Introducción a SQL

Tema 1. Consultas básicas con una tabla

1.1. Creación de una tabla

Una base de datos, en general, estará formada por varios bloques de información llamados "**tablas**". En nuestro caso, nuestra única tabla almacenará los datos de nuestros amigos. Por tanto, el siguiente paso será decidir qué datos concretos (lo llamaremos "**campos**") guardaremos de cada amigo. Deberemos pensar también qué tamaño necesitaremos para cada uno de esos datos, porque al gestor de bases de datos habrá que dárselo bastante cuadriculado. Por ejemplo, podríamos decidir lo siguiente:

nombre - texto, hasta 20 letras

dirección - texto, hasta 40 letras

edad - números, de hasta 3 cifras

Cada gestor de bases de datos tendrá una forma de llamar a esos tipos de datos. Por ejemplo, en SQLite podemos usar "VARCHAR" para referirnos a texto hasta una cierta longitud variable, y "NUMERIC" o “int” para números de una determinada cantidad de cifras. Además, en la mayoría de gestores de base de datos, será recomendable (a veces incluso obligatorio) que los nombres de los campos no contengan acentos ni caracteres internacionales, y los nombres de tablas y campos se suelen indicar en minúsculas, de modo que la orden necesaria para crear esta tabla sería:

```
CREATE TABLE personas (
  nombre varchar(20),
  direccion varchar(40),
  edad numeric(3)
);
```

1.2. Introducción de datos

Para introducir datos usaremos la orden "**INSERT INTO**", e indicaremos tras la palabra "**VALUES**" los valores de los campos, en el mismo orden en el que se habían definido los campos. Los valores de los campos de texto se deberán indicar entre comillas, y los valores para campos numéricos ni necesitarán comillas, así:

```
INSERT INTO personas VALUES ('juan', 'su casa', 25);
```

Si no queremos indicar todos los datos, deberemos detallar qué campos vamos a introducir y en qué orden, así:

```
INSERT INTO personas (nombre, direccion) VALUES ('Héctor', 'calle 1');
```

(Como se ve en este ejemplo, los datos sí podrán contener acentos y respetar las mayúsculas y minúsculas donde sea necesario).

De igual modo, también será necesario indicar los campos si se desea introducir los datos en un orden distinto al de definición:

```
INSERT INTO personas (direccion, edad, nombre) VALUES ("Calle 2", 30, "Daniel");
```

(Y como puedes apreciar en este ejemplo, los campos de texto se pueden indicar también entre comillas dobles, si te resulta más cómodo).

1.3. Mostrar datos

Para ver los datos almacenados en una tabla usaremos el formato "SELECT campos FROM tabla". Si queremos ver todos los campos, lo indicaremos con un asterisco:

```
SELECT * FROM personas;
```

que, en nuestro caso, daría como resultado

```
+---------+-----------+------+
| nombre  | direccion | edad |
+---------+-----------+------+
| juan    | su casa   |   25 |
| Héctor  | calle 1   | NULL |
| Daniel  | Calle 2   |   30 |
+---------+-----------+------+
```

Como puedes observar, no habíamos introducido los detalles de la edad de Héctor. No se muestra un cero ni un espacio en blanco, sino que se anota que hay un "**valor nulo**" (NULL) para ese campo. Usaremos esa peculiaridad más adelante.

Si queremos ver **sólo ciertos campos**, deberemos detallar sus nombres, en el orden deseado, separados por comas:

```
SELECT nombre, direccion FROM personas;
```

y obtendríamos

```
+---------+-----------+
| nombre  | direccion |
+---------+-----------+
| juan    | su casa   |
| Héctor  | calle 1   |
| Daniel  | Calle 2   |
+---------+-----------+
```

1.4. Buscar por contenido

Normalmente no querremos ver todos los datos que hemos introducido, sino sólo aquellos que cumplan cierta condición. Esta condición se indica añadiendo un apartado **WHERE** a la orden "SELECT". Por ejemplo, se podrían ver los datos de una persona a partir de su nombre de la siguiente forma:

```
SELECT nombre, direccion FROM personas WHERE nombre = 'juan';
+--------+-----------+
| nombre | direccion |
+--------+-----------+
| juan   | su casa   |
+--------+-----------+
```

En ocasiones no querremos comparar todo el campo con un texto exacto, sino ver si contiene un cierto texto (por ejemplo, porque sólo sepamos un apellido o parte de la calle). En ese caso, no compararíamos con el símbolo "igual" (=), sino que usaríamos la palabra "**LIKE**", y para las partes que no conozcamos usaremos el comodín "%", como en este ejemplo:

```
SELECT nombre, direccion FROM personas WHERE direccion LIKE '%calle%';
```

que nos diría el nombre y la dirección de nuestros amigos que viven en lugares que contengan la palabra "calle", aunque ésta pueda estar precedida por cualquier texto (%) y quizá también con cualquier texto (%) a continuación:

```
+---------+-----------+
| nombre  | direccion |
+---------+-----------+
| Héctor  | calle 1   |
| Daniel  | Calle 2   |
+---------+-----------+
```

1.5. Mostrar datos ordenados

Cuando una consulta devuelva muchos resultados, resultará poco legible si éstos se encuentran desordenados. Por eso, será frecuente añadir "ORDER BY" seguido del nombre del campo que se quiere usar para ordenar (por ejemplo "ORDER BY nombre").

Es posible indicar varios campos de ordenación, de modo que se mire más de un criterio en caso de que un campo coincida (por ejemplo "ORDER BY apellidos, nombre").

Si se quiere que alguno de los criterios de ordenación se aplique en orden contrario (por ejemplo, textos de la Z a la A o números del valor más grande al más pequeño), se puede añadir la palabra "DESC" (de "descending", "descendiendo") al criterio correspondiente, como en "ORDER BY precio DESC". Si no se indica esa palabra, se sobreentiende que se quiere obtener los resultados en orden ascendente (lo que indicaría la palabra "ASC", que es opcional).

```
SELECT nombre, direccion
FROM personas
WHERE direccion LIKE '%calle%'
ORDER BY nombre, direccion DESC;
+---------+-----------+
| nombre  | direccion |
+---------+-----------+
| Daniel  | Calle 2   |
```

```
| Héctor  | calle 1   |
+---------+-----------+
```

1.6. Ejercicios propuestos

1.1. Crea una tabla llamada "ciudades". De cada ciudad se deseará guardar su nombre (de 40 letras o menos) y su población (de 9 cifras o menos).

1.2. Usando el formato abreviado de INSERT, añade la ciudad "Alicante", con 328.648 habitantes.

1.3. Indicando primero la población y luego el nombre, añade una ciudad de 3.141.991 habitantes llamada "Madrid".

1.4. Muestra todos los datos de todas las ciudades existentes en la base de datos.

1.5. Muestra la población de la ciudad llamada "Madrid".

1.6. Muestra el nombre y la población de todas las ciudades que contienen una letra "e" en su nombre.

1.7. Muestra el nombre y la población de todas las ciudades, ordenadas de la más poblada a la menos poblada. Si dos o más ciudades tienen misma cantidad de habitantes, se mostrarán ordenadas de la ‘A’ a la ‘Z’.

1.8. Unifica todos los ejercicios en un único archivo.

Tema 2. Consultas básicas con dos tablas

2.1 Formalizando conceptos

Hay una serie de detalles que hemos pasado por alto y que deberíamos formalizar un poco antes de seguir:

- **SQL** es un lenguaje de consulta a bases de datos. Sus siglas vienen de **Structured Query Language** (lenguaje de consulta estructurado).
- **MySQL** es un "gestor de bases de datos", es decir, una aplicación informática que se usa para almacenar, obtener y modificar datos (realmente, se les exige una serie de cosas más, como controlar la integridad de los datos o el acceso por parte de los usuarios, pero por ahora nos basta con eso).
- En SQL, las órdenes que tecleamos deben terminar en **punto y coma** (;). Si tecleamos una orden como "SELECT * FROM personas", sin el punto y coma final, y pulsamos Intro, MySQL responderá mostrando "->" para indicar que todavía no hemos terminado la orden y nos queda algo más por introducir. Precisamente por eso, las órdenes se pueden partir para que ocupen varias líneas, como hemos hecho con "CREATE TABLE".
- Las órdenes de SQL se pueden introducir en mayúsculas o minúsculas. Aun así, es frecuente emplear mayúsculas para las órdenes y minúsculas para los campos y tablas, de modo que la sentencia completa resulte más legible. Así, para mostrar todos los nombres almacenados en nuestra tabla personas, se podría hacer con:

```
SELECT nombre FROM personas;
```

2.2 ¿Por qué varias tablas?

Puede haber varios motivos:

Por una parte, podemos tener bloques de información claramente distintos. Por ejemplo, en una base de datos que guarde la información de una empresa tendremos datos como los artículos que distribuimos y los clientes que nos los compran, que no deberían guardarse en una misma tabla.

Por otra parte, habrá ocasiones en que descubramos que los datos, a pesar de que se podrían clasificar dentro de un mismo "bloque de información" (tabla), serían redundantes: existiría gran cantidad de datos repetitivos, y esto puede dar lugar a dos problemas:

- Espacio desperdiciado.
- Posibilidad de errores al introducir los datos, lo que daría lugar a inconsistencias:

Veamos unos datos de ejemplo, que podrían ser parte de una tabla ficticia:

```
+---------+----------+----------+
| nombre  | direccion | ciudad    |
+---------+----------+----------+
| juan    | su casa   | alicante  |
| alberto | calle uno | alicante  |
| pedro   | su calle  | alicantw  |
+---------+----------+----------+
```

Si en vez de repetir "alicante" en cada una de esas fichas, utilizásemos un código de ciudad, por ejemplo "a", gastaríamos menos espacio (en este ejemplo, 7 bytes menos en cada ficha; con la capacidad de un ordenador actual eso no sería un gran problema).

Por otra parte, hemos tecleado mal uno de los datos: en la tercera ficha no hemos indicado "alicante", sino "alicantw", de modo que, si hacemos consultas sobre personas de Alicante, la última de ellas no aparecería. Al teclear menos, es también más difícil cometer este tipo de errores.

A cambio, necesitaremos una segunda tabla, en la que guardemos los códigos de las ciudades, y el nombre al que corresponden (por ejemplo: si codigoDeCiudad = "a", la ciudad es "alicante").

Esta tabla se podría crear así:

```
CREATE TABLE ciudades (
 codigo varchar(3),
 nombre varchar(30)
);
```

2.3 Las claves primarias

Generalmente, y especialmente cuando se usan varias tablas enlazadas, será necesario tener algún dato que nos permita distinguir de forma clara los datos que tenemos almacenados. Por ejemplo, el nombre de una persona no es único: pueden aparecer en nuestra base de datos varios usuarios llamados "Juan López". Si son nuestros clientes, debemos saber cuál es cuál, para no cobrar a uno de ellos un dinero que corresponde a otro. Eso se suele solucionar guardando algún dato adicional que sí sea único para cada cliente, como puede ser el Documento Nacional de Identidad, o el Pasaporte. Si no hay ningún dato claro que nos sirva, en ocasiones añadiremos un "código de cliente", inventado por nosotros, o algo similar.

Estos datos que distinguen claramente unas "fichas" de otras los llamaremos "**claves primarias**".

Se puede crear una tabla indicando su calve primaria si se añade "PRIMARY KEY" al final de la orden "CREATE TABLE", así:

```
CREATE TABLE ciudades (
  codigo varchar(3),
  nombre varchar(30),
  PRIMARY KEY (codigo)
);
```

o bien al final de la definición del campo correspondiente, así:

```
CREATE TABLE ciudades (
  codigo varchar(3) PRIMARY KEY,
  nombre varchar(30)
);
```

2.4 Creando datos

Comenzaremos creando una nueva base de datos, de forma similar al ejemplo anterior:

```
CREATE DATABASE ejemplo2;
USE ejemplo2;
```

Después creamos la tabla de ciudades, que guardará su nombre y su código. Este código será el que actúe como "clave primaria", para distinguir otra ciudad. Por ejemplo, hay una ciudad llamada "Toledo" en España, pero también otra en Argentina, otra en Uruguay, dos en Colombia, una en Ohio (Estados Unidos) ... el nombre claramente no es único, así que podríamos usar código como "te" para Toledo de España, "ta" para Toledo de Argentina y así sucesivamente.

La forma de crear la tabla con esos dos campos y con esa clave primaria sería:

```
CREATE TABLE ciudades (
  codigo varchar(3),
  nombre varchar(30),
  PRIMARY KEY (codigo)
);
```

Mientras que la tabla de personas sería casi igual al ejemplo anterior, pero añadiendo un nuevo dato: el código de la ciudad

```
CREATE TABLE personas (
  nombre varchar(20),
  direccion varchar(40),
  edad decimal(3),
  codciudad varchar(3)
);
```

Para introducir datos, el hecho de que exista una clave primaria no supone ningún cambio, salvo por el hecho de que no se nos dejaría introducir dos ciudades con el mismo código:

```
INSERT INTO ciudades VALUES ('a', 'alicante');
INSERT INTO ciudades VALUES ('b', 'barcelona');
INSERT INTO ciudades VALUES ('m', 'madrid');

INSERT INTO personas VALUES ('juan', 'su casa', 25, 'a');
INSERT INTO personas VALUES ('pedro', 'su calle', 23, 'm');
INSERT INTO personas VALUES ('alberto', 'calle uno', 22, 'b');
```

2.5 Mostrando datos

Cuando queremos mostrar datos de varias tablas a la vez, deberemos hacer unos pequeños cambios en las órdenes "select" que hemos visto:

- En primer lugar, indicaremos varios nombres después de "FROM" (los de cada una de las tablas que necesitemos).
- Además, puede ocurrir que tengamos campos con el mismo nombre en distintas tablas (por ejemplo, el nombre de una persona y el nombre de una ciudad), y en ese caso deberemos escribir el nombre de la tabla antes del nombre del campo.

Por eso, una consulta básica sería algo parecido (sólo parecido) a:

```
SELECT personas.nombre, direccion, ciudades.nombre FROM personas, ciudades;
```

Pero esto todavía tiene problemas: estamos combinando TODOS los datos de la tabla de personas con TODOS los datos de la tabla de ciudades, de modo que obtenemos 3x3 = 9 resultados:

```
+---------+----------+----------+
| nombre  | direccion | nombre    |
+---------+----------+----------+
| juan    | su casa   | alicante  |
| pedro   | su calle  | alicante  |
| alberto | calle uno | alicante  |
| juan    | su casa   | barcelona |
| pedro   | su calle  | barcelona |
| alberto | calle uno | barcelona |
| juan    | su casa   | madrid    |
| pedro   | su calle  | madrid    |
| alberto | calle uno | madrid    |
+---------+----------+----------+
9 rows in set (0.00 sec)
```

Pero esos datos no son reales: si "juan" vive en la ciudad de código "a", sólo debería mostrarse junto al nombre "alicante". Nos falta indicar esa condición: "el código de ciudad que aparece en la persona debe ser el mismo que el código que aparece en la ciudad", así:

```
SELECT personas.nombre, direccion, ciudades.nombre
FROM personas, ciudades
WHERE personas.codciudad = ciudades.codigo;
```

Esta será la forma en que trabajaremos normalmente. Este último paso se puede evitar en ciertas circunstancias, pero ya lo veremos más adelante. El resultado de esta consulta sería:

```
+---------+----------+-----------+
| nombre  | direccion | nombre    |
+---------+----------+-----------+
| juan    | su casa   | alicante  |
| alberto | calle uno | barcelona |
| pedro   | su calle  | madrid    |
+---------+----------+-----------+
```

Ese sí es el resultado correcto. Cualquier otra consulta que implique las dos tablas deberá terminar comprobando que los dos códigos coinciden. Por ejemplo, para ver qué personas viven en la ciudad llamada "madrid", haríamos:

```
SELECT personas.nombre, direccion, edad
FROM personas, ciudades
WHERE ciudades.nombre='madrid'
AND personas.codciudad = ciudades.codigo;
+--------+----------+------+
| nombre | direccion | edad |
+--------+----------+------+
| pedro  | su calle  |   23 |
+--------+----------+------+
```

Y para saber las personas de ciudades que comiencen con la letra "b", haríamos:

```
SELECT personas.nombre, direccion, ciudades.nombre
FROM personas, ciudades
```

```
WHERE ciudades.nombre LIKE 'b%'
AND personas.codciudad = ciudades.codigo;
+---------+----------+----------+
| nombre  | direccion | nombre    |
+---------+----------+----------+
| alberto | calle uno | barcelona |
+---------+----------+----------+
```

Si en nuestra tabla puede haber algún dato que se repita, como la dirección, podemos pedir un listado sin duplicados, usando la palabra "distinct":

```
SELECT DISTINCT direccion FROM personas;
```

2.6. Ejercicios propuestos

2.1. Crea una base de datos llamada "ejercicio2". En ella guardaremos información (muy poca) de marcas y modelos de coches. De cada marca almacenaremos el nombre y el país de origen, junto con un código de 3 letras. Para cada modelo anotaremos la marca, el nombre y el segmento al que pertenece (por ejemplo, "urbano", "compacto", "familiar", "todoterreno", etc.) Usaremos sólo dos tablas: una para marcas y otra para modelos, y sólo usaremos clave principal en "marcas".

2.2. Usando el formato detallado de INSERT, añade la marca "Ferrari", con país de origen "Italia". Su código será "F". Añade también, con código "SAL", la marca "Saleen", de "Estados Unidos".

2.3. Con el formato abreviado de INSERT, añade el modelo "S7" de "Saleen", que pertenece al segmento llamado "deportivo". En el mismo segmento, añade el F40 de Ferrari.

2.4. Muestra las marcas y modelos de todos los coches del segmento "deportivo".

2.5. Muestra marca, país y modelo de todos los vehículos cuya marca comienza por "F".

2.6. Muestra país, modelo y segmento de los coches cuyo modelo contenga una letra "S".

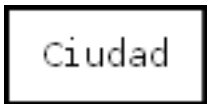
Tema 3. Contacto con los diagramas Entidad-Relación

3.1. ¿Qué es modelo entidad-relación?

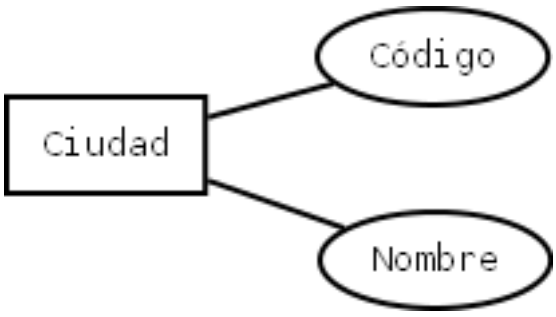
Es una notación alternativa, gráfica, que se suele usar como paso inicial en la planificación de una base de datos. Permite pasar de una descripción textual de un sistema de información a una representación visual, que resulta más fácil de comprobar y de refinar.
No profundizaremos mucho, pero veremos varios ejemplos de complejidad creciente, así como la manera de obtener un esquema de base de datos a partir de un diagrama Entidad-Relación.

3.2. Representación de entidades y atributos

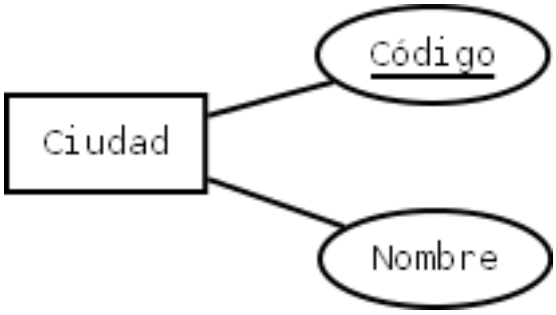
Una entidad representa un bloque de información. En general, como primera aproximación, se corresponderá con una tabla de la base de datos. Las entidades se representan dentro de rectángulos. Por ejemplo, una primera aproximación a una entidad "Ciudad", que representara a cada ciudad de la que vamos a guardar información, podría ser:



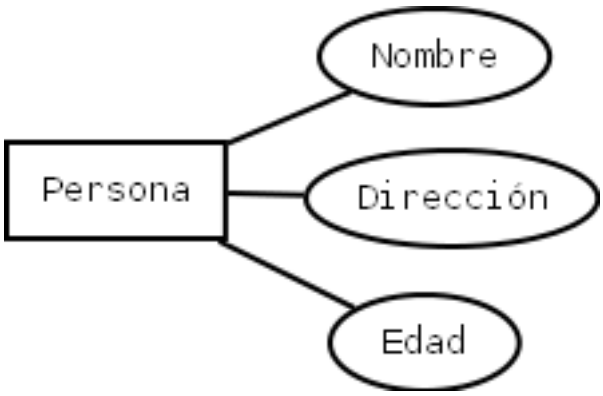
La forma más habitual de representar los atributos (las propiedades que cada entidad, que equivalen a los "campos" de la tabla) es dentro de elipses que estarán unidas a la entidad a la que pertenecen:



Si una entidad tiene clave primaria, se indica subrayando su nombre:



Y la entidad "persona" será similar, con todos sus atributos en elipses (y ninguno estará subrayado, al no tener clave primaria):



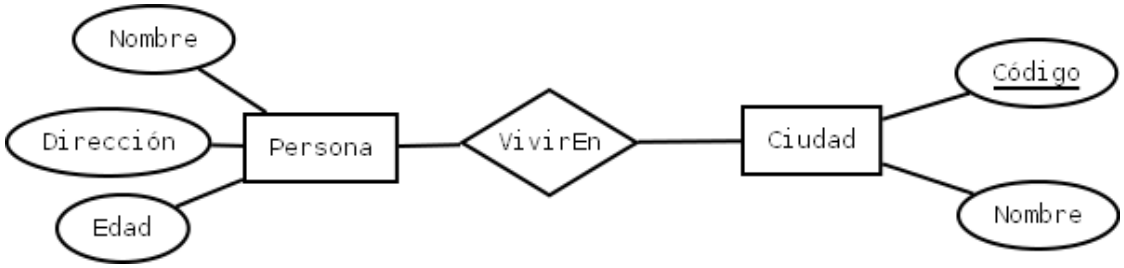
Existe una notación alternativa, más compacta, que no sigue el estándar entidad-relación, pero que puede llegar a resultar útil en caso de sistemas de información muy grandes. Consiste en incluir en un mismo rectángulo el nombre de la tabla y la lista de sus atributos (y opcionalmente incluso sus tipos de datos):

Persona	
Nombre	Txt
Dirección	Txt
Edad	Num

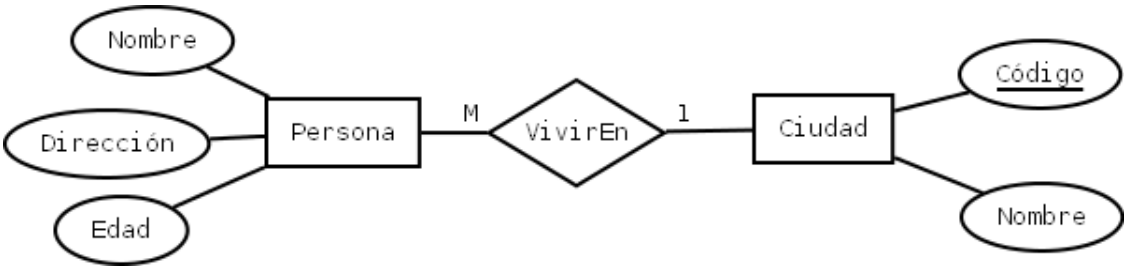
Nosotros no usaremos esa notación, porque está más relacionada con la implementación de la base de datos que con su diseño.

3.3. Relaciones

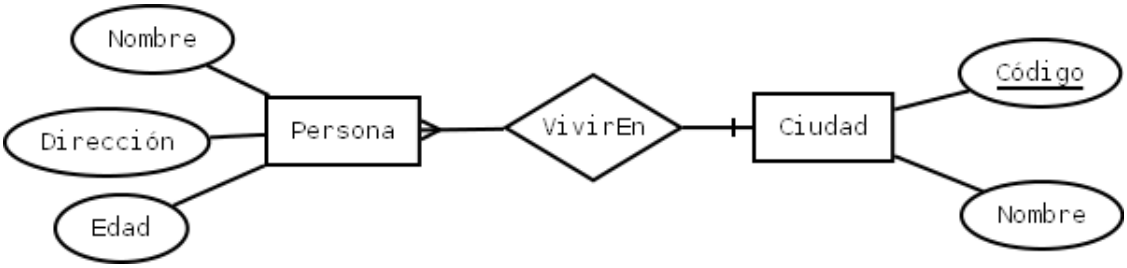
El nombre del modelo entidad-relación se debe a que son tan importantes las entidades individuales como las relaciones que existen entre ellas. Las relaciones normalmente se indican dentro de rombos, y tendrán un nombre que será un verbo. Por ejemplo, una persona puede "vivir en" una cierta ciudad, lo que se reflejaría en algo como:



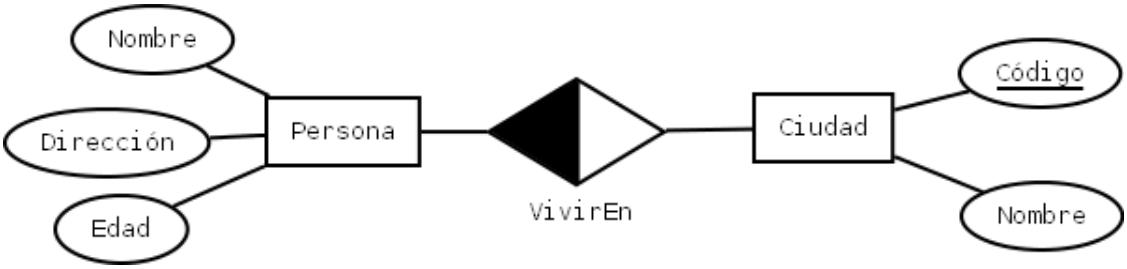
Un segundo detalle importante en las relaciones es la "cardinalidad": cuántas "ocurrencias" de la tabla A se relacionan con cuántas de la tabla B. Por ejemplo, si hablamos de personas que viven actualmente en ciudades, podríamos suponer que la relación es "de muchos a uno" (M:1), porque muchas personas viven en cada ciudad, mientras que cada persona sólo vive (en un momento dado, siendo optimistas) en una única ciudad.
Existen varias formas de reflejar esa cardinalidad. Una de las más sencillas es usar un "1" o una "M" en los correspondientes lados de la relación:



Otra es emplear "palitos": un palo perpendicular a la línea en la entidad que sólo aparece una vez, y tres palos (típicamente en forma de "tenedor", aunque hay quien los dibuja todos ellos perpendiculares) en el lado del "muchos", así:



Y otra es sombrear el lado del "muchos":



Personalmente, esta última notación es la que me parece más legible, porque permite saber de un vistazo rápido la cardinalidad, especialmente en los casos de relaciones más complejas, como las "ternarias", en las que intervienen tres entidades, y que veremos más adelante.

3.4. Pero hay más...

Hay muchos más detalles que se pueden representar en el modelo Entidad-Relación, como por ejemplo la "cardinalidad mínima" de una relación, es decir, el hecho de que un cierto dato debe existir siempre en una relación. Pero por ahora no vamos a afinar más.

3.5. Conversión a tablas

Una de las ventajas del modelo Entidad-Relación es que la conversión de un diagrama a la correspondiente base de datos (usando el "modelo relacional", que es el más extendido y el que estamos empleando nosotros), es casi inmediata.
Igual que no vamos a ver todas las posibilidades del Entidad-Relación, tampoco vamos a ver todas las pautas de conversión, pero sí las más sencillas (que además son las más habituales):

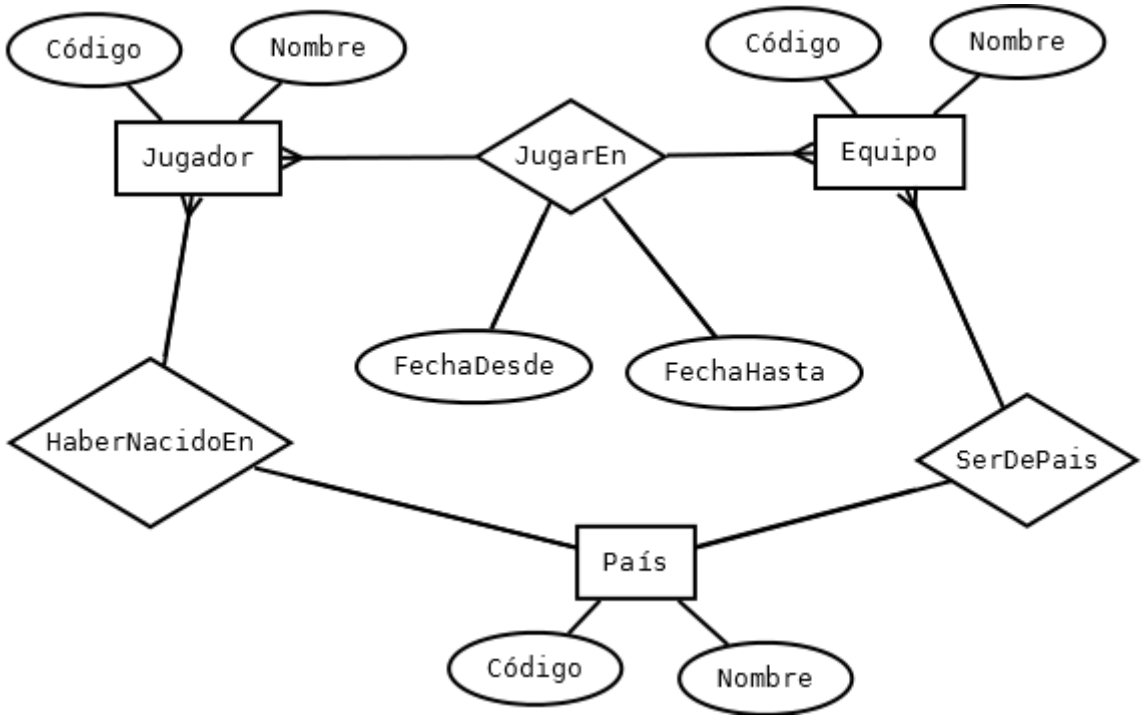
- Por lo general, una entidad se convertirá en una tabla.
- Una relación 1:M (uno a muchos) se reflejará añadiendo el código de la tabla del "lado del 1" en la tabla del "lado del M". Por ejemplo, en el caso de las personas y ciudades, en las que cada ciudad tiene muchas personas y cada persona vive en una ciudad, se añadirá para cada persona el código de la ciudad.
- Una relación M:M (muchos a muchos) se convertirá en una nueva tabla que contendrá ambas claves primarias (y su clave primaria estará formada por ambas claves a la vez). Por ejemplo, si nos interesa reflejar en nuestro sistema de información que cada persona puede haber vivido en varias ciudades, la relación pasaría a ser M:M, y entonces aparecería una nueva tabla "vivir", cuyos registros contendrían la clave de la ciudad y la clave de la persona (dato que actualmente no aparece en nuestra base de datos).

Vamos a ver un ejemplo un poco más completo antes de pasar a la tanda de ejercicios:

- Queremos informatizar un primer bloque de información de sobre equipos deportivos. Para cada equipo nos interesará saber su nombre, su país y la lista de sus jugadores. De cada jugador también querremos saber por ahora sólo el nombre y su país de nacimiento. Además, para que no sea tan trivial, querremos contemplar la posibilidad de que un jugador pueda haber formado parte de distintos equipos, en cada uno de ellos entre ciertas fechas.

De lo anterior se puede extraer que habrá dos entidades ("equipo" y "jugador"), que estarán relacionados por "jugar en" (M:M). Además, el dato del país, que sería repetitivo, debería sacarse a una nueva tabla, que estará relacionada 1:M con ambas tablas (cada jugador habrá nacido en un país, cada equipo pertenecerá a un país).

El diagrama sería así:



Y su conversión a tablas sería:

- País: código, nombre
- Equipos: código, nombre, códigoDePaís
- Jugadores: código, nombre, códigoDePaís
- JugarEn: códigoJugador, códigoEquipo, fechaDesde, fechaHasta

3.6. Ejercicios propuestos

3.1. Deseamos informatizar una lista de empleados técnicos de nuestra empresa, que sean capaces de resolver problemas de nuestros clientes. Por eso, para cada empleado nos interesará guardar información sobre todas sus habilidades técnicas (por ejemplo, "bases de datos" o "programación") así como los idiomas que maneja con soltura (por ejemplo, "inglés" o "alemán"). Como puede haber varias personas que tengan una cierta habilidad técnica o que hablen un cierto idioma, usaremos tablas para esos datos y relaciones "muchos a muchos". Además, querremos valorar de 1 a 5 el nivel que cada empleado tiene con una habilidad técnica o con un idioma (distinguiendo en este caso

entre nivel hablado y nivel escrito). Crea un diagrama Entidad-Relación que muestre cómo automatizar este sistema de información. (Pista: las preguntas 3.3 a 3.7 te ayudarán a plantear qué estructura da respuesta a todo lo que se puede necesitar, así como a saber qué tipo de datos puedes emplear)

3.2. Convierte a tablas el sistema Entidad-Relación, dentro de una nueva base de datos llamada "ejercicio3".

3.3. Añade a los usuarios (y habilidades e idiomas) los siguientes datos:

- Aurora, con nivel de 5 estrellas en PHP, 4 estrellas en Javascript, 5 estrellas en diseño gráfico y 3 estrellas en idioma inglés.
- Adrián, con nivel de 4 estrellas en PHP, 4 estrellas en Javascript, 5 estrellas en montaje de equipos y 2 estrellas en idioma inglés.
- Enrique, con 5 estrellas en electrónica y 2 estrellas en idioma inglés.
- Gala, con 5 estrellas en inglés, 5 estrellas en francés y 5 estrellas en atención al cliente.

3.4. Muestra el nombre de todas las personas que hablen francés.

3.5. Muestra los nombres de los empleados con conocimientos de diseño gráfico, ordenados del más experto (5 estrellas) al menos experto (1 estrella).

3.6. Muestra las habilidades de Adrián, ordenadas de aquella en la que es más experto a aquella en la que menos. Si dos habilidades coinciden, deberás ordenarlas alfabéticamente.

3.7. Nos llama un cliente que sólo habla inglés y que quiere hacer una consulta técnica. Por eso, deberás obtener los nombres de los empleados con conocimientos de PHP y de inglés, ordenados de mayor a menor nivel de inglés, y, en caso de coincidir, de mayor a menor nivel de PHP.

Tema 4. Borrar información

4.1. ¿Qué información hay?

Un primer paso antes de ver cómo borrar información es saber qué información tenemos almacenada.

Podemos saber las bases de datos que hay creadas en nuestro sistema con:

```
SHOW DATABASES;
```

Una vez que estamos trabajando con una base de datos concreta (con la orden "USE"), podemos saber las tablas que contiene con:

```
SHOW TABLES;
```

Y para una tabla concreta, podemos saber los campos (columnas) que la forman con "SHOW COLUMNS FROM":

```
SHOW COLUMNS FROM personas;
```

Por ejemplo, esto daría como resultado:

```
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| nombre    | varchar(20)  | YES  |     | NULL    |       |
| direccion | varchar(40)  | YES  |     | NULL    |       |
| edad      | decimal(3,0) | YES  |     | NULL    |       |
| codciudad | varchar(3)   | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

4.2. Borrar toda la base de datos

En alguna ocasión, como ahora que estamos practicando, nos puede interesar borrar toda la base de datos. La orden para conseguirlo es "DROP DATABASE":

```
DROP DATABASE ejemplo2;
```

Si esta orden es parte de una secuencia larga de órdenes, que hemos cargado por ejemplo con la orden "source", puede ocurrir que la base de datos no exista. En ese caso, obtendríamos un mensaje de error y se interrumpiría el proceso en ese punto. Podemos evitarlo añadiendo "IF EXISTS", para que se borre la base de datos sólo si realmente existe:

```
DROP DATABASE ejemplo2 IF EXISTS;
```

4.3. Borrar una tabla

Es más frecuente que creemos alguna tabla de forma incorrecta. La solución razonable es corregir ese error, cambiando la estructura de la tabla, pero todavía no sabemos hacerlo. De momento veremos cómo borrar una tabla. La orden es "DROP TABLE":

```
DROP TABLE personas;
```

Al igual que para las bases de datos, podemos hacer que la tabla se borre sólo cuando realmente existe:

```
DROP TABLE personas IF EXISTS;
```

4.4. Borrar datos de una tabla

También podemos borrar los datos que cumplen una cierta condición. La orden es "DELETE FROM", y con la cláusula "WHERE" indicamos las condiciones que se deben cumplir, de forma similar a como hacíamos en la orden "SELECT":

```
DELETE FROM personas WHERE nombre = 'juan';
```

Esto borraría todas las personas llamadas "juan" que estén almacenadas en la tabla "personas".

Cuidado: si no se indica el bloque "WHERE", no se borrarían los datos que cumplen una condición (porque NO HAY condición), sino TODOS los datos existentes. Si es eso lo que se pretende, una forma más rápida de conseguirlo es usar "TRUNCATE TABLE":

```
TRUNCATE TABLE personas;
```

4.5. Ejercicios propuestos

4.1. Crea una base de datos llamada "ejercicio4". En ella guardaremos información de artículos de revistas. De cada revista almacenaremos el nombre, el mes y el año, junto con un código de no más de 8 letras. Para cada artículo anotaremos un código, el título, la revista en la que aparece, la página inicial y la página final (se trata de una relación 1:M, ya que cada revista puede contener varios artículos y cada artículo sólo aparecerá en una revista). Diseña el diagrama Entidad-Relación y crea las tablas.

4.2. Añade la revista "Byte 9", del mes 10 de 1984, con código "BY009". Añade también la revista "PcWorld España 195", del mes 2 de 2003, con código "PCWE195".

4.3. Incluye también los artículos:

- "The IBM PC AT", con código "AT", en la revista Byte 9, de la página 108 a la 111.
- "Database Types", con código "DbTypes", en la revista Byte 9, de la página 138 a la 142.
- "12 Distribuciones Linux", con código "DistLinux", en la revista PCWE195, de la página 96 a la 109.

4.4. Muestra todos los artículos, ordenados por año, mes y título.

4.5. Muestra todos los artículos de revistas "Byte" que contengan la palabra "PC" en su nombre, ordenados por título.

4.6. Crea una tabla "CopiadeArticulos", con los mismos campos que la tabla Artículos. Usa la orden "INSERT INTO CopiadeArticulos (SELECT * FROM articulos)" para volcar a la nueva tabla todos los datos que existían en la antigua.

4.7. Borra de CopiadeArticulos aquellos artículos que comiencen en páginas por encima de la 120. Muestra los nombres de artículos existentes y su página inicial, ordenados por número de página inicial, para comprobar que el borrado es correcto.

4.8. Borra de CopiadeArticulos aquellos artículos que aparezcan en revistas Byte. Muestra los nombres de artículos existentes y el nombre de la revista a la que pertenecen, ordenados por revista y luego por título de artículo, para comprobar que el borrado es correcto.

4.9. Borra la tabla CopiadeArticulos y comprueba que ya no aparece en el sistema.

Tema 5. Modificar información

5.1 Modificación de datos

Ya sabemos borrar datos, pero existe una operación más frecuente que esa (aunque también ligeramente más complicada): modificar los datos existentes. Con lo que sabíamos hasta ahora, podíamos conseguir modificar información: si un dato es incorrecto, podríamos borrarlo y volver a introducirlo, pero esto, obviamente, no es lo más razonable... debería existir alguna orden para cambiar los datos. Efectivamente, la hay. El formato habitual para modificar datos de una tabla es "UPDATE tabla SET campo=nuevoValor WHERE condicion".

Por ejemplo, si hemos escrito "Alberto" en minúsculas ("alberto"), lo podríamos corregir con:

```
UPDATE personas SET nombre = 'Alberto' WHERE nombre = 'alberto';
```

Y si queremos corregir todas las edades para sumarles un año se haría con

```
UPDATE personas SET edad = edad+1;
```

(al igual que habíamos visto para "select" y para "delete", si no indicamos la parte del "where", como en este último ejemplo, los cambios se aplicarán a todos los registros de la tabla).

5.2. Ejercicios propuestos sobre modificación de datos

(Los tres primeros ejercicios son idénticos -salvo por el nombre de la base de datos- a los del apartado anterior, así que puedes volver a hacerlos o bien ampliar la estructura que habías creado en el apartado 4).

5.2.1. Crea una base de datos llamada "ejercicio5". En ella guardaremos información de artículos de revistas. De cada revista almacenaremos el nombre, el mes y el año, junto con un código de no más de 8 letras. Para cada artículo anotaremos un código, el título, la revista en la que aparece, la página inicial y la página final (se trata de una relación 1:M, ya que cada revista puede contener varios artículos y cada artículo sólo aparecerá en una revista). Diseña el diagrama Entidad-Relación y crea las tablas.

5.2.2. Añade la revista "Byte 9", del mes 10 de 1984, con código "BY009". Añade también la revista "PcWorld España 195", del mes 2 de 2003, con código "PCWE195".

5.2.3. Incluye también los artículos:

- "The IBM PC AT", con código "AT", en la revista Byte 9, de la página 108 a la 111.
- "Database Types", con código "DbTypes", en la revista Byte 9, de la página 138 a la 142.
- "12 Distribuciones Linux", con código "DistLinux", en la revista PCWE195 , de la página 96 a la 109.

5.2.4. Muestra todos los datos: nombre de revista, mes, año, nombre de artículo, página inicial. Deben aparecer ordenados por nombre de artículo.

5.2.5. Corrige el artículo "Database Types" ("DbTypes"): no comienza en la página 138 sino en la 137.

5.2.6. Cambia el nombre del artículo "12 Distribuciones Linux" para que pase a ser "12 Distribuciones GNU/Linux".

5.3. Modificar la estructura de una tabla

Modificar la estructura de una tabla es algo más complicado: añadir campos, eliminarlos, cambiar su nombre o el tipo de datos. En general, para todo ello se usará la orden "ALTER TABLE". Vamos a ver las posibilidades más habituales.

Para añadir un campo usaríamos "ADD":

```
ALTER TABLE ciudades ADD habitantes decimal(7);
```

Si no se indica otra cosa, el nuevo campo se añade al final de la tabla. Si queremos que sea el primer campo, lo indicaríamos añadiendo "first" al final de la orden. También podemos hacer que se añada después de un cierto campo, con "AFTER nombreCampo".

Podemos modificar el tipo de datos de un campo con "MODIFY". Por ejemplo, podríamos hacer que el campo "habitantes" no fuera un "decimal" sino un entero largo ("bigint") con:

```
ALTER TABLE ciudades MODIFY habitantes bigint;
```

Si queremos cambiar el nombre de un campo, debemos usar "CHANGE" (se debe indicar el nombre antiguo, el nombre nuevo y el tipo de datos). Por ejemplo, podríamos cambiar el nombre "habitantes" por "numhabitantes":

```
ALTER TABLE ciudades CHANGE habitantes numhabitantes bigint;
```

Si queremos borrar algún campo, usaremos "drop column":

```
ALTER TABLE ciudades DROP COLUMN numhabitantes;
```

Muchas de estas órdenes se pueden encadenar, separadas por comas. Por ejemplo, podríamos borrar dos campos con "alter table ciudades drop column num habitantes, drop column provincia;"

También podríamos cambiar el nombre de una tabla con "RENAME":

```
ALTER TABLE ciudades RENAME ciudad;
```

Y si hemos olvidado indicar la clave primaria en una tabla, también podemos usar ALTER TABLE para detallarla a posteriori:

```
ALTER TABLE ciudades ADD PRIMARY KEY(codigo);
```

5.4. Ejercicios propuestos sobre modificación de estructura de tablas

5.7. En la base de datos llamada "ejercicio5" (o en "ejercicio4", si estás partiendo de la base de datos anterior), amplía la tabla Revista añadiendo al final de todos los campos un campo adicional de tipo texto: el país el que se edita (por ejemplo, "España" o "Estados Unidos").

5.8. En la tabla de Artículos, añade el campo Autor (texto), antes del número de página inicial.

Tema 6. Operaciones matemáticas

6.1 Operaciones matemáticas

Desde SQL podemos realizar operaciones a partir de los datos antes de mostrarlos. Por ejemplo, podemos mostrar cual era la edad de una persona hace un año, con

```
SELECT edad-1 FROM personas;
```

Los operadores matemáticos que podemos emplear son los habituales en cualquier lenguaje de programación, ligeramente ampliados: + (suma), - (resta y negación), * (multiplicación), / (división). La división calcula el resultado con decimales; si queremos trabajar con números enteros, también tenemos los operadores DIV (división entera) y MOD (resto de la división):

```
SELECT 5/2, 5 div 2, 5 mod 2;
```

Daría como resultado

```
+--------+---------+---------+
| 5/2    | 5 div 2 | 5 mod 2 |
+--------+---------+---------+
| 2.5000 |       2 |       1 |
+--------+---------+---------+
```

Para que resulte más legible, se puede añadir un "alias" a cada "nuevo campo" que se genera al plantear las operaciones matemáticas:

```
SELECT 5/2 divReal, 5 div 2 divEnt, 5 mod 2 resto;
+---------+--------+-------+
| divReal | divEnt | resto |
+---------+--------+-------+
|  2.5000 |      2 |     1 |
+---------+--------+-------+
```

También podríamos utilizar incluso operaciones a nivel de bits, como las del lenguaje C (>> para desplazar los bits varias posiciones a la derecha, << para desplazar a la izquierda, | para una suma lógica bit a bit, & para un producto lógico bit a bit y ^ para una operación XOR):

```
SELECT 25 >> 1, 25 << 1, 25 | 10, 25 & 10, 25 ^ 10;
```

que daría

```
+---------+---------+---------+---------+---------+
| 25 >> 1 | 25 << 1 | 25 | 10 | 25 & 10 | 25 ^ 10 |
+---------+---------+---------+---------+---------+
|      12 |      50 |      27 |       8 |      19 |
+---------+---------+---------+---------+---------+
```

6.2 Funciones de agregación

También podemos aplicar ciertas funciones matemáticas a todo un conjunto de datos de una tabla. Por ejemplo, podemos saber cuál es la edad más baja de entre las personas que tenemos en nuestra base de datos, haríamos:

```
SELECT min(edad) FROM personas;
```

Las funciones de agregación más habituales son:

- min = mínimo valor
- max = máximo valor
- sum = suma de los valores
- avg = media de los valores
- count = cantidad de valores

La forma más habitual de usar "count" es pidiendo con "count(*)" que se nos muestren todos los datos que cumplen una condición. Por ejemplo, podríamos saber cuántas personas tienen una dirección que comience por la letra "s", así:

```sql
SELECT count(*) FROM personas WHERE direccion LIKE 's%';
```

6.3. Ejercicios propuestos

(Estos ejercicios parten de la base de datos "ejercicio5" o "ejercicio4").

6.1. Muestra el nombre de cada artículo y la cantidad de páginas que ocupa (página final - página inicial + 1; por ejemplo, un artículo que empiece en la página 3 y acabe en la 4 ocupa dos páginas).

6.2. Muestra la cantidad de artículos que hay en la base de datos.

6.3. Muestra la cantidad de revistas que contienen "byte" en su nombre.

6.4. Muestra la media del año de publicación de las revistas de la base de datos.

6.5. Muestra el año de publicación de la revista más antigua de la base de datos (el mínimo de los años) y el de la más moderna (el máximo).

6.6. Muestra el total de páginas que tenemos indexadas (mira la pregunta 6.1 para ver una pista de cómo calcular las páginas de cada artículo).

Tema 7. Valores nulos

7.1 Cero y valor nulo

En ocasiones querremos dejar un campo totalmente vacío, sin valor.

Para las cadenas de texto, existe una forma "parecida" de conseguirlo, que es con una cadena vacía, indicada con dos comillas que no contengan ningún texto entre ellas (ni siquiera espacios en blanco): ''

En cambio, para los números, no basta con guardar un 0 para indicar que no se sabe el valor: no es lo mismo un importe de 0 euros que un importe no detallado. Por eso, existe un símbolo especial para indicar cuando no existe valor en un campo (tanto en valores numéricos como en textos).

Este símbolo especial es la palabra NULL. Por ejemplo, añadiríamos datos parcialmente en blanco a una tabla haciendo

```
INSERT INTO personas
 (nombre, direccion, edad)
VALUES (
 'pedro', '', NULL
);
```

En el ejemplo anterior, y para que sea más fácil comparar las dos alternativas, he conservado las comillas sin contenido para indicar una dirección vacía, y he usado NULL para la edad, pero sería más correcto usar NULL en ambos casos para indicar que no existe valor, así:

```
INSERT INTO personas
 (nombre, direccion, edad)
VALUES (
 'pedro', NULL, NULL
);
```

Para saber si algún campo está vacío, compararíamos su valor con NULL, pero de una forma un tanto especial: no con el símbolo "igual" (=), sino con la palabra IS. Por ejemplo, sabríamos cuales de las personas de nuestra base de datos tienen dirección usando

```
SELECT * FROM personas
  WHERE direccion IS NOT NULL;
```

Y, de forma similar, sabríamos quien no tiene dirección, así:

```
SELECT * FROM personas
  WHERE direccion IS NULL;
```

En el momento de crear la tabla, a no ser que se indique lo contrario, los valores de cualquier campo podrán ser nulos, excepto para las claves primarias. Si se desea que algún campo no pueda tener valores nulos, se puede añadir NOT NULL en el momento de definir la tabla, así:

```
CREATE TABLE ciudades (
  codigo varchar(3) PRIMARY KEY,
  nombre varchar(30) NOT NULL
);
```

7.2. Ejercicios propuestos

(Estos ejercicios también parten de la base de datos "ejercicio5" o "ejercicio4").

7.1. Crea una base de datos "ejercicio7", con una única tabla "ciudades". Cada ciudad tendrá un código (clave primaria), un nombre (no nulo) y una cantidad de habitantes (que sí podrá tener valores nulos).

7.2. Añade las ciudades: Boston (código BO, 4.180.000 habitantes) y Kyoto (código KY, sin indicar el número de habitantes).

7.3. Intenta añadir la ciudad Astana, sin código.

7.4. Intenta añadir la ciudad de código ELX pero de la que aún no sabemos el nombre.

7.5. Muestra el nombre y población de todas las ciudades para las que sabemos la cantidad de habitantes, ordenadas de la menos poblada a la más poblada.

7.6. Muestra el nombre de las ciudades que tienen 0 habitantes, ordenadas alfabéticamente.

7.7. Muestra el nombre de las ciudades para las que no conocemos la cantidad de habitantes, ordenadas alfabéticamente.

Tema 8. Valores agrupados

8.1 Agrupando los resultados

Puede ocurrir que no nos interese un único valor agrupado para todos los datos (el total, la media, la cantidad de datos), sino el resultado para un grupo específico de datos. Por ejemplo: podemos desear saber no sólo la cantidad de clientes que hay registrados en nuestra base de datos, sino también la cantidad de clientes que viven en cada ciudad.

La forma de obtener subtotales es creando grupos con la orden "GROUP BY", y entonces pidiendo un valor agrupado (count, sum, avg, ...) para cada uno de esos grupos. Como limitación, sólo se podrán pedir como resultados los valores agrupados (como la cantidad o la media) y el criterio de agrupamiento.

Por ejemplo, en nuestra tabla "personas", podríamos saber cuántas personas aparecen de cada edad, con:

```
SELECT count(*), edad FROM personas GROUP BY edad;
```

que daría como resultado

```
+----------+------+
| count(*) | edad |
+----------+------+
|        1 |   22 |
|        1 |   23 |
|        1 |   25 |
+----------+------+
```

8.2 Filtrando los datos agrupados

Pero podemos llegar más allá: podemos no trabajar con todos los grupos posibles, sino sólo con los que cumplen alguna condición.

La condición que se aplica a los grupos no se indica con "where", sino con "having" (que se podría traducir como "los que tengan..."). Un ejemplo:

```
SELECT count(*), edad FROM personas GROUP BY edad HAVING edad > 24;
```

que mostraría

```
+----------+------+
| count(*) | edad |
+----------+------+
|        1 |   25 |
+----------+------+
```

8.3. Ejercicios propuestos

8.1. Crea una base de datos "ejercicio8", con una única tabla "ordenadores". De cada ordenador se desea guardar un código (que será la clave primaria), una marca (no nula), un modelo y un año de lanzamiento (mira los datos de ejemplo para deducir los tipos de datos necesarios).

8.2. Añade los ordenadores:

- IBM5150, IBM, PC (5150), 1981
- SPEC48, Sinclair, ZX Spectrum 48K, 1982
- CPC464, Amstrad, CPC464, 1984
- HB55, Sony, Hit-Bit 55 MSX, 1984
- QL, Sinclair, QL, 1984
- PPC640DD, Amstrad, PPC640 DD, 1988

8.3. Muestra la cantidad total de ordenadores que tenemos registrados en nuestra base de datos.

8.4. Muestra la cantidad de ordenadores de cada marca.

8.5. Muestra la cantidad de ordenadores lanzados en 1984 o más tarde.

8.6. Muestra la cantidad de ordenadores por cada marca, pero teniendo en cuenta sólo lanzados en 1984 o más tarde.

8.7. Muestra la cantidad de ordenadores de cada marca, pero sólo para las marcas de las que tengamos 2 o más equipos.

8.8. Como verás con más detalle en el apartado 10, la función SUBSTRING permite obtener una subcadena. Por ejemplo, SUBSTRING(modelo,1,2) permitiría saber las dos primeras letras del modelo de un ordenador. Usando esta función, deberás mostrar la cantidad de ordenadores que tenemos cuya marca empieza con una letra "S".

8.9. Muestra la cantidad de ordenadores, agrupados por la inicial de su marca.

8.10. Muestra el nombre (modelo) del ordenador más moderno del que tenemos constancia para cada marca.

Tema 9. Subconsultas

9.1. ¿Qué es una subconsulta?

A veces tenemos que realizar operaciones más complejas con los datos, operaciones en las que nos interesaría ayudarnos de una primera consulta auxiliar que extrajera la información en la que nos queremos basar. Esta consulta auxiliar recibe el nombre de "subconsulta" o "subquery".

Por ejemplo, si queremos saber qué clientes tenemos en la ciudad que más habitantes tenga, la forma "razonable" de conseguirlo sería saber en primer lugar cual es la ciudad que más habitantes tenga, y entonces lanzar una segunda consulta para ver qué clientes hay en esa ciudad.

Como la estructura de nuestra base de datos de ejemplo es muy sencilla, no podemos hacer grandes cosas, pero un caso parecido al anterior (aunque claramente más inútil) podría ser saber qué personas tenemos almacenadas que vivan en la última ciudad de nuestra lista.

Para ello, la primera consulta (la "subconsulta") sería saber cuál es la última ciudad de nuestra lista. Si suponemos que los códigos numéricos sob creciente, y lo hacemos tomando la ciudad que tenga el último código (el mayor de ellos), la consulta podría ser:

```
SELECT MAX(codigo) FROM ciudades;
```

Vamos a imaginar que pudiéramos hacerlo en dos pasos. Si llamamos "maxCodigo" a ese código obtenido, la "segunda" consulta podría ser:

```
SELECT * FROM personas WHERE codciudad= maxCodigo;
```

Pero estos dos pasos se pueden dar en uno: al final de la "segunda" consulta (la "grande") incluimos la primera consulta (la "subconsulta"), entre paréntesis, así

```
SELECT * FROM personas WHERE codciudad= (
  SELECT MAX(codigo) FROM ciudades
);
```

9.2. Subconsultas que devuelven conjuntos de datos

Si la subconsulta no devuelve un único dato, sino un conjunto de datos, la forma de trabajar será básicamente la misma, pero para comprobar si el valor coincide con alguno de la lista, no usaremos el símbolo "=", sino la palabra "in".

Por ejemplo, vamos a hacer una consulta que nos muestre las personas que viven en ciudades cuyo nombre tiene una "a" en segundo lugar (por ejemplo, serían ciudades válidas Madrid o Barcelona, pero no Alicante).

Para consultar qué letras hay en ciertas posiciones de una cadena, podemos usar SUBSTRING (en el próximo apartado veremos las funciones más importantes de manipulación de cadenas, pero podemos anticipar ya que recibe tres datos: el nombre del campo, la posición inicial y la longitud). Así, una forma de saber qué ciudades tienen una letra A en su segunda posición sería:

```
SELECT codigo FROM ciudades
WHERE SUBSTRING(nombre,2,1)='a';
```

Como esta subconsulta puede devolver más de un resultado, deberemos usar IN para incluirla en la consulta principal, que quedaría de esta forma:

```
SELECT * FROM personas
WHERE codciudad IN
(
  SELECT codigo FROM ciudades WHERE SUBSTRING(nombre,2,1)='a'
);
```

9.3. Ejercicios propuestos

9.1. Partiremos de la base de datos "ejercicio8", que creaste en el ejercicio 8.1. Si no lo habías hecho, crea ahora una nueva base de datos "ejercicio9", que tendrá una única tabla "ordenadores". De cada ordenador se desea guardar un código (que será la clave primaria), una marca (no nula), un modelo y un año de lanzamiento (mira los datos de ejemplo para deducir los tipos de datos necesarios).
Añade los ordenadores:

- ATM, Oric, Atmos, 1984
- ZX80, Sinclair, ZX80, 1980
- VIC20, Commodore, VIC-20, 1981

- VG8235, Philips, VG8235 MSX2, 1985
- C64, Commodore, 64, 1982
- 520ST, Atari, 520ST, 1985

9.3. Muestra la(s) marca(s) de la(s) que tenemos más ordenadores.

9.4. Muestra los modelos de los ordenadores pertenecientes a la(s) marca(s) de la(s) que tenemos más ordenadores.

9.5. Muestra los modelos de los ordenadores pertenecientes a la(s) marca(s) a la que pertenece el ordenador más reciente, ordenados alfabéticamente (usando el modelo como criterio de ordenación).

9.6. Muestra todos los datos de los ordenadores de la primera marca (alfabéticamente, ordenadas de la 'A' a la 'Z').

9.7. Muestra los modelos de los ordenadores que no pertenezcan a las marcas que comienzan con letra A, usando una subconsulta.

Tema 10. Funciones de cadena

En MySQL tenemos muchas funciones para manipular cadenas: calcular su longitud, extraer un fragmento situado a la derecha, a la izquierda o en cualquier posición, eliminar espacios finales o iniciales, convertir a hexadecimal y a binario, etc. Vamos a comentar las más habituales. Los ejemplos estarán aplicados directamente sobre cadenas, pero (por supuesto) también se pueden aplicar a campos de una tabla:

10.1. Funciones de conversión a mayúsculas/minúsculas

- LOWER o LCASE convierte una cadena a minúsculas: SELECT LOWER('Hola'); ⇒ hola
- UPPER o UCASE convierte una cadena a mayúsculas: SELECT UPPER('Hola'); ⇒ HOLA

10.2. Funciones de extracción de parte de la cadena

- LEFT(cadena, longitud) extrae varios caracteres del comienzo (la parte izquierda) de la cadena: SELECT LEFT('Hola',2); ⇒ Ho
- RIGHT(cadena, longitud) extrae varios caracteres del final (la parte derecha) de la cadena: SELECT RIGHT('Hola',2); ⇒ la
- MID(cadena, posición, longitud), SUBSTR(cadena, posición, longitud) o SUBSTRING(cadena, posición, longitud) extrae varios caracteres de cualquier posición de una cadena, tantos como se indique en "longitud": SELECT SUBSTRING('Hola',2,3); ⇒ ola (Nota: a partir MySQL 5 se permite un valor negativo en la posición, y entonces se comienza a contar desde la derecha -el final de la cadena-)
- CONCAT une (concatena) varias cadenas para formar una nueva: SELECT CONCAT('Ho', 'la'); ⇒ Hola
- CONCAT_WS une (concatena) varias cadenas para formar una nueva, usando un separador que se indique (With Separator): SELECT CONCAT_WS('-','Ho','la','Que','tal'); ⇒ Ho-la-Que-tal
- LTRIM devuelve la cadena sin los espacios en blanco que pudiera contener al principio (en su parte izquierda): SELECT LTRIM(' Hola'); ⇒ Hola
- RTRIM devuelve la cadena sin los espacios en blanco que pudiera contener al final (en su parte derecha): SELECT RTRIM('Hola '); ⇒ Hola
- TRIM devuelve la cadena sin los espacios en blanco que pudiera contener al principio ni al final: SELECT TRIM(' Hola '); ⇒ Hola (Nota: realmente, TRIM puede eliminar cualquier prefijo, no sólo espacios; mira el manual de MySQL para más detalles)

10.3. Funciones de conversión de base numérica

- BIN convierte un número decimal a binario: SELECT BIN(10); ⇒ 1010
- HEX convierte un número decimal a hexadecimal: SELECT HEX(10); ⇒ 'A' (Nota: HEX también tiene un uso alternativo menos habitual: puede recibir una cadena, y entonces mostrará el código ASCII en hexadecimal de sus caracteres: SELECT HEX('Hola'); ⇒ '486F6C61')
- OCT convierte un número decimal a octal: SELECT OCT(10); ⇒ 12
- CONV(número,baseInicial,baseFinal) convierte de cualquier base a cualquier base: SELECT CONV('F3',16,2); ⇒ 11110011
- UNHEX convierte una serie de números hexadecimales a una cadena ASCII, al contrario de lo que hace HEX: SELECT UNHEX('486F6C61'); ⇒ 'Hola')

10.4. Otras funciones de modificación de la cadena

- INSERT(cadena,posición,longitud,nuevaCadena) inserta en la cadena otra cadena: SELECT INSERT('Hola', 2, 2, 'ADIOS'); ⇒ HADIOSa
- REPLACE(cadena,de,a) devuelve la cadena pero cambiando ciertas secuencias de caracteres por otras: SELECT REPLACE('Hola', 'l', 'LLL'); ⇒ HoLLLa
- REPEAT(cadena,numero) devuelve la cadena repetida varias veces: SELECT REPEAT(' Hola',3); ⇒ HolaHolaHola
- REVERSE(cadena) devuelve la cadena "del revés": SELECT REVERSE('Hola'); ⇒ aloH
- SPACE(longitud) devuelve una cadena formada por varios espacios en blanco: SELECT SPACE(3); ⇒ " "

10.5. Funciones de información sobre la cadena

- CHAR_LENGTH o CHARACTER_LENGTH devuelve la longitud de la cadena en caracteres
- LENGTH devuelve la longitud de la cadena en bytes
- BIT_LENGTH devuelve la longitud de la cadena en bits
- INSTR(cadena,subcadena) o LOCATE(subcadena,cadena,posInicial) devuelve la posición de una subcadena dentro de la cadena: SELECT INSTR('Hola','ol'); ⇒ 2

10.6. Ejercicios propuestos

10.1. Crea una base de datos "ejercicio10", en la que guardaremos información sobre selecciones nacionales de baloncesto. Para ello tendremos: una tabla "PAISES" y una tabla "JUGADORES", unidas por una relación 1:M (cada país podrá tener muchos jugadores y cada jugador sólo podrá formar parte -en un instante dado- de la selección de un país). De cada país guardaremos el nombre (por ejemplo, "España") y un código

que actuará como clave primaria (por ejemplo, "ESP). De cada jugador anotaremos código, nombre, apellidos, posición y, como resultado de esa relación 1:M, código de la selección a la que pertenece.

10.2a. Añade los países:

- ESP, España
- ARG, Argentina
- AUS, Australia
- LIT, Lituania

10.2b. Añade los jugadores:

- RUB, Ricky, Rubio, Base (España)
- NAV, Juan Carlos, Navarro, Alero (España)
- SCO, Luis, Scola, Ala-Pivot (Argentina)
- DEL, Carlos, Delfino, Escolta (Argentina)
- MAC, Jonas, Maciulis, Alero (Lituania)
- BOG, Andrew, Bogut, Pivot (Australia)

10.3. Muestra los nombres y apellidos de todos los jugadores, en mayúsculas, ordenados por apellido y nombre.

10.4. Muestra el nombre y apellidos del jugador o jugadores cuyo apellido es el más largo (formado por más letras).

10.5. Muestra el apellido, una coma, un espacio y después el nombre de todos los jugadores de "España" (aparecerán datos como "Rubio, Ricky"). Para ello, usa la función "CONCAT". Los resultados deben aparecer como si se tratase de un campo llamado "nombreJug".

10.6. Muestra las 4 primeras letras de los apellidos de los jugaodres que tenemos anotados de "Argentina", ordenados de forma descendente.

10.7. Muestra los nombres de todos los jugadores, reemplazando "Ricky" por "Ricard".

10.8. Muestra "Don " seguido del nombre y del apellido de los jugadores (aparecerán datos como "Don Andrew Bogut"), usando "CONCAT" e "INSERT" para crar al vuelo un nuevo campo llamado "nombreJug".

10.9. Muestra el nombre y apellidos de todos los jugadores cuyo país contenga una N en el nombre. Debes eliminar los espacios iniciales y finales de ambos campos, en caso de que existan.

10.10. Muestra al revés el apellido de los jugadores de Australia que tenemos en nuestra base de datos.

10.11. Muestra una cadena formada por 10 guiones, 10 espacios y otros 10 guiones.

Tema 11. Join

11.1. Acercamiento a la necesidad de los JOIN

Sabemos enlazar varias tablas para mostrar datos que estén relacionados, empleando "WHERE". Por ejemplo, podríamos mostrar nombres de deportistas, junto con los nombres de los deportes que practican. Pero todavía hay un detalle que se nos escapa: ¿cómo hacemos si queremos mostrar todos los deportes que hay en nuestra base de datos, incluso aunque no haya deportistas que los practiquen?

Vamos a crear una base de datos sencilla para ver un ejemplo de cual es este problema y de cómo solucionarlo.

Nuestra base de datos se llamará "ejemploJoins":

```
CREATE DATABASE ejemploJoins;
USE ejemploJoins;
```

En ella vamos a crear una primera tabla en la que guardaremos "capacidades" de personas (cosas que saben hacer):

```
CREATE TABLE capacidad(
 codigo varchar(4),
 nombre varchar(20),
 PRIMARY KEY(codigo)
);
```

También crearemos una segunda tabla con datos básicos de personas:

```
CREATE TABLE persona(
 codigo varchar(4),
 nombre varchar(20),
 codcapac varchar(4),
 PRIMARY KEY(codigo)
);
```

Vamos a introducir datos de ejemplo:

```
INSERT INTO capacidad VALUES
 ('c','Progr.C'),
 ('pas','Progr.Pascal'),
 ('j','Progr.Java'),
 ('sql','Bases datos SQL');

INSERT INTO persona VALUES
 ('ju','Juan','c'),
 ('ja','Javier','pas'),
 ('jo','Jose','perl'),
 ('je','Jesus','html');
```

Antes de seguir, comprobamos que todo está bien:

```
SELECT * FROM capacidad;
+--------+-----------------+
| codigo | nombre          |
+--------+-----------------+
| c      | Progr.C         |
| j      | Progr.Java      |
| pas    | Progr.Pascal    |
| sql    | Bases datos SQL |
+--------+-----------------+
SELECT * FROM persona;
+--------+--------+----------+
| codigo | nombre | codcapac |
+--------+--------+----------+
| ja     | Javier | pas      |
| je     | Jesus  | html     |
| jo     | Jose   | perl     |
| ju     | Juan   | c        |
+--------+--------+----------+
```

Como se puede observar, hay dos capacidades en nuestra base de datos para las que no conocemos a ninguna persona; de igual modo, existen dos personas que tienen capacidades sobre las que no tenemos ningún detalle.

Por eso, si mostramos las personas con sus capacidades de la forma que sabemos, sólo aparecerán las parejas de persona y capacidad para las que todo está claro (existe persona y existe capacidad), es decir:

```
SELECT * FROM capacidad, persona
WHERE persona.codcapac = capacidad.codigo;
+--------+-------------+--------+--------+----------+
| codigo | nombre      | codigo | nombre | codcapac |
+--------+-------------+--------+--------+----------+
| c      | Progr.C     | ju     | Juan   | c        |
| pas    | Progr.Pascal | ja    | Javier | pas      |
+--------+-------------+--------+--------+----------+
```

Podemos resumir un poco esta consulta, para mostrar sólo los nombres, que son los datos que más nos interesan:

```
SELECT persona.nombre, capacidad.nombre
FROM persona, capacidad
WHERE persona.codcapac = capacidad.codigo;
+--------+-------------+
| nombre | nombre      |
+--------+-------------+
| Juan   | Progr.C     |
| Javier | Progr.Pascal |
+--------+-------------+
```

Hay que recordar que la orden "where" es **obligatoria**: si no indicamos esa condición, se mostraría el "**producto cartesiano**" de las dos tablas: todos los pares (persona, capacidad), aunque no estén relacionados en nuestra base de datos:

```
SELECT persona.nombre, capacidad.nombre
FROM persona, capacidad;
+--------+----------------+
| nombre | nombre         |
```

```
+--------+-----------------+
| Javier | Progr.C         |
| Jesus  | Progr.C         |
| Jose   | Progr.C         |
| Juan   | Progr.C         |
| Javier | Progr.Java      |
| Jesus  | Progr.Java      |
| Jose   | Progr.Java      |
| Juan   | Progr.Java      |
| Javier | Progr.Pascal    |
| Jesus  | Progr.Pascal    |
| Jose   | Progr.Pascal    |
| Juan   | Progr.Pascal    |
| Javier | Bases datos SQL |
| Jesus  | Bases datos SQL |
| Jose   | Bases datos SQL |
| Juan   | Bases datos SQL |
+--------+-----------------+
```

11.2. JOIN cruzados, internos y externos

Pues bien, con órdenes "join" podemos afinar cómo queremos enlazar (en inglés, "join", unir) las tablas. Por ejemplo, si queremos ver todas las personas y todas las capacidades, aunque no estén relacionadas, como en el ejemplo anterior, algo que **no suele tener sentido** en la práctica, lo podríamos hacer con un "**cross join**" (unir de forma cruzada):

```
SELECT persona.nombre, capacidad.nombre
FROM persona CROSS JOIN capacidad;
+--------+-----------------+
| nombre | nombre          |
+--------+-----------------+
| Javier | Progr.C         |
| Jesus  | Progr.C         |
| Jose   | Progr.C         |
| Juan   | Progr.C         |
| Javier | Progr.Java      |
```

```
| Jesus  | Progr.Java      |
| Jose   | Progr.Java      |
| Juan   | Progr.Java      |
| Javier | Progr.Pascal    |
| Jesus  | Progr.Pascal    |
| Jose   | Progr.Pascal    |
| Juan   | Progr.Pascal    |
| Javier | Bases datos SQL |
| Jesus  | Bases datos SQL |
| Jose   | Bases datos SQL |
| Juan   | Bases datos SQL |
+--------+-----------------+
```

Si sólo queremos ver los datos que coinciden en ambas tablas, lo que antes conseguíamos comparando los códigos con un "where", también podemos usar un "**inner join**" (unión interior; se puede abreviar simplemente "join"):

```
SELECT persona.nombre, capacidad.nombre
FROM persona INNER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
+--------+--------------+
| nombre | nombre       |
+--------+--------------+
| Juan   | Progr.C      |
| Javier | Progr.Pascal |
+--------+--------------+
```

Pero aquí llega la novedad: si queremos ver todas las personas y sus capacidades (si existen), mostrando incluso aquellas personas para las cuales no tenemos constancia de ninguna capacidad, usaríamos un "**left join**" (unión por la izquierda, también se puede escribir "left outer join", unión exterior por la izquierda, para dejar claro que se van a incluir datos que están sólo en una de las dos tablas):

```
SELECT persona.nombre, capacidad.nombre
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
+--------+--------------+
| nombre | nombre       |
```

```
+--------+--------------+
| Juan   | Progr.C      |
| Javier | Progr.Pascal |
| Jesus  | NULL         |
| Jose   | NULL         |
+--------+--------------+
```

De igual modo, si queremos ver todas las capacidades, incluso aquellas para las que no hay detalles sobre personas, podemos escribir el orden de las tablas al revés en la consulta anterior, o bien usar "**right join**" (o "right outer join"):

```
SELECT persona.nombre, capacidad.nombre
FROM persona RIGHT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
+--------+-----------------+
| nombre | nombre          |
+--------+-----------------+
| Javier | Progr.Pascal    |
| Juan   | Progr.C         |
| NULL   | Progr.Java      |
| NULL   | Bases datos SQL |
+--------+-----------------+
```

El significado de "LEFT" y de "RIGHT" hay que buscarlo en la posición en la que se enumeran las tablas en el bloque "FROM". Así, la última consulta se puede escribir también como un LEFT JOIN si se indica la capacidad en segundo lugar (en la parte izquierda), así:

```
SELECT persona.nombre, capacidad.nombre
FROM capacidad LEFT OUTER JOIN persona
ON persona.codcapac = capacidad.codigo;
+--------+-----------------+
| nombre | nombre          |
+--------+-----------------+
| Javier | Progr.Pascal    |
| Juan   | Progr.C         |
| NULL   | Progr.Java      |
| NULL   | Bases datos SQL |
```

```
+--------+----------------+
```

Otros gestores de bases de datos permiten combinar el "right join" y el "left join" en una única consulta, usando "full outer join", algo que no permite MySQL en su versión actual, pero que se puede imitar de una forma que veremos en el próximo apartado.

11.3. Ejercicios propuestos

11.1. Crea una base de datos "ejercicio11", en la que guardaremos información sobre deportistas y disciplinas deportivas. Tendremos una tabla "DEPORTES" y una tabla "DEPORTISTAS", unidas por una relación M:M (cada deporte puede ser particado por más de un deportista y cada deportista podría practicar más de un deporte). Además, existirá una tabla "PAISES". Para cada país guardaremos el nombre (por ejemplo, "España") y un código que actuará como clave primaria (por ejemplo, "ESP). De cada deportista anotaremos código, nombre, apellidos. Para cada deporte, nos interesará código y nombre. La relación M:M se reflejará en una nueva tabla "PRACTICAR", cada uno de cuyos registros estará formado por el código de deporte y el código de deportista (ambos formarán la clave primaria -compuesta- de esta nueva tabla). La relación 1:M entre deportista y país se reflejará en que de cada deportista deberá se deberá anotar también el código del país al que representa.

11.2a. Añade los países:

- JAM, Jamaica
- ESP, España
- USA, Estados Unidos de América
- AUS, Australia
- RUS, Rusia

11.2b. Añade los deportistas:

- BOL, Usain, Bolt (Jamaica)
- POW, Asafa, Powell (Jamaica)
- CRA, Saúl, Craviotto (España)
- TAU, Diana, Taurasi (Estados Unidos)
- PHE, Michael, Phelps (Estados Unidos)
- MUR, Andy, Murray (UK)

11.2c. Añade los deportes:

- ATL, Atletismo
- REM, Remo
- BAL, Baloncesto
- NAT, Natación
- BAD, Badminton

11.2d. Y la relación Practicar:

- Usain Bolt, Velocidad
- Saúl Craviotto, Remo
- Diana Taurasi, Baloncesto
- Michael Phelps, Natación

11.3. Muestra los nombres y apellidos de todos los deportistas, junto al nombre del país al que pertenecen.

11.4. Muestra los nombres y apellidos de todos los deportistas, junto al nombre del país al que pertenecen, en caso de que dicho país aparezca en la base de datos. También se deben mostrar los datos de los deportistas cuyo país no se haya introducido aún.

11.5. Muestra los nombres de todos los países, junto a los apellidos de los deportistas de ese país. Deben aparecer también los países de los que no conozcamos ningún deportista.

11.6. Muestra los nombres y apellidos de todos los deportistas, junto al nombre del deporte que practican (sólo aquellos de los que tengamos constancia que realmente practican algún deporte).

11.7. Muestra los nombres y apellidos de todos los deportistas, junto al nombre del deporte que practican (incluso los que no sepamos cuál es su deporte).

11.8. Muestra el nombre de todos los deportes, junto con los nombres y apellidos de los deportistas que lo practican (incluso si para algún deporte aún no hemos introducido ningún deportista).

Tema 12. Unión

12.1. Unión de dos consultas

Se puede unir dos consultas en una, empleando la palabra "UNION". El requisito es que ambas consultas deben devolver campos "similares" (mismo nombre y tipo de datos). Como primer ejemplo (un tanto innecesario), podríamos crear una consulta que muestre las personas cuyo nombre contiene una "o" y aquellas cuyo nombre tiene una "e":

```
SELECT nombre FROM persona
WHERE nombre LIKE '%o%'
union
SELECT nombre FROM persona
WHERE nombre LIKE '%e%';
```

que mostraría como resultado:

```
+--------+
| nombre |
+--------+
| Jose   |
| Javier |
| Jesus  |
+--------+
```

(Como has podido observar, "Jose" no aparece repetido, porque el operador "UNION" elimina duplicados).

En este caso, el uso de "UNION" es poco razonable porque se trata de dos consultas casi idénticas, que se podían haber realizado simplemente con un "OR" y se obtendrá el mismo resultado (quizá en distinto orden, ya que no hemos utilizado "ORDER BY"):

```
SELECT nombre FROM persona
WHERE nombre LIKE '%o%'
```

```
OR nombre LIKE '%e%';
+--------+
| nombre |
+--------+
| Javier |
| Jesus  |
| Jose   |
+--------+
```

La verdadera utilidad de "UNION" aparece cuando se trabaja sobre conjuntos de datos diferentes, incluso distintas tablas. Por ejemplo, podríamos obtener los nombres de todas las personas y los de todas las capacidades en una misma consulta:

```
SELECT nombre FROM persona
union
SELECT nombre FROM capacidad;
```

que mostraría:

```
+-----------------+
| nombre          |
+-----------------+
| Javier          |
| Jesus           |
| Jose            |
| Juan            |
| Progr.C         |
| Progr.Java      |
| Progr.Pascal    |
| Bases datos SQL |
+-----------------+
```

E incluso se podrían mostrar datos originalmente muy distintos, si se renombran empleando un alias:

```
SELECT concat('Persona: ', nombre) AS detalle
FROM persona
```

```
union
SELECT concat('Habilidad: ', upper(nombre)) AS detalle
FROM capacidad;
+---------------------------+
| detalle                   |
+---------------------------+
| Persona: Javier           |
| Persona: Jesus            |
| Persona: Jose             |
| Persona: Juan             |
| Habilidad: PROGR.C        |
| Habilidad: PROGR.JAVA     |
| Habilidad: PROGR.PASCAL   |
| Habilidad: BASES DATOS SQL |
+---------------------------+
```

12.2. Imitando un FULL OUTER JOIN

En el apartado anterior comentábamos que la versión actual de MySQL no permite usar "full outer join" para mostrar todos los datos que hay en dos tablas enlazadas, aunque alguno de esos datos no tenga equivalencia en la otra tabla.

También decíamos que se podría imitar haciendo a la vez un "right join" y un "left join".

En general, tenemos la posibilidad de unir dos consultas en una usando "union", así:

```
SELECT persona.nombre, capacidad.nombre
FROM persona RIGHT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo
union
SELECT persona.nombre, capacidad.nombre
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
+--------+----------------+
| nombre | nombre         |
+--------+----------------+
```

```
| Javier | Progr.Pascal    |
| Juan   | Progr.C         |
| NULL   | Progr.Java      |
| NULL   | Bases datos SQL |
| Jesus  | NULL            |
| Jose   | NULL            |
+--------+-----------------+
```

Los datos no aparecen ordenados. Si se desea que lo estén, se puede incluir la "UNION" dentro de una subconsulta (será necesario usar un "alias" para la subconsulta), así:

```
SELECT * FROM
(
SELECT persona.nombre , capacidad.nombre habilidad
FROM persona RIGHT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo
union
SELECT persona.nombre, capacidad.nombre
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo
) resultado
ORDER BY nombre;
+--------+-----------------+
| nombre | habilidad       |
+--------+-----------------+
| NULL   | Progr.Java      |
| NULL   | Bases datos SQL |
| Javier | Progr.Pascal    |
| Jesus  | NULL            |
| Jose   | NULL            |
| Juan   | Progr.C         |
+--------+-----------------+
```

(Como puedes ver, al ordenar resultados, los datos nulos aparecen antes de los que sí tienen valor).

Nota: en algunos gestores de bases de datos, podemos no sólo crear "uniones" entre dos tablas, sino también realizar otras operaciones habituales entre conjuntos, como

calcular su intersección ("intersection") o ver qué elementos hay en la primera pero no en la segunda (diferencia, "difference"). Estas posibilidades no están disponibles en la versión actual de MySQL.

12.3. Ejercicios propuestos

12.1. Partiendo de la base de datos "ejercicio11", que tenía información sobre deportistas y disciplinas deportivas: muestra en una misma consulta los nombres y apellidos de todos los deportistas, además de los nombres de las disciplinas deportivas.

12.2. Muestra los nombres y apellidos de todos los deportistas, junto al nombre del país al que pertenecen, en caso de que dicho país aparezca en la base de datos. Deben aparecer todos los países y todos los deportistas, aunque alguno pudiera estar sin relacionar (exista algún deportista del que no conozcamos el país o algún país del que no tengamos anotados aún deportistas).

12.3. Muestra los nombres y apellidos de todos los deportistas, junto al nombre de la disciplina deportiva que practican. Deben aparecer todas las disciplinas y todos los deportistas, aunque alguno pudiera estar sin relacionar.

12.4. Muestra los nombres y apellidos de todos los deportistas (en formato "Apellido, nombre"), junto al nombre de la disciplina deportiva que practican. Deben mostrarse todas las disciplinas y todos los deportistas, aunque alguno pudiera estar sin relacionar. Los datos deben aparecer ordenados según el apellido del deportista.

Tema 13. Vistas

13.1. Creación de vistas

Una consulta compleja puede ser acabar teniendo un gran tamaño, que haga que resulte difícil de leer, y que sea incómodo ampliarla aún más para añadir condiciones adicionales.

Como mejora, podemos crear "**vistas**", que nos permitan llamar de forma más breve a una consulta y, de paso, pueden ayudar a filtrar la cantidad de información a la que queremos que ciertos usuarios tengan acceso, no dándoles acceso a todas las tablas sino sólo a ciertas vistas:

```
CREATE VIEW personasycapac AS
SELECT persona.nombre nompers, capacidad.nombre nomcapac
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

Y esta "vista" se utiliza igual que si fuera una tabla:

```
SELECT * FROM personasycapac;
+---------+--------------+
| nompers | nomcapac     |
+---------+--------------+
| Javier  | Progr.Pascal |
| Jesus   | NULL         |
| Jose    | NULL         |
| Juan    | Progr.C      |
+---------+--------------+
```

De modo que ahora podemos aplicar condiciones sobre esa vista usando WHERE, así como mostrar datos ordenados o aplicar cualquier otra manipulación:

```
SELECT * FROM personasycapac
WHERE substring(nompres,3,1) = 's'
ORDER BY nompres DESC;
+---------+--------------+
| nompers | nomcapac     |
+---------+--------------+
| Javier  | Progr.Pascal |
| Jesus   | NULL         |
| Jose    | NULL         |
| Juan    | Progr.C      |
+---------+--------------+
```

Cuando una vista deje de sernos útil, podemos eliminarla con "drop view".

13.2. Ejercicios propuestos

13.1. Partiendo de la base de datos "ejercicio11", que tenía información sobre deportistas y disciplinas deportivas: crea una vista "personasydeportes" que permita obtener los nombres y apellidos de todos los deportistas, junto con el nombre del deporte que practican (este último dato aparecerá con el nombre "deporte").

13.2. Usa la vista "personasydeportes" para obtener los nombres y apellidos de los deportistas que practican deportes cuyo nombre comienza con "B".

13.3. Crea una vista "personasypaises" que permita muestre un campo "persona" (que estará formado por los apellidos, una coma, un espacio y el nombre de cada deportista) y un campo "pais", que será el nombre del país, o NULL si no se ha indicado el país.

13.4. Usa la vista "personasypaises" para obtener los nombres y apellidos de los deportistas de "España", ordenados por apellido.

Tema 14. Triggers

14.1. Contacto con los "triggers"

En MySQL (a partir de la versión 5.0.2) se permite utilizar "disparadores" (triggers), que son una serie de pasos que se pondrán en marcha cuando ocurra un cierto evento en una tabla.

Los eventos pueden ser un INSERT, un UPDATE o un DELETE de datos de la tabla, y podemos detallar si queremos que los pasos se den antes (BEFORE) del evento o después (AFTER) del evento.

Como ejemplo habitual, podríamos hacer un BEFORE INSERT para comprobar que los datos son válidos antes de guardarlos realmente en la tabla.

Pero vamos a empezar probar con un ejemplo que, aunque sea menos útil, será más fácil de aplicar.

Vamos a crear una base de datos sencilla, con sólo dos tablas. En una tabla guardaremos datos de personas, y en la otra anotaremos cuando se ha introducido cada dato. La estructura básica sería ésta:

```
CREATE DATABASE ejemplotriggers;
USE ejemplotriggers;

CREATE TABLE persona (
 codigo varchar(10),
 nombre varchar(50),
 edad decimal(3),
 PRIMARY KEY (`codigo`)
);

CREATE TABLE nuevosDatos (
 codigo varchar(10),
 cuando date,
 tipo char(1)
);
```

Para que se añada un dato en la segunda tabla cada vez que insertemos en la primera, creamos un TRIGGER que saltará con un AFTER INSERT. Para indicar los pasos que debe hacer, se usa la expresión "FOR EACH ROW" (para cada fila), así:

```
CREATE TRIGGER modificacion
  AFTER INSERT ON persona
FOR EACH ROW
  INSERT INTO nuevosDatos
  VALUES (NEW.codigo, CURRENT_DATE, 'i');
```

(Los datos que introduciremos serán: el código de la persona, la fecha actual y una letra "i" para indicar que el cambio ha sido la "inserción" de un dato nuevo).

Si ahora introducimos un dato en la tabla personas:

```
INSERT INTO persona
VALUES ('1','Juan',20);
```

La tabla de "nuevosDatos" habrá cambiado:

```
SELECT * FROM nuevosDatos;
+--------+------------+------+
| codigo | cuando     | tipo |
+--------+------------+------+
| 1      | 2007-12-05 | i    |
+--------+------------+------+
```

(Nota 1: Si en vez de monitorizar los INSERT, queremos controlar los UPDATE, el valor actual del nombre es "NEW.nombre", pero también podemos saber el valor anterior con "OLD.nombre", de modo que podríamos almacenar en una tabla todos los detalles sobre el cambio que ha hecho el usuario).

(Nota 2: Si no queremos guardar sólo la fecha actual, sino la fecha y la hora, el campo debería ser de tipo DATETIME, y sabríamos el instante actual con "NOW()"):

```
CREATE TRIGGER modificacion
  AFTER INSERT ON persona
```

```
FOR EACH ROW
  INSERT INTO nuevosDatos
  VALUES (NEW.codigo, NOW(), 'i');
```

Si queremos indicar que se deben dar secuencias de pasos más largas, deberemos tener en cuenta dos cosas: cuando sean varias órdenes, deberán encerrarse entre BEGIN y END; además, como cada una de ellas terminará en punto y coma, deberemos cambiar momentáneamente el "delimitador" (DELIMITER) de MySQL, para que no piense que hemos terminado en cuanto aparezca el primer punto y coma:

```
DELIMITER |

CREATE TRIGGER validacionPersona BEFORE INSERT ON persona
FOR EACH ROW BEGIN
  SET NEW.codigo = UPPER(NEW.codigo);
  SET NEW.edad = IF(NEW.edad = 0, NULL, NEW.edad);
END;
|
DELIMITER ;
```

(Nota 3: Ese nuevo delimitador puede ser casi cualquiera, siempre y cuando no se algo que aparezca en una orden habitual. Hay quien usa |, quien prefiere ||, quien usa //, etc.)

En este ejemplo, usamos SET para cambiar el valor de un campo. En concreto, antes de guardar cada dato, convertimos su código a mayúsculas (usando la función UPPER, que ya conocíamos), y guardamos NULL en vez de la edad si la edad tiene un valor incorrecto (0, por ejemplo), para lo que usamos la función IF, que aún no conocíamos. Esta función recibe tres parámetros: la condición a comprobar, el valor que se debe devolver si se cumple la condición, y el valor que se debe devolver cuando no se cumpla la condición.

Si añadimos un dato que tenga un código en minúsculas y una edad 0, y pedimos que se nos muestre el resultado, veremos esto:

```
INSERT INTO persona
VALUES ('p','Pedro',0)
+--------+---------+------+
| codigo | nombre  | edad |
```

```
+--------+---------+------+
| 1      | Juan    |   20 |
| P      | Pedro   | NULL |
+--------+---------+------+
```

Cuando un TRIGGER deje de sernos útil, podemos eliminarlo con DROP TRIGGER.

(Más detalles sobre TRIGGERS en el apartado 20 del manual de referencia de MySQL 5.0; más detalles sobre IF y otras funciones de control de flujo (CASE, IFNULL, etc) en el apartado 12.2 del manual de referencia de MySQL 5.0)

14.2. Ejercicios propuestos

14.1. Amplía esta base de datos de ejemplo, para que antes de cada borrado, se anote en una tabla de "copia de seguridad" el dato que se va a borrar.

14.2. Amplía esta base de datos de ejemplo, para que se antes de cada modificación se anote en una tabla "historico" el valor que antes tenía el registro que se va a modificar, junto con la fecha y hora actual.

Ejercicio resuelto con 1 tabla

Vamos a aplicar buena parte de lo que conocemos para hacer un ejercicio de repaso que haga distintas manipulaciones a una única tabla. Será una tabla que contenga datos de productos: código, nombre, precio y fecha de alta, para que podamos trabajar con datos de texto, numéricos y de tipo fecha.

Los pasos que realizaremos (por si alguien se atreve a intentarlo antes de ver la solución) serán:

- Crear la base de datos
- Comenzar a usarla
- Introducir 3 datos de ejemplo
- Mostrar todos los datos
- Mostrar los datos que tienen un cierto nombre
- Mostrar los datos que comienzan por una cierta inicial
- Ver sólo el nombre y el precio de los que cumplen una condición (precio > 22)
- Ver el precio medio de aquellos cuyo nombre comienza con "Silla"
- Modificar la estructura de la tabla para añadir un nuevo campo: "categoría"
- Dar el valor "utensilio" a la categoría de todos los productos existentes
- Modificar los productos que comienza por la palabra "Silla", para que su categoría sea "silla"
- Ver la lista categorías (sin que aparezcan datos duplicados)
- Ver la cantidad de productos que tenemos en cada categoría

Damos por sentado que MySQL está instalado. El primer paso es crear la base de datos:

```
CREATE DATABASE productos1;
```

Y comenzar a usarla:

```
USE productos1;
```

Para crear la tabla haríamos:

```
CREATE TABLE productos (
 codigo varchar(3),
 nombre varchar(30),
```

```
  precio decimal(6,2),
  fechaalta date,
  PRIMARY KEY (codigo)
);
```

Para introducir varios datos de ejemplo:

```
INSERT INTO productos VALUES ('a01','Afilador', 2.50, '2007-11-02');
INSERT INTO productos VALUES ('s01','Silla mod. ZAZ', 20, '2007-11-03');
INSERT INTO productos VALUES ('s02','Silla mod. XAX', 25, '2007-11-03');
```

Podemos ver todos los datos para comprobar que son correctos:

```
SELECT * FROM productos;
```

y deberíamos obtener

```
+--------+----------------+--------+------------+
| codigo | nombre         | precio | fechaalta  |
+--------+----------------+--------+------------+
| a01    | Afilador       |   2.50 | 2007-11-02 |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 |
+--------+----------------+--------+------------+
```

Para ver qué productos se llaman "Afilador":

```
SELECT * FROM productos WHERE nombre='Afilador';
+--------+----------+--------+------------+
| codigo | nombre   | precio | fechaalta  |
+--------+----------+--------+------------+
| a01    | Afilador |   2.50 | 2007-11-02 |
+--------+----------+--------+------------+
```

Si queremos saber cuáles comienzan por S:

```
SELECT * FROM productos WHERE nombre LIKE 'S%';
+--------+---------------+--------+------------+
| codigo | nombre        | precio | fechaalta  |
+--------+---------------+--------+------------+
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 |
+--------+---------------+--------+------------+
```

Si queremos ver cuales tienen un precio superior a 22, y además no deseamos ver todos los campos, sino sólo el nombre y el precio:

```
SELECT nombre, precio FROM productos WHERE precio > 22;
+---------------+--------+
| nombre        | precio |
+---------------+--------+
| Silla mod. XAX |  25.00 |
+---------------+--------+
```

Precio medio de las sillas:

```
SELECT avg(precio) FROM productos WHERE LEFT(nombre,5) = 'Silla';
+-------------+
| avg(precio) |
+-------------+
|   22.500000 |
+-------------+
```

Esto de mirar las primeras letras para saber si es una silla o no.… quizá no sea la mejor opción. Parece más razonable añadir un nuevo dato: la "categoría". Vamos a modificar la estructura de la tabla para hacerlo:

```
ALTER TABLE productos ADD categoria varchar(10);
```

Comprobamos qué ha ocurrido con un "select" que muestre todos los datos:

```
SELECT * FROM productos;
+--------+---------------+--------+------------+-----------+
| codigo | nombre        | precio | fechaalta  | categoria |
+--------+---------------+--------+------------+-----------+
| a01    | Afilador      |   2.50 | 2007-11-02 | NULL      |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 | NULL      |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 | NULL      |
+--------+---------------+--------+------------+-----------+
```

Ahora mismo, todas las categorías tienen el valor NULL, y eso no es muy útil. Vamos a dar el valor "utensilio" a la categoría de todos los productos existentes

```
UPDATE productos SET categoria='utensilio';
```

Y ya que estamos, modificaremos los productos que comienza por la palabra "Silla", para que su categoría sea "silla"

```
UPDATE productos SET categoria='silla' WHERE LEFT(nombre,5) = 'Silla';
+--------+---------------+--------+------------+-----------+
| codigo | nombre        | precio | fechaalta  | categoria |
+--------+---------------+--------+------------+-----------+
| a01    | Afilador      |   2.50 | 2007-11-02 | utensilio |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 | silla     |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 | silla     |
+--------+---------------+--------+------------+-----------+
```

Para ver la lista categorías (sin que aparezcan datos duplicados), deberemos usar la palabra "distinct"

```
SELECT DISTINCT categoria FROM productos;
+-----------+
| categoria |
+-----------+
| utensilio |
| silla     |
```

```
+-----------+
```

Finalmente, para ver la cantidad de productos que tenemos en cada categoría, deberemos usar "count" y agrupar los datos con "group by", así:

```
SELECT categoria, count(*) FROM productos GROUP BY categoria;
+-----------+----------+
| categoria | count(*) |
+-----------+----------+
| silla     |        2 |
| utensilio |        1 |
+-----------+----------+
```

Ejercicio propuesto con 1 tabla

Queremos crear una base de datos para almacenar información sobre PDAs. En un primer acercamiento, usaremos una única tabla llamada PDA, que tendrá como campos:

- Código
- Nombre
- Sistema Operativo
- Memoria (Mb)
- Bluetooth (s/n)

1. Crear la tabla.

2. Introducir en ella los datos:

- ptx, Palm Tungsten TX, PalmOS, 128, s
- p22, Palm Zire 22, PalmOS, 16, n
- i3870, Compaq Ipaq 3870, Windows Pocket PC 2002, 64, s

3. Realizar las consultas

a) Equipos con más de 64 Mb de memoria.

b) Equipos cuyo sistema operativo no sea "PalmOS".
c) Equipos cuyo sistema operativo contenga la palabra "Windows".
d) Lista de sistemas operativos (sin duplicados)
e) Nombre y código del equipo que más memoria tiene.
f) Nombre y marca (supondremos que la marca es la primera palabra del nombre, hasta el primer espacio) de cada equipo, ordenado por marca y a continuación por nombre.
g) Equipos con menos memoria que la media.
h) Cantidad de equipos con cada sistema operativo.
i) Sistemas operativos para los que tengamos 2 o más equipos en la base de datos.

4. Añadir a la tabla PDA un campo "precio", con valor NULL por defecto.

5. Modificar el dato del equipo con código "p22", para indicar que su precio es 119,50. Listar los equipos cuyo precio no conocemos.

Ejercicio propuesto con 2 tablas

1. Crear una base de datos llamada "deportes", y en ella dos tablas: jugador y equipo. Del jugador se desea almacenar: código (txt 12), nombre, apellido 1, apellido 2, demarcación (ej: delantero). De cada equipo: código (txt 8), nombre, deporte (ej: baloncesto). Cada equipo estará formado por varios jugadores, y supondremos que cada jugador sólo puede formar parte de un equipo.

2. Introducir los datos:

En equipos:

- rcm, Real Campello, baloncesto
- can, Canoa, natación
- ssj, Sporting de San Juan, futbol

En jugadores:

- rml, Raúl, Martínez, López, pivot (juega en el Real Campello)
- rl, Raúl, López, , saltador (del Canoa)
- jl, Jordi, López, , nadador crawl(del Canoa)
- rol, Roberto, Linares, , base (juega en el Real Campello)

3. Crear una consulta que muestre: nombre de deportista, primer apellido, demarcación, nombre de equipo (para todos los jugadores de la base de datos).

4. Crear una consulta que muestre el nombre de los equipos para los que no sabemos los jugadores.

5. Crear una consulta que muestre nombre y apellidos de los jugadores cuyo primer o segundo apellido es "López".

6. Crear una consulta que muestre nombre y apellidos de los nadadores.

7. Crear una consulta que muestre la cantidad de jugadores que hay en cada equipo.

8. Crear una consulta que muestre la cantidad de jugadores que hay en cada deporte.

9. Crear una consulta que muestre el equipo que más jugadores tiene.

10. Añadir a la tabla de jugadores un campo en el que poder almacenar la antigüedad (en años), que tenga como valor por defecto NULL, y modificar la ficha de "Roberto Linares" para indicar que su antigüedad es de 4 años.