

10. Acceso a bases de datos relacionales

10.1. Nociónes mínimas de bases de datos relacionales

Una base de datos relacional es una estructura mucho más compleja (pero también más versátil) que un simple fichero de texto o binario.

En una base de datos relacional hay varios bloques de datos, llamados "**tablas**" (por ejemplo, la tabla de "clientes", la tabla de "proveedores", la tabla de "productos"...). Cada tabla está formada por una serie de "**registros**" (por ejemplo, el cliente "Juan López", el cliente "Pedro Álvarez"...). Finalmente, cada registro está formado por varios "**campos**" (de cada cliente podemos almacenar su nombre, su dirección postal, su teléfono, su correo electrónico, etc).

10.2. Nociónes mínimas de lenguaje SQL

El lenguaje SQL (Structured Query Language) es un lenguaje muy extendido, que permite hacer consultas a una base de datos relacional. Por eso, vamos a ver algunas de las órdenes que podemos usar para almacenar y obtener datos, y posteriormente las aplicaremos desde un programa en C# que conecte a una base de datos relacional. Cuando ya seamos capaces de guardar datos y recuperarlos, veremos alguna orden más avanzada para extraer la información o para modificarla.

10.2.1. Creando la estructura

Como primer ejemplo, crearemos una base de datos sencilla, que llamaremos "ejemplo1". Esta base de datos contendrá una única tabla, llamada "agenda", que contendrá algunos datos de cada uno de nuestros amigos. Como es nuestra primera base de datos, no pretendemos que sea perfecta, sino sencilla, así que apenas guardaremos tres datos de cada amigo: el nombre, la dirección y la edad.

Para crear la base de datos que contiene todo usaremos "create database", seguido del nombre que tendrá la base de datos:

```
create database ejemplo1;
```

En nuestro caso, nuestra base de datos almacenará una única tabla, que contendrá los datos de nuestros amigos. Por tanto, el siguiente paso será decidir qué datos concretos ("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, es habitual tener un tipo de datos llamado "VARCHAR" para referirnos a texto hasta una cierta

longitud, y varios tipos de datos numéricos de distinto tamaño. Si usamos "INT" para indicar que nos basta con un entero no muy grande, la orden necesaria para crear esta tabla sería:

```
create table personas (
    nombre varchar(20),
    direccion varchar(40),
    edad int
);
```

10.2.2. Introduciendo datos

Para introducir datos usaremos la orden "insert", e indicaremos tras la palabra "values" los valores para los campos de texto entre comillas, y los valores para campos numéricos sin comillas, así:

```
insert into personas values ('juan', 'su casa', 25);
```

Este formato nos obliga a indicar valores para todos los campos, y exactamente en el orden en que se diseñaron. Si no queremos introducir todos los datos, o queremos hacerlo en otro orden, o no recordamos con seguridad el orden, hay otra opción: detallar también en la orden "insert" los nombres de cada uno de los campos, así:

```
insert into personas
    (nombre, direccion, edad)
values (
    'pedro', 'su calle', 23
);
```

10.2.3. Mostrando datos

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

```
select * from personas;
```

que, con nuestros datos, daría como resultado (en el intérprete de comandos de algunos gestores de bases de datos, como MySQL):

nombre	direccion	edad
juan	su casa	25
pedro	su calle	23

Si queremos ver sólo ciertos campos, detallamos sus nombres, separados por comas:

```
select nombre, direccion from personas;
```

y obtendríamos

nombre	direccion
juan	su casa
pedro	su calle

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", así:

```
select nombre, direccion from personas where nombre = 'juan';
```

que nos diría el nombre y la dirección de nuestros amigos llamados "juan":

nombre	direccion
juan	su casa

A veces no querremos comparar con un texto exacto, sino sólo con parte del contenido del campo (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 conoczamos 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 llamados que viven en calles que contengan la palabra "calle", precedida por cualquier texto (%) y con cualquier texto (%) a continuación:

nombre	direccion
pedro	su calle

10.3. Acceso a bases de datos con SQLite

SQLite es un gestor de bases de datos de pequeño tamaño, que emplea el lenguaje SQL para las consultas, y del que existe una versión que se distribuye como un fichero DLL que acompañará al ejecutable de nuestro programa, o bien como un fichero en C que se podría integrar con las fuentes de nuestro programa... si usamos lenguaje C en nuestro proyecto.

En nuestro caso, para acceder a SQLite desde C#, tenemos disponible alguna adaptación de la biblioteca original. Una de ellas es `System.Data.SQLite`, que se puede descargar de <http://sqlite.phxsoftware.com/>

Con ella, los pasos a seguir para crear una base de datos y guardar información en ella serían:

- Crear una conexión a la base de datos, mediante un objeto de la clase `SQLiteConnection`, en cuyo constructor indicaremos detalles como la ruta del fichero, la versión de SQLite, y si el fichero se debe crear (`new=True`) o ya existe.
- Con un objeto de la clase `SQLiteCommand` detallaremos cuál es la orden SQL a ejecutar, y la lanzaremos con `ExecuteNonQuery`.
- Nos devolverá la cantidad de filas afectadas, que en nuestro caso, para una introducción de un dato, debería ser 1; si nos devuelve un dato menor que uno, nos indica que no se ha podido guardar correctamente.
- Finalmente cerraremos la conexión con `Close`:

Un fuente que dé estos pasos podría ser:

```
/*
/* Ejemplo en C#
/* ejemploSQLite1.cs
/*
/*
/* Ejemplo de acceso a
/* bases de datos con
/* SQLite (1)
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*
*/
using System;
using System.Data.SQLite; //Utilizamos la DLL

public class ejemploSQLite1
{
    public static void Main()
    {
        Console.WriteLine("Creando la base de datos...");

        // Creamos la conexion a la BD.
        // El Data Source contiene la ruta del archivo de la BD
        SQLiteConnection conexion =
            new SQLiteConnection
                ("Data Source=ejemplo01.sqlite;Version=3;New=True;Compress=True;");
        conexion.Open();

        // Creamos la tabla
        string creacion = "create table personas (
            + nombre varchar(20),direccion varchar(40),edad int);";
        SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
        cmd.ExecuteNonQuery();

        // E insertamos dos datos
        string insercion = "insert into personas values ('juan', 'su casa', 25);";
        cmd = new SQLiteCommand(insercion, conexion);
    }
}
```

```

int cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into personas (nombre, direccion, edad)"
    + " values ('pedro', 'su calle', 23);";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

// Finalmente, cerramos la conexión
conexion.Close();

Console.WriteLine("Creada.");
}
}

```

Deberemos compilar con la versión 2 (o superior) de la "plataforma punto Net". En Mono, esto supone emplear el compilador "gmcs" en vez de "mcs":

```
gmcs ejemploSQLite.cs /r:System.Data.SQLite.dll
```

Y para lanzar el ejecutable necesitaremos tener también la versión 2 (o superior) de la plataforma .Net, o bien lanzarlo a través de Mono:

```
mono ejemploSQLite.exe
```

Tanto el fichero "System.Data.SQLite.dll" como el "sqlite3.dll" deberán estar en la misma carpeta que nuestro ejecutable, para que éste funcione correctamente.

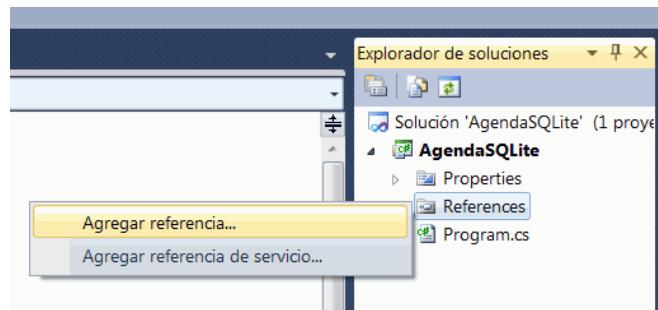
Como alternativa, también podemos crear el ejecutable usando el compilador que incorpora la "plataforma punto Net", que es un fichero llamado "csc.exe" en la carpeta de la versión de la plataforma que tengamos instalada (es frecuente tener varias). Por ejemplo, para la versión 2 haríamos

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc ejemploSQLite.cs /r:System.Data.SQLite.dll
```

Este fichero DLL es de 32 bits, de modo que si usamos una versión de Windows 64 bits no funcionaría correctamente. La solución, si compilamos desde línea de comandos, es indicar que es para "plataforma x86", añadiendo la opción "/platform:x86", así:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc ejemploSQLite.cs /r:System.Data.SQLite.dll
/platform:x86
```

Si usamos el editor de Visual Studio o de SharpDevelop, tendríamos que añadir ese fichero DLL a las "referencias" de nuestro proyecto. Lo podemos hacer desde la ventana del "Explorador de soluciones", pulsando el botón derecho sobre "References", "Añadir referencia" y escogiendo el fichero System.Data.SQLite.dll desde la pestaña "Examinar":



Al igual que si compilamos desde la línea de comandos, tanto el fichero "System.Data.SQLite.dll" como el "sqlite3.dll" deberán estar en la misma carpeta que nuestro ejecutable (típicamente "bin\debug").

Como curiosidad, esta base de datos de prueba se podría haber creado previamente desde el propio entorno de SQLite, o con algún gestor auxiliar, como la extensión de Firefox llamada SQLite Manager o como la utilidad SQLite Database Browser.

Para mostrar los datos, el primer y último paso serían casi iguales, pero no los intermedios:

- Crear una conexión a la base de datos, indicando en este caso que el fichero ya existe (new=false).
- Con un objeto de la clase SQLiteCommand detallaremos cuál es la orden SQL a ejecutar, y la lanzaremos con ExecuteReader.
- Con "Read", leeremos cada dato (devuelve un bool que indica si se ha conseguido leer correctamente), y accederemos a los campos de cada dato como parte de un array: dato[0] será el primer campo, dato[1] será el segundo y así sucesivamente.
- Finalmente cerraremos la conexión con Close:

```
/*
 * Ejemplo en C#
 * ejemploSQLite2.cs
 *
 * Ejemplo de acceso a
 * bases de datos con
 * SQLite (2)
 *
 * Introducción a C#,
 * Nacho Cabanes
 */
```

```
using System;
using System.Data.SQLite;

public class ejemploSQLite2
{

    public static void Main()
{
```

```

// Creamos la conexión a la BD
// El Data Source contiene la ruta del archivo de la BD
SQLiteConnection conexion =
    new SQLiteConnection
    ("Data Source=ejemplo01.sqlite;Version=3;New=False;Compress=True;");
conexion.Open();

// Lanzamos la consulta y preparamos la estructura para leer datos
string consulta = "select * from personas";
SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
SQLiteDataReader datos = cmd.ExecuteReader();

// Leemos los datos de forma repetitiva
while (datos.Read())
{
    string nombre = Convert.ToString(datos[0]);
    int edad = Convert.ToInt32(datos[2]);
    // Y los mostramos
    System.Console.WriteLine("Nombre: {0}, edad: {1}",
        nombre, edad);
}

// Finalmente, cerramos la conexión
conexion.Close();
}

```

Que mostraría:

Nombre: juan, edad: 25
 Nombre: pedro, edad: 23

Nota: en este segundo ejemplo, mostrábamos dato[0] (el primer dato, el nombre) y dato [2] (el tercer dato, la edad) pero no el dato intermedio, dato[1] (la dirección).

Ejercicios propuestos:

- **(10.3.1)** Crea una versión de la base de datos de ficheros (ejemplo 46) que realmente guarde en una base de datos (de SQLite) los datos que maneja. Deberá guardar todos antes de terminar cada sesión de uso, y volverlos a cargar al comienzo de la siguiente.

10.4. Un poco más de SQL: varias tablas

10.4.1. La necesidad de varias tablas

Puede haber varios motivos por los que nos interese trabajar con más de una tabla.

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, y estos dos bloques de información no deberían guardarse en una misma tabla.

Por otra parte, habrá ocasiones en que veamos 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 un ejemplo:

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 (registros), utilizásemos un código de ciudad, por ejemplo "a", gastaríamos menos espacio (en este ejemplo, 7 bytes menos en cada ficha).

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 códigoDeCiudad = "a", la ciudad es "alicante").

10.4.2. Las claves primarias

Generalmente, 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.

Llamaremos "claves primarias" a estos datos que distinguen claramente unas "fichas" (registros) de otras.

10.4.3. Enlazar varias tablas usando SQL

Vamos a crear la tabla de ciudades, que guardará el nombre de cada una de ellas y su código. Este código será el que actúe como "clave primaria", para distinguir otra ciudad. Por ejemplo, hay una ciudad llamado "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 códigos 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 permitiría 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');
```

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 $3 \times 3 = 9$ 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. 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", usaríamos "like":

```
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;
```

10.4.4. Varias tablas con SQLite desde C#

Vamos a crear un fuente de C# que ponga a prueba los ejemplos anteriores:

```
/*
 * Ejemplo en C#
 * ejemploSQLite3.cs
 *
 * Ejemplo de acceso a
 * bases de datos con
 * SQLite (3)
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;
using System.Data.SQLite;

public class ejemploSQLite3
{
    public static void Crear()
    {
        Console.WriteLine("Creando la base de datos...");

        // Creamos la conexion a la BD.
        // El Data Source contiene la ruta del archivo de la BD
        SQLiteConnection conexion =
            new SQLiteConnection
            ("Data Source=ejemplo02.sqlite;Version=3;New=True;Compress=True;");
        conexion.Open();

        // Creamos las tablas
        Console.WriteLine(" Creando la tabla de ciudades");
        string creacion = "create table ciudades ( "
            + " codigo varchar(3), nombre varchar(30),"
            + " primary key (codigo));";
```

```

SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
cmd.ExecuteNonQuery();

Console.WriteLine(" Creando la tabla de personas");
creacion = "create table personas ( "
    + " nombre varchar(20), direccion varchar(40),"
    + " edad decimal(3), codciudad varchar(3));";
cmd = new SQLiteCommand(creacion, conexion);
cmd.ExecuteNonQuery();

// E insertamos datos
Console.WriteLine(" Introduciendo ciudades");
string insercion = "insert into ciudades values "
    +"('a', 'alicante');";
cmd = new SQLiteCommand(insercion, conexion);
int cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into ciudades values "
    +"('b', 'barcelona');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into ciudades values "
    +"('m', 'madrid');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

Console.WriteLine(" Introduciendo personas");
insercion = "insert into personas values "
    +"('juan', 'su casa', 25, 'a');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into personas values "
    +"('pedro', 'su calle', 23, 'm');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into personas values "
    +"('alberto', 'calle uno', 22, 'b');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

// Finalmente, cerramos la conexión
conexion.Close();

Console.WriteLine("Base de datos creada.");

```

```

}

public static void Mostrar()
{
    // Creamos la conexión a la BD
    // El Data Source contiene la ruta del archivo de la BD
    SQLiteConnection conexion =
        new SQLiteConnection
            ("Data Source=ejemplo02.sqlite;Version=3;New=False;Compress=True;");
    conexion.Open();

    // Lanzamos la consulta y preparamos la estructura para leer datos
    string consulta = "select personas.nombre, direccion, ciudades.nombre "
        +"from personas, ciudades where personas.codciudad = ciudades.codigo;";
    SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
    SQLiteDataReader datos = cmd.ExecuteReader();

    Console.WriteLine("Datos:");
    // Leemos los datos de forma repetitiva
    while (datos.Read())
    {
        // Y los mostramos
        Console.WriteLine(" {0} - {1} - {2}",
            Convert.ToString(datos[0]), Convert.ToString(datos[1]),
            Convert.ToString(datos[2]));
    }

    // Finalmente, cerramos la conexión
    conexion.Close();
}

public static void Main()
{
    Crear();
    Mostrar();
}
}

```

Su resultado sería:

```

Creando la base de datos...
Creando la tabla de ciudades
Creando la tabla de personas
Introduciendo ciudades
Introduciendo personas
Base de datos creada.

Datos:
juan - su casa - alicante
pedro - su calle - madrid
alberto - calle uno - barcelona

```

Ejercicios propuestos:

- **(10.4.1)** Mejora el ejercicio 10.3.1 para que, además de el nombre del fichero y su tamaño, guarde una categoría (por ejemplo, "utilidad" o "vídeo"). Estas categorías estarán almacenadas en una segunda tabla.

10.5. Borrado y modificación de datos

Podemos **borrar** los datos que cumplen una cierta condición. La orden es "delete from", y con "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 la parte de "where", no se borrarían los datos que cumplen una condición, sino TODOS los datos.

Para **modificar** datos de una tabla, el formato habitual 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", los cambios se aplicarán a todos los registros de la tabla).

Ejercicios propuestos:

- **(10.5.1)** Crea una versión del ejercicio 10.5.1 que no guarde todos los datos al salir, sino que actualice con cada nueva modificación: inserte los nuevos datos inmediatamente, permita borrar un registro (reflejando los cambios inmediatamente) y modificar los datos de un registro (ídem).

10.6. Operaciones matemáticas con los datos

Desde SQL podemos realizar operaciones a partir de los datos antes de mostrarlos. Por ejemplo, podemos mostrar cuál 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

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 cuantas personas tienen una dirección que comience por la letra "s", así:

```
select count(*) from personas where direccion like 's%';
```

10.7 Grupos

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 de datos. Por ejemplo: 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 una valor agrupado (count, sum, avg, ...) para cada uno de esos grupos. Por ejemplo, en nuestra tabla "personas", podríamos saber cuantas 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

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

En el lenguaje SQL existe mucho más que lo que hemos visto aquí, pero para nuestro uso desde C# y SQLite debería ser suficiente.

Ejercicios propuestos:

- **(10.7.1)** Crea una versión del ejercicio 10.4.1 que permita saber cuántos ficheros hay pertenecientes a cada categoría.

10.8 Un ejemplo completo con C# y SQLite

Vamos a crear un pequeño ejemplo que, para una única tabla, permita añadir datos, mostrar todos ellos o buscar los que cumplen una cierta condición:

```
/*
 * Ejemplo en C#
 */
/* AgendaSQLite.cs */
/*
 */
/* Ejemplo de acceso a */
/* bases de datos con */
/* SQLite (4) */
/*
 */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
 */
```

```
using System;
using System.IO;
using System.Data.SQLite;
```