PROGRAMACIÓN AVANZADA CON PYTHON

(CEFIRE CTEM)

Introducción a las bases de datos

Autora: María Paz Segura Valero (segura_marval@gva.es)

CONTENIDO

1. Introducción

Conforme vamos creando programas más sofisticados, aparece la necesidad de almacenar de forma persistente la información que manejan para poder utilizarla en otro momento. Por ejemplo, en el disco duro del ordenador, el móvil, la nube, etc.

Tradicionalmente han existidos dos sistemas de almacenamiento de información: los *ficheros* y las *bases de datos*.

Aprender a gestionar la información que almacenamos en **ficheros** es relativamente sencilla y puede ser útil para almacenar información muy básica. Por ejemplo: lista de credenciales, datos de un vuelo, aplicaciones web permitidas, etc.

Pero si necesitamos almacenar gran cantidad de información relacionada entre sí entonces debemos aprender a trabajar con **bases de datos**, ya que ofrecen mayores prestaciones.

2. Bases de datos

Una **base de datos** está formada por un conjunto de datos relacionados entre sí que recoge las necesidades de información de una organización.

Ejemplos:

- La información que maneja la aplicación *Ítaca* de la *Generalitat Valenciana*: profesorado, alumnado, calificaciones, horarios, expedientes, etc.
- Datos de una agenda electrónica: nombres, direcciones, correos electrónicos de los contactos, etc.
- Gestión de reservas en una pista deportiva: usuarios, fechas, horas, pistas, etc.

Para facilitar la creación y gestión de las bases de datos existen los **Sistemas gestores de bases de datos** (SGBD) que son conjuntos de programas que permiten crear la estructura de las bases de datos y manipular la información que almacenan. Además, proporcionan una capa de seguridad extra ya que se ocupan de mantener la integridad de los datos, permitir el acceso simultáneo a los mismos, controlar los permisos de acceso a determinada información, ofrecer sistemas de recuperación ante fallos, etc.

Existen distintos modelos de bases de datos (jerárquicos, transaccionales, documentales, orientados a objetos, etc) pero nosotros nos centraremos en el **modelo relacional** cuya característica principal es que almacena los datos y las relaciones entre los mismos en unas estructuras de datos llamadas **tablas**.

2.1. Tablas, columnas y filas

Una **tabla** permite almacenar datos de una entidad, objeto o concepto de la vida real y las relaciones entre sí.

La representación gráfica de una tabla suele ser una rejilla formada por filas y columnas. Las *columnas* se corresponden con el nombre de los atributos o campos de los que queremos almacenar información (por ejemplo: ciudad, fecha de nacimiento, altura) y las *filas* se corresponden con los registros o elementos de los que estamos almacenando dicha información.

Ejemplo: Tabla PERSONA que almacena información de cinco campos (dni, nombre, edad, ciudad, altura) de cada persona de una organización.

Columnas: nombres de los atributos

Filas: cada fila se corresponde con una persona

DNI	Nombre	Edad	Ciudad	Altura
111111111	Ana	15	Morella	165
222222222	Pepito	8	Elche	132
333333333	Mireia	8	Manises	170
444444444	Joan	82	Oliva	170

2.2. Claves y relaciones

Una vez que tenemos claros los datos que queremos almacenar en una tabla de una entidad necesitamos encontrar la manera de diferenciar unos registros de otros. Esta función la realizan las claves candidatas.

Una **clave candidata** es un conjunto de atributos de una tabla (mínimo uno) que permiten identificar **unívoca** y **mínimamente** un registro de otro. Dicho de otra forma, no habrá varias filas de una tabla con el mismo valor en los mismos atributos. Por ejemplo: el DNI de una persona sería una clave candidata del ejemplo anterior.

A veces, las claves candidatas aparecen de forma natural y otras veces hay que añadir un campo extra a la tabla. Por ejemplo, un campo numérico secuencial.

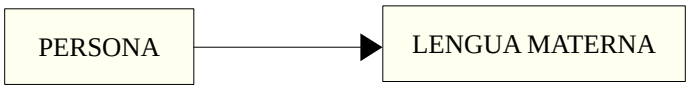
De entre todas las claves candidatas que aparezcan en una tabla, elegiremos una como **clave primaria** y el resto quedarán como **claves alternativas**.

Ya que las claves candidatas sirven para identificar un registro de una tabla, no se permite almacenar valores nulos en ellas. Es decir, es obligatorio almacenar un valor en cada campo perteneciente a una clave candidata.

Hasta ahora hemos visto cómo almacenar información de una entidad en una tabla, pero ¿cómo almacenamos las relaciones entre distintas entidades?

Veamos algunos ejemplos de relaciones entre distintas entidades.

Ejemplo 1: una persona tiene una lengua materna (aunque luego pueda hablar más de un idioma) y una misma lengua materna puede ser hablada por muchas personas. Relación 1:N.



Ejemplo 2: las personas pueden afiliarse a distintas asociaciones por motivos diversos y una asociación puede tener afiliadas muchas personas. Relación N:N.

Existen más tipos de relaciones entre entidades pero nos centraremos en estas dos por ser las más comunes.

Ahora ya podemos dar respuesta a nuestra pregunta. Las relaciones entre entidades se almacenan como claves ajenas entre tablas.

Una **clave ajena** en una entidad es un conjunto de atributos (mínimo uno) que se corresponde con la clave primaria de otra entidad, o de la misma según el tipo de relación.

Veamos su aplicación a los ejemplos anteriores.

Ejemplo 1: Relación 1:N entre PERSONA y LENGUA MATERNA.

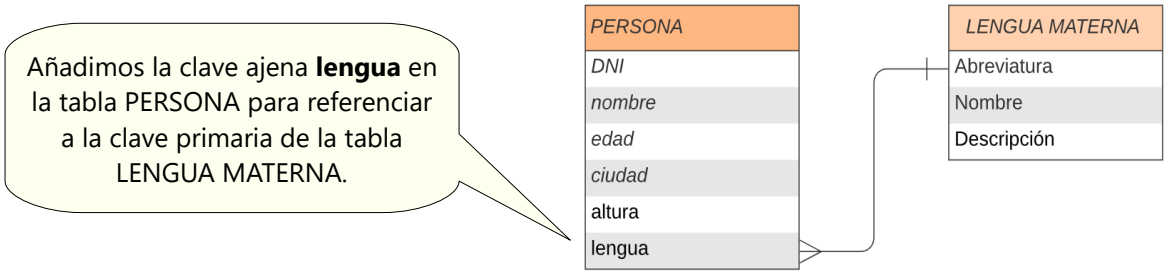


Tabla PERSONA

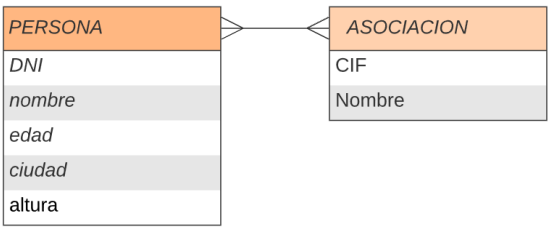
DNI	nombre	edad	ciudad	altura	lengua
111111111	Ana	15	Morella	165	cs
222222222	Pepito	8	Elche	132	cs
333333333	Mireia	8	Manises	170	va
444444444	Joan	82	Oliva	170	fr

Tabla LENGUA MATERNA

Abreviatura	Nombre	Descripción
cs	castellano	España
va	valenciano	C.Valenciana
en	inglés	Reino Unido
fr	francés	Francia

Como se puede observar, un registro de la tabla PERSONA puede estar relacionada con un registro de la tabla LENGUA MATERNA (por ejemplo: Ana y Pepito) pero un registro de la tabla LENGUA MATERNA puede estar relacionada con varios registros de la tabla PERSONA o con ninguno.

Ejemplo 2: Relación N:N entre PERSONA y ASOCIACIÓN.



En este caso, una *persona* puede estar afiliada a muchas *asociaciones* y una *asociación* puede tener muchas *personas* afiliadas.

La solución pasa por crear una nueva tabla llamada AFILIACIONES que contendrá un clave ajena hacia la tabla PERSONA y una clave ajena hacia la tabla ASOCIACION. Además, podremos añadir nuevos campos como la *fecha de alta* de dicha *persona* en dicha *asociación*.

Tabla PERSONA

DNI	nombre	edad	ciudad	altura
111111111	Ana	15	Morella	165
222222222	Pepito	8	Elche	132
333333333	Mireia	8	Manises	170
444444444	Joan	82	Oliva	170

Tabla ASOCIACION

CIF	Nombre
1	Asociación de vecinos
2	Programando juntos
3	Club de cocina

Tabla AFILIACIONES

persona	asociacion	fecha alta
444444444	1	15/01/2020
111111111	1	30/04/2021
111111111	2	01/05/2021

3. El SGBD integrado SQLite

SQLite es un SGBD contenido en una librería del lenguaje C que viene integrada en Python. Así que no necesitamos realizar ninguna instalación extra para trabajar con bases de datos SQLite en programas de Python. Simplemente habrá que **importar la librería** correspondiente, como a continuación:

```
import sqlite3
```

No obstante, puede ser muy útil poder acceder al contenido de nuestras bases de datos para hacer comprobaciones sin tener que crear un programa específico en Python y también probar las instrucciones SQL que vamos a ir aprendiendo durante esta unidad. Así que, instalaremos el **terminal de comandos** de SQLite.

3.1. Instalar el terminal de SQLite

Existen versiones del terminal **sqlite3** para varios sistemas operativos: Android, Linux, Mac OS, Windows, etc. En los siguientes apartados se explica con detalle cómo instalar el terminal para Windows y Lliurex.

3.1.1. Instalar en Windows

Sigue los pasos que se indican a continuación:

- Accede a la página web https://www.sqlite.org/download.html
- Busca el apartado **Precompiled Binaries for Windows**.

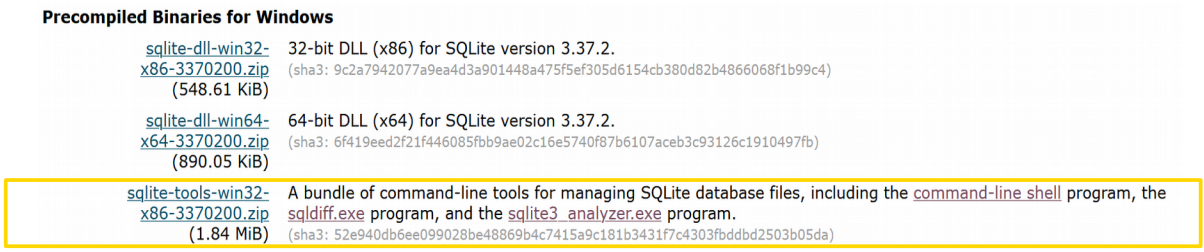


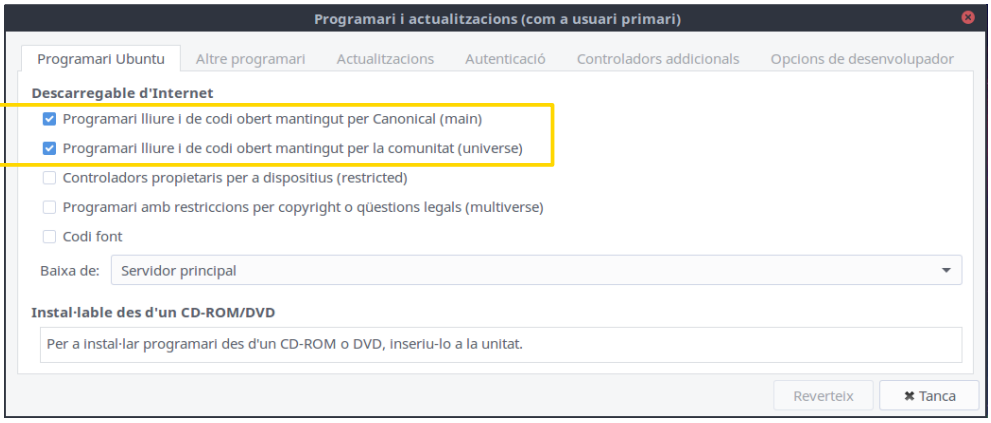
Figura 1: Página web de descargas

- Descarga el fichero *sqlite-tools-win32-x86-3370200.zip* o similar.
- Descomprime el fichero **.zip** y copia el fichero ejecutable **sqlite3.exe** en la carpeta que quieras. Te recomiendo ubicarlo dentro de tu carpeta de proyectos de Python.

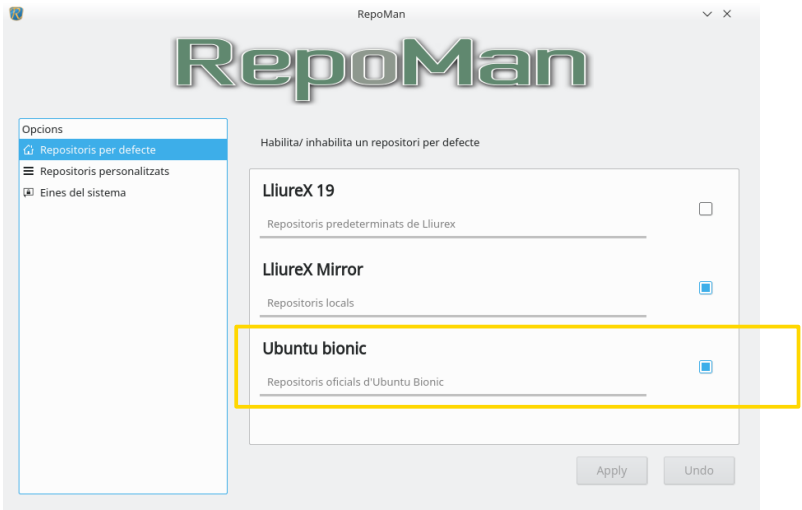
3.1.2. Instalar en Lliurex

En la página oficial aparece un fichero para Linux de 32 bits pero es posible que tu sistema operativo sea de 64 bits y tengas problemas para hacerlo funcionar, así que una alternativa es seguir estos pasos:

- Comprobamos los repositorios del sistema operativo:
  - En Lliurex 16, asegúrate de tener marcados los dos primeros repositorios en la opción de menú **Parámetros/Repositorios** de la aplicación **Gestor de paquetes Synaptic**.



  - En Lliurex 19, debes tener marcado el repositorio **Ubuntu bionic** en la pestaña **Repositoris por defecto** de la aplicación **RepoMan**.



- Instalamos el paquete **sqlite3** con el siguiente comando:

```
sudo apt install sqlite3
```

Para instalar en otros sistemas operativos, consulta la página web oficial: https://www.sqlite.org/download.html

3.2. Uso del terminal SQLite

Una vez que tenemos instalado el terminal SQLite en nuestro sistema operativo, podemos abrirlo de varias formas:

a) Sin especificar una base de datos: en ese caso estaremos trabajando con una base de datos temporal y sus efectos se perderán cuando cerremos el terminal. No obstante, siempre podremos salvar el contenido de los datos temporales en una base de datos con el comando `.save` que veremos más adelante.

```
sqlite3
```

b) Especificando la base de datos: si la base de datos existe entonces se abrirá, pero si la base de datos no existe entonces se creará en ese momento.

```
sqlite3 nombre_fichero.db
```

*Para el sistema operativo Windows deberás escribir **sqlite3.exe** en lugar de sqlite3, como en Linux o Mac OS.*

Una vez estemos dentro del terminal de SQLite podremos ejecutar directamente instrucciones SQL o comandos específicos de **sqlite3**. En la siguiente tabla se presentan algunos de estos comandos.

Comando	Descripción

`.open`	Abre una base de datos existente o la crea si no existe.
	
	Ejemplo 1: Abrir una base de datos que se encuentra en la misma carpeta que el ejecutable `sqlite3`.
	`.open colegio.db`
	
	Ejemplo 2: Abrir una base de datos que se encuentra en otra carpeta.
	`.open C:/bbdd/colegio.db`

`.save`	Guarda en el fichero de base de datos la definición y contenido de la base de datos actual. Hay que tener cuidado porque no pide confirmación para modificar una base de datos ya existente. Ejemplo 1: Guardar una base de datos en la misma carpeta que `sqlite3`. `.save colegio.db`

	Ejemplo 2: Guardar una base de datos que se encuentra en otra carpeta. `.save C:/bbdd/colegio.db`
`.read`	Este comando ejecuta las instrucciones SQL que contenga un fichero de texto. Ejemplo: `.open myscript.sql`

`.schema`	Muestra la definición de la base de datos
	
	Ejemplo:
	`sqlite> .schema`
	`CREATE TABLE tbl2 (f1 varchar(30) primary key, f2 text, f3 real);`
	`CREATE TABLE tbl1 (f1 varchar(30) primary key);`

`.tables`	Muestra el nombre de las tablas existentes
	
	Ejemplo:
	`sqlite> .tables`
	`tbl1  tbl2`

`.mode column`	Visualiza los resultados de las consultas en formato columna.
`.exit`	Cerrar el terminal de SQLite

`.help`	Muestra ayuda sobre los comandos del terminal. Si escribimos el nombre de un comando a continuación, muestra la información solamente de ese comando.
	
	Ejemplo:
	`sqlite> .help schema`
	`.schema ?PATTERN?  Show the CREATE statements matching PATTERN`
	`  Options:`
	`  --indent  Try to pretty-print the schema`
	`  --nosys  Omit objects whose names start with "sqlite_"`

4. Lenguaje SQL

El lenguaje SQL (*Structured Query Language*) es el lenguaje más popular para trabajar con bases de datos relacionales. Se compone de varios sublenguajes.

Aunque SQL ofrece una gran variedad de instrucciones, en esta unidad nos centraremos en las instrucciones básicas de estos sublenguajes:

- El *Lenguaje de definición de datos* o *DDL* que permite crear la estructura de la base de datos.
- El *Lenguaje de manipulación de datos* o *DML* que permite insertar, modificar y borrar datos en la base de datos.
- El *Lenguaje de consultas* o *DQL* que permite recuperar o seleccionar información de la base de datos.

4.1. DDL: Lenguaje de definición de datos

El **DDL** o *Lenguaje de definición de datos* ofrece instrucciones para crear, modificar y borrar distintos elementos de las bases de datos. Nosotros nos centraremos en la creación y borrado de tablas.

4.1.1. Crear tablas

Para crear tablas de datos utilizaremos la instrucción CREATE con la que definiremos las columnas que contendrá, el tipo de datos que se almacenarán en las columnas y las restricciones necesarias (por ejemplo, la definición de las claves candidatas y ajenas).

La sintaxis es la siguiente:

```
CREATE TABLE [IF NOT EXISTS] nombre_tabla (
columna1 tipo_dato [NOT NULL],
...
columnan tipo_dato [NOT NULL],

PRIMARY KEY(lista_columnas) [,
FOREIGN KEY(lista_columnas)
    REFERENCES tabla_referenciada(campos_ref)]
);
```

Definición de las columnas

Definición de las claves

En la cabecera debemos indicar el nombre de la tabla teniendo en cuenta que no pueden existir varias tablas con el mismo nombre en la misma base de datos.

- Si utilizamos la expresión **IF NOT EXISTS** entonces la instrucción CREATE solamente se ejecutará en caso de que no exista una tabla en la base de datos con dicho nombre.
- Si no utiliza la expresión **IF NOT EXISTS** y existe una tabla con el mismo nombre, se producirá un error.

En la parte de la definición de las columnas debemos indicar la siguiente información:

- **columna$_x$** es el nombre de la columna que estamos creando. Dentro de una misma tabla no puede haber dos columnas con el mismo nombre.
- **tipo_dato** indica el tipo de valores que vamos a poder almacenar en dicha columna. Los tipos de datos que podremos utilizar dependerán del SGBD en el que vayamos

a utilizar esta sentencia. Nosotros utilizaremos los siguientes: INTEGER (número entero), REAL (número real o decimal), TEXT (cadena de texto).

- **NOT NULL** indica que no es un valor optativo. Si se indica en la definición de un campo estamos indicando que no se pueden crear registros en esta tabla para los que no se asignen valores a este campo.

En la parte de la definición de las claves debemos indicar la siguiente información:

- La clave primaria utilizando la expresión **PRIMARY KEY(*lista_columnas*)** donde ***lista_columnas*** es la lista de nombres de columnas, separadas por comas, que forman parte de la clave primaria.

  Ejemplos:

  ```
  PRIMARY KEY(DNI)
  PRIMARY KEY(nombre, apellido1, apellido2)
  ```

- Si la tabla tiene campos que actúan como claves ajenas, entonces deberemos utilizar una expresión **FOREIGN KEY(*lista_columnas*) REFERENCES *tabla_referenciada*(*campos_ref*)** por cada clave ajena que contenga la tabla. La ***lista_columnas*** es la lista de columnas de la tabla que forman parte de la clave ajena y ***campos_ref*** son los campos de la tabla referenciada a la que hacen referencia (normalmente coinciden con la clave primaria de la tabla referenciada).

  Ejemplos:

  ```
  FOREIGN KEY(lengua) REFERENCES LENGUA_MATERNA(abreviatura)
  FOREIGN KEY(nom_inqui, ape1_inqui, ape2_inqui) REFERENCES
     INQUILINO(nombre, apellido1, apellido2)
  ```

Te recomiendo que abras el terminal **sqlite3** y pruebes las sentencias de los ejemplos que hay a continuación.

Ejemplo 1: Creamos la tabla LENGUA_MATERNA con la definición de los campos y la clave primaria.

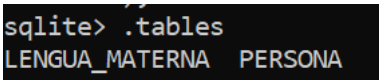
```
CREATE TABLE LENGUA_MATERNA (
abreviatura TEXT,
nombre TEXT,
descripcion TEXT,
PRIMARY KEY(abreviatura)
);
```

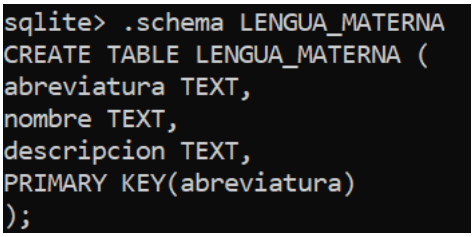
Ejemplo 2: Ahora creamos la tabla PERSONA con la definición de los campos y la definición de todas las claves, tanto la primaria como la clave ajena.

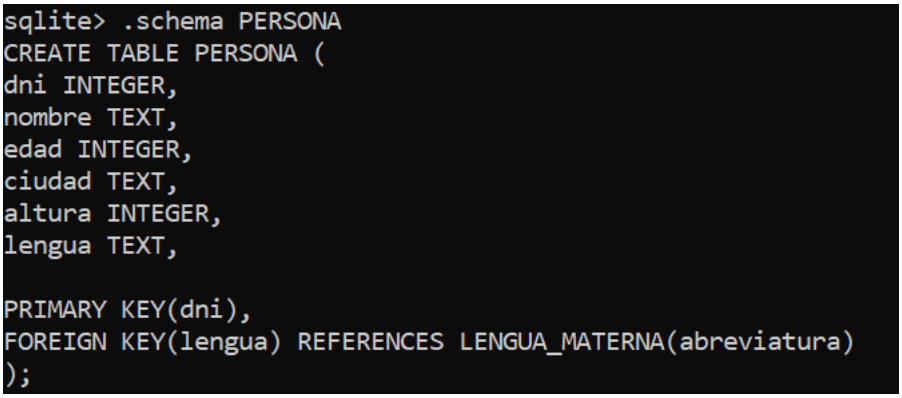
```
CREATE TABLE PERSONA (
dni INTEGER,
nombre TEXT,
edad INTEGER,
ciudad TEXT,
altura INTEGER,
lengua TEXT,

PRIMARY KEY(dni),
FOREIGN KEY(lengua) REFERENCES LENGUA_MATERNA(abreviatura)
);
```

Para comprobar que se han creado correctamente las dos tablas en la base de datos, puedes ejecutar los siguientes comandos: `.tables`, `.schema` *`nombre_tabla`*.







4.1.2. Borrar tablas

Si queremos borrar una tabla de la base de datos, y no nos referimos a vaciarla de contenido sino a borrar la estructura completa de la tabla, podemos utilizar la siguiente sentencia:

DROP TABLE [IF EXISTS] *nombre_tabla*;

Donde ***nombre_tabla*** es el nombre de la tabla que queremos eliminar de la base de datos.

- Si la tabla no existe entonces dará un error.
- Podemos utilizar la expresión **IF EXISTS** para evitarnos este problema. De esta forma, si existe entonces la borrará pero si no existe entonces no pasará nada.

La sentencia **DROP TABLE** borra una tabla cada vez, así que deberemos ejecutar tantas sentencias de este tipo como tablas queramos eliminar y siempre en el orden inverso en el que las hemos creado si contienen claves ajenas.

Ejemplo 1: Borramos una tabla sin la cláusula **IF EXISTS**.

```
DROP TABLE persona;
```

```
sqlite> select * from persona;
dni         nombre  edad  ciudad   altura  lengua
---------   ------  ----  -------  ------  ------
111111111   Ana     15    Morella  165.0   cs
222222222   Pepito  8     Elche    132.0   cs
333333333   Mireia  8     Manises  170.0   va
444444444   Joan    82    Oliva    170.0   va
```

La primera vez que ejecutamos **DROP TABLE** la tabla existe, así que, se borra sin problemas. La segunda vez, se produce un error porque estamos intentando borrar una tabla inexistente.

```
sqlite> DROP TABLE persona;
sqlite> select * from persona;
Error: no such table: persona
sqlite> DROP TABLE persona;
Error: no such table: persona
```

Ejemplo 2: Borramos una tabla con la cláusula **IF EXISTS**.

```
DROP TABLE IF EXISTS persona;
```

```
sqlite> select * from persona;
dni         nombre  edad  ciudad   altura  lengua
---------   ------  ----  -------  ------  ------
111111111   Ana     15    Morella  165.0   cs
222222222   Pepito  8     Elche    132.0   cs
333333333   Mireia  8     Manises  170.0   va
444444444   Joan    82    Oliva    170.0   va
```

```
sqlite> DROP TABLE IF EXISTS persona;
sqlite> select * from persona;
Error: no such table: persona
sqlite> DROP TABLE IF EXISTS persona;
```

La tabla se borra cuando existe pero, si no existe, no da error la sentencia **DROP TABLE**.

4.2. DML: Lenguaje de manipulación de datos

El **DML** o *Lenguaje de manipulación de datos* ofrece instrucciones para poblar las tablas de registros. Las sentencias **INSERT**, **UPDATE** y **DELETE** nos permitirán añadir registros, modificar los datos de los ya existentes o eliminar algunos de ellos.

4.2.1. INSERT: inserción de registros

Cuando ya tenemos creadas las tablas de la base de datos, podemos utilizar la siguiente instrucción para añadir registros a las mismas:

```
INSERT INTO nombre_tabla [(columna1, ..., columnan)]
VALUES(valor1, ..., valorn);
```

Donde ***nombre_tabla*** es el nombre de la tabla a la que queremos añadir un registro. A continuación, entre paréntesis, podemos indicar la lista de columnas de la tabla en el orden que queramos. Si no se indica, entonces se tomará el orden de las columnas conforme aparecen en la definición de la tabla.
Y, por último, indicamos la lista de valores que tomarán las columnas del registro. El primer valor se almacenará en la primera columna de la lista, el segundo valor en la segunda columna y así sucesivamente.

Ejemplo 1: Omitimos la lista de columnas.

```
INSERT INTO LENGUA_MATERNA
VALUES('cs', 'castellano', 'España');
```

```
abreviatura  nombre      descripcion
-----------  ----------  -----------
cs           castellano  España
```

Ejemplo 2: Indicamos la lista de columnas en otro orden distinto del que aparece en la definición de la tabla y funciona sin problemas.

```
INSERT INTO LENGUA_MATERNA(descripcion, nombre, abreviatura)
VALUES('valenciano', 'C.Valenciana', 'va');
```

```
abreviatura  nombre      descripcion
-----------  ----------  ------------
cs           castellano  España
va           valenciano  C.Valenciana
```

4.2.2. DELETE: borrado de registros

Si lo que queremos es borrar varios registros de una tabla o todos ellos, podemos utilizar la sentencia DELETE.

Ésta es su sintaxis:

```
DELETE FROM nombre_tabla
[WHERE condición_de_búsqueda];
```

En la expresión **WHERE** debemos indicar la condición de búsqueda para seleccionar los registros que queremos eliminar. La veremos con mayor profundidad en la sentencia de selección **SELECT**.

Ejemplo 1: Borramos los registros cuya abreviatura sea igual a "cs".

```
DELETE FROM LENGUA_MATERNA WHERE abreviatura = "cs";
```

```
sqlite> select * from lengua_materna;
abreviatura  nombre      descripcion
-----------  ----------  ------------
cs           castellano  España
va           valenciano  C.Valenciana
en           inglés      Reino Unido
fr           francés     Francia
```



```
sqlite> DELETE FROM LENGUA_MATERNA
   ...> WHERE abreviatura = "cs";
sqlite> select * from lengua_materna;
abreviatura  nombre      descripcion
-----------  ----------  ------------
va           valenciano  C.Valenciana
en           inglés      Reino Unido
fr           francés     Francia
```

Ejemplo 2: Borramos aquellas personas que vivan en Manises y tengan menos de 50 años.

```
DELETE FROM PERSONA WHERE ciudad = "Manises" and edad < 50;
```

```
sqlite> select * from persona;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
333333333  Mireia  8     Manises  170.0   va
444444444  Joan    82    Oliva    170.0   va
```



```
sqlite> DELETE FROM PERSONA
   ...> WHERE ciudad = "Manises" and edad < 50;
sqlite> select * from persona;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
444444444  Joan    82    Oliva    170.0   va
```

Si omitimos la parte del **WHERE** entonces se borrarán todos los registros de una tabla, aunque no su estructura. Ésta es la diferencia con la instrucción **DROP TABLE**.

Ejemplo 3: Borramos todos los registros de la tabla LENGUA_MATERNA.

```
DELETE FROM LENGUA_MATERNA;
```

```
sqlite> select * from lengua_materna;
abreviatura  nombre      descripcion
-----------  ----------  ------------
cs           castellano  España
va           valenciano  C.Valenciana
en           inglés      Reino Unido
fr           francés     Francia
```



```
sqlite> DELETE FROM LENGUA_MATERNA;
sqlite> select * from lengua_materna;
sqlite> .schema lengua_materna
CREATE TABLE LENGUA_MATERNA (
abreviatura TEXT,
nombre TEXT,
descripcion TEXT,
PRIMARY KEY(abreviatura)
);
```

4.2.3. UPDATE: modificación de registros

Podemos modificar todos los registros de una tabla o una selección de ellos. Y podemos modificar el valor de varias columnas de un registro o solo de una de ellas.

Ésta es la sintaxis de la sentencia UPDATE:

```
UPDATE nombre_tabla
SET    columna1 = nueva_valor1,
       ...
       columnan  = nueva_valorn
[WHERE condición_de_búsqueda];
```

La sintaxis de esta sentencia es parecida a la de la sentencia **DELETE** excepto por el apartado **SET**. En él se deben indicar los valores nuevos de las columnas que queremos modificar.

En este caso también se puede omitir el apartado **WHERE** si queremos actualizar todos los registros de la tabla.

Ejemplo 1: Modificamos la edad de todas las personas.

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

```
sqlite> select * from persona;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
333333333  Mireia  8     Manises  170.0   va
444444444  Joan    82    Oliva    170.0   va
```

```
sqlite> UPDATE PERSONA
   ...> SET edad = edad + 1;
sqlite> select * from persona;
dni         nombre  edad  ciudad   altura  lengua
---------   ------  ----  -------  ------  ------
111111111   Ana     16    Morella  165.0   cs
222222222   Pepito  9     Elche    132.0   cs
333333333   Mireia  9     Manises  170.0   va
444444444   Joan    83    Oliva    170.0   va
```

Ejemplo 2: Modificamos la altura de Joan porque debe ser 175 y no 170.

```
UPDATE PERSONA
SET altura = 175
WHERE nombre = "Joan";
```

```
sqlite> select * from persona;
dni         nombre  edad  ciudad   altura  lengua
---------   ------  ----  -------  ------  ------
111111111   Ana     15    Morella  165.0   cs
222222222   Pepito  8     Elche    132.0   cs
333333333   Mireia  8     Manises  170.0   va
444444444   Joan    82    Oliva    170.0   va
```



```
sqlite> UPDATE PERSONA
   ...> SET altura = 175
   ...> WHERE nombre = "Joan";
sqlite> select * from persona;
dni         nombre  edad  ciudad   altura  lengua
---------   ------  ----  -------  ------  ------
111111111   Ana     15    Morella  165.0   cs
222222222   Pepito  8     Elche    132.0   cs
333333333   Mireia  8     Manises  170.0   va
444444444   Joan    82    Oliva    175.0   va
```

4.3. DQL: Lenguaje de consulta de datos

El *Lenguaje de consultas de datos* o *DQL* permite realizar consultas de los registros de las tablas de una base de datos. Para ello, se utiliza la instrucción más conocida de este lenguaje: la sentencia **SELECT**.

Aunque esta instrucción ofrece muchas prestaciones y podemos construir una sentencia **SELECT** muy sofisticada, en esta introducción a bases de datos, nos centraremos en una versión más simple de la misma.

4.3.1. Seleccionar registros de una tabla

Una primera aproximación a la sintaxis de la sentencia **SELECT** puede ser la siguiente:

```
SELECT lista_columnas
FROM nombre_tabla
```

[WHERE *condición_búsqueda*]
[ORDER BY *nombre_columna* [ASC|DESC]];

Donde:

- En la cláusula **SELECT** indicaremos el nombre de las columnas de la tabla que queremos ver, separadas por comas.
- En la cláusula **FROM** indicaremos el nombre de la tabla que queremos consultar.
- En la cláusula **WHERE** escribiremos la condición de búsqueda que deberán cumplir los registros que se mostrarán en el resultado de la consulta.
  - Esta condición de búsqueda se forma con expresiones lógicas. Podemos comparar valores con operadores relacionales (>, >=, <, >=, =, !=) y concatenar expresiones lógicas con los operadores lógicos (and, or, not).
  - Si no se utiliza esta cláusula entonces se mostrarán todos los registros de la tabla.

Para conocer más de esta cláusula, consulta la web https://www.sqlitetutorial.net/sqlite-where/

- En la cláusula **ORDER BY** indicaremos la lista de columnas que se utilizarán para ordenar los registros.
  - Se ordenarán según la lista de campos especificada de izquierda a derecha. Por defecto en orden ascendente (**ASC**) aunque podemos cambiar el orden a descendente (**DESC**).
  - Si no se utiliza esta cláusula entonces los registros se ordenarán por la lista de campos especificada en la cláusula **SELECT**, de izquierda a derecha y ascendentemente.

Ejemplo 1: Muestra todos los registros de la tabla PERSONA.

```
SELECT * FROM PERSONA;
```

```
sqlite> select * from persona;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
333333333  Mireia  8     Manises  170.0   va
444444444  Joan    82    Oliva    170.0   va
```

Ejemplo 2: Muestra los campos *nombre* y *ciudad* de todos los registros de la tabla PERSONA.

```
SELECT nombre, ciudad FROM PERSONA;
```

```
sqlite> select nombre, ciudad from persona;
nombre  ciudad
------  -------
Ana     Morella
Pepito  Elche
Mireia  Manises
Joan    Oliva
```

Ejemplo 3: Muestra todos los registros de la tabla PERSONA ordenados descendentemente por la *altura* y ascendentemente por el *nombre*.

```
SELECT * FROM PERSONA
ORDER BY altura DESC, nombre ASC;
```

```
sqlite> select * from persona order by altura desc, nombre asc;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
444444444  Joan    82    Oliva    170.0   va
333333333  Mireia  8     Manises  170.0   va
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
```

Ejemplo 4: Muestra los registros de la tabla PERSONA con *altura* menor de 170.

```
SELECT * FROM PERSONA
WHERE altura < 170;
```

```
sqlite> select * from persona where altura < 170;
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
222222222  Pepito  8     Elche    132.0   cs
```

Ejemplo 5: Muestra los registros de la tabla PERSONA con *altura* menor de 170 y *edad* igual a 8.

```
SELECT * FROM PERSONA
WHERE altura < 170 and edad = 8;
```

```
sqlite> select * from persona where altura < 170 and edad = 8;
dni        nombre  edad  ciudad  altura  lengua
---------  ------  ----  ------  ------  ------
222222222  Pepito  8     Elche   132.0   cs
```

4.3.2. Seleccionar registros de varias tablas

También podemos seleccionar registros de varias tablas que estén relacionados. Normalmente los uniremos por alguna clave ajena. Podemos incluir en la cláusula **WHERE** las condiciones de unión de los registros de dichas tablas.

SELECT ***lista_columnas***
FROM ***tabla$_1$ T1, tabla$_2$ T2, ..., tabla$_n$ Tn***
WHERE

Donde:

- En la cláusula **FROM** escribimos la lista de tablas implicadas.
- Para cada nombre de tabla podemos especificar un alias que utilizaremos delante de sus campos para indicar la tabla y evitar ambigüedades.

Ejemplo 1: Mostrar los datos de aquellas personas cuya lengua materna sea la de España.

```
SELECT * FROM PERSONA P, LENGUA_MATERNA LM
WHERE P.lengua = LM.abreviatura and
      LM.descripcion = "España";
```

```
sqlite> SELECT * FROM PERSONA P, LENGUA_MATERNA LM
   ...> WHERE P.lengua = LM.abreviatura and LM.descripcion = "España";
dni        nombre  edad  ciudad   altura  lengua  abreviatura  nombre      descripcion
---------  ------  ----  -------  ------  ------  -----------  ----------  -----------
111111111  Ana     15    Morella  165.0   cs      cs           castellano  España
222222222  Pepito  8     Elche    132.0   cs      cs           castellano  España
```

Se pueden mezclar las condiciones de unión de tablas con las condiciones de búsqueda de los registros.

Ejemplo 2: Mostrar los datos de aquellas personas cuya lengua materna sea la de España y tengan 8 años.

```
SELECT * FROM PERSONA P, LENGUA_MATERNA LM
WHERE P.lengua = LM.abreviatura and
      LM.descripcion = "España" and P.edad = 8;
```

```
sqlite> SELECT * FROM PERSONA P, LENGUA_MATERNA LM
   ...> WHERE P.lengua = LM.abreviatura and
   ...>  LM.descripcion = "España" and P.edad = 8;
dni        nombre  edad  ciudad  altura  lengua  abreviatura  nombre      descripcion
---------  ------  ----  ------  ------  ------  -----------  ----------  -----------
222222222  Pepito  8     Elche   132.0   cs      cs           castellano  España
```

Para mostrar todos los campos de una tabla podemos indicarlo así: ***nombre_tabla.****

Ejemplo 3: Mostrar los datos de las personas afiliadas a la "Asociación de vecinos".

```
SELECT P.*
FROM PERSONA P, AFILIACIONES AF, ASOCIACION ASO
WHERE P.dni = AF.persona and AF.asociacion = ASO.cif and
      ASO.nombre = 'Asociación de vecinos';
```

```
sqlite> SELECT P.*
   ...> FROM PERSONA P, AFILIACIONES AF, ASOCIACION ASO
   ...> WHERE P.dni = AF.persona and AF.asociacion = ASO.cif and
   ...>  ASO.nombre = 'Asociación de vecinos';
dni        nombre  edad  ciudad   altura  lengua
---------  ------  ----  -------  ------  ------
111111111  Ana     15    Morella  165.0   cs
444444444  Joan    82    Oliva    170.0   va
```

5. Conclusiones

A lo largo de este documento hemos explicado qué es una base de datos, cuáles son los elementos básicos para crear una (tablas y relaciones) y qué instrucciones SQL podemos utilizar para gestionarlas.

El mundo de las bases de datos es mucho más amplio de lo que hemos podido ver aquí y el lenguaje SQL ofrece muchas más instrucciones y cláusulas de las que hemos presentado. Pero éste puede ser un buen punto de partida. Te invito a seguir conociendo más sobre bases de datos a través de tutoriales en Internet, por ejemplo: https://www.sqlitetutorial.net

En el aula virtual dispones de los siguientes ficheros para hacer pruebas:

Comando	Descripción
ejemplos.sql	Fichero de texto que contiene las instrucciones SQL necesarias para crear la base de datos que hemos utilizado en los ejemplos de este documento y poblar las tablas con registros. Puedes ejecutarlo en sqlite3 con `.read ejemplos.sql`

ejemplos.db	Fichero de base de datos después de ejecutar las instrucciones SQL del fichero anterior. Puedes abrirlo en sqlite3 y trabajar sobre él: `.open ejemplos.db`

6. Fuentes de información

- Bases de datos SQLite de *Hektor Profe*: https://docs.hektorprofe.net/python/bases-de-datos-sqlite/
- DB-API 2.0 interfaz para bases de datos SQLite: https://docs.python.org/es/3.10/library/sqlite3.html
- Command Line Shell For SQLite: https://www.sqlite.org/cli.html
- Tutorial de SQLite: https://www.sqlitetutorial.net
- «Curso de Programación en Python», José Luis Tomás Navarro.