misma tabla, añadiremos algo al nombre para distinguirla (id_usuario_emisor e id_usuario_receptor, por ejemplo)
5. Se recomienda que los nombres de las tablas sean en plural (users mejor que user, orders mejor que order)
6. Antes de definir un tipo de dato como numérico, comprobar si realmente voy a operar con él como tal
7. Cuidado con los campos contraseña. Realmente nunca se guarda tal cual sino como un hash utilizando algún algoritmo, por lo que la longitud real es mayor (la longitud de un hash creado con SHA1 es de 40 caracteres y con SHA2 hasta 128)