# Introducción a


por Daniel Ponsoda Montiel

Autor: Daniel Ponsoda Montiel

Segundo curso de Administración de sistemas informáticos.

Asignatura: Sistemas gestores de bases de datos.

I.E.S. San Vicente *(San Vicente del Raspeig, Alicante)*.



Versiones de este documento:

- 23 de Enero de 2008. Primera versión.

# Índice de contenido

# ¿Qué es SQLite?

SQLite es una librería compacta y autocontenida de código abierto y distribuida bajo dominio público que implementa un gestor de bases de datos SQL embebido, sin configuración y transaccional.

Los usuarios más conocidos que la utilizan actualmente en sus aplicaciones son: Adobe, Apple, Mozilla, Google, McAfee, Microsoft, Philips, Sun y Toshiba, entre otros.

## *Características de la librería*

### Compacta

Con todas las características habilitadas, el tamaño de la librería es inferior a 250Kb. Deshabilitando características opcionales, el tamaño puede quedarse por debajo de los 180Kb. Esto la hace muy apropiada para usarla en dispositivos con poca memoria, como teléfonos móviles, PDAs y reproductores MP3.

Aunque también hay una relación entre uso de memoria y velocidad. Generalmente, SQLite funcionará más rápido cuanto más memoria se le reserve.

### Autocontenida

Requiere muy poco soporte de librerias externas o del sistema operativo. Esto la hace adecuada para usarla en pequeños dispositivos que no son tan completos como los PC de escritorio.

Está escrita en ANSI-C y debería compilarse fácilmente con cualquier compilador de C estándar. Hace un uso mínimo de las librerías estandar de C. Sólo utiliza siete funciones que son: memset(), memcpy(), memcmp(), strcmp(), malloc(), free() y realloc().

Es posible configurarlo en la compilación para que use memoria estática en lugar de dinámica para no necesitar malloc(), free() y realloc(). Las funciones para el manejo de fechas requieren soporte adicional de la librería de C, pero también se pueden deshabilitar durante la compilación.

Además de la versión normal del código fuente, que incluye todo el árbol de ficheros, existe una versión (algamation) que incluye todo el fuente en un único fichero C. Para dar soporte SQLite, basta con linkar el fichero “sqlite3.c” al proyecto e incluir su correspondiente cabecera “sqlite3.h”.

## *Características de la base de datos*

- **Fichero único.** La base de datos se almacena en un único fichero, cuyo formato es multiplataforma (Es posible leer el fichero en sistemas de 32 y 64 bits o en arquitecturas big-endian y little-endian). Estas características hacen que SQLite sea popular para usarlo como formato de archivo de las aplicaciones. O dicho de otra forma: Se puede usar SQLite como sustituto de Oracle o como sustituto de fopen().
- **Manifiesto de tipado.** La mayoría de motores SQL utilizan tipado estático. Cada columna de una tabla se asocia con un tipo de datos, y solo pueden introducirse valores de un tipo particular. SQLite elimina esta restricción, y hace que el tipo de datos pueda ser una propiedad del valor en sí, y no de la columna.

- **Registros de longitud variable.** Muchos otros motores SQL, fijan una cantidad de espacio para cada todas las filas, de forma que, por ejemplo, si declaramos una columna como varchar(100), el motor reservará 100 bytes de espacio para todas las filas sin tener en cuenta la información que se guarde. SQLite, por el contrario, usa sólo la cantidad de disco que necesita para almacenar la información en una fila.
- **Seguridad de los datos.** Más de dos tercios del código están dedicados puramente a la prueba y verificación. Una aplicación automatizada ejecuta cientos de miles de pruebas empleando millones de consultas SQL. SQLite responde perfectamente a fallos de reserva de memoria, y errores de E/S de disco.

## *Características del gestor de la base de datos*

### Embebido

La mayoría de motores de bases de datos, como MySQL, Oracle o SQL Server, están implementados como un servicio (o demonio en Unix). Los programas que quieren acceder a la base de datos se comunican con el servidor usando algún tipo de protocolo para enviar peticiones y recibir resultados. Esto es lo que se conoce como una aplicación cliente-servidor.

SQLite no funciona de esta manera. Con SQLite, el proceso que quiere acceder a la base de datos, lee y escribe directamente en disco. No hay servicio intermediario. De esta manera se puede hacer una aplicación totalmente autónoma y portable.

Esto tiene ventajas y desventajas. La principal ventaja es que no debemos tener un servicio que instalar, configurar, inicializar, mantener, etc.

Por otro lado, el uso de un servidor para la base de datos provee mayor protección frente a bugs en el lado de cliente. Un fallo de segmentación en el cliente no puede afectar a la memoria que se encuentra en el servidor. Además, como el servidor es un único proceso, puede controlar mejor la concurrencia.

No obstante, una característica de SQLite es que es la única base de datos sin servidor (que el autor sepa) que permite el acceso de múltiples aplicaciones a la misma base de datos.

### No necesita configuración

Debido a que SQLite es un SGBD embebido en la aplicación, no necesita instalar ni configurar nada más aparte de la aplicación en cuestión.

### Transaccional

Una base de datos transaccional es aquella cuyos cambios y consultas son atómicos, consistentes, aislados y durables (ACID), y por tanto es capaz de realizar transacciones seguras. Las transacciones en SQLite tienen estas características, incluso cuando se interrumpen por el fallo del programa, del sistema operativo o de la alimentación del ordenador. Todos los cambios de una transacción en SQLite se hacen completamente o no se hacen.

## *Diferencias en el lenguaje SQL*

### Restricciones

- **Claves ajenas**. Pueden usarse en las tablas, pero no serán interpretadas como tales.
- **Soporte parcial de triggers**. Características no implementadas:
  - FOR EACH sólo admite FOR EACH ROW.
  - Triggers de tipo INSTEAD OF sólo están permitidos en vistas.
  - Triggers recursivos.
- **Soporte parcial de ALTER TABLE**. Sólo están permitidos RENAME TABLE y ADD COLUMN.
- **Soporte parcial de manejo de cardinalidad.** Sólo está implementado LEFT JOIN.
- **Las vistas son de sólo lectura.**
- **No existen los comandos GRANT y REVOKE.** Debido a que SQLite lee y escribe en un simple fichero, los únicos permisos de acceso que pueden aplicarse son los del fichero. Los comandos GRANT y REVOKE, usados comúnmente en aplicaciones cliente/servidor no están implementados porque carecen de sentido en un motor de base de datos embebido.

### Añadidos

- Orden **REPLACE**. Es un alias de “INSERT OR REPLACE”, que ha sido agregado para compatibilidad con MySQL.
- Una consulta puede llevar la clausula no estándar **ON CONFLICT**, que permite definir lo que se debe de hacer cuando se encuentre un problema durante la ejecución de una orden. La sintaxis es: **ON CONFLICT algoritmo**, donde “algoritmo” puede ser uno de los siguientes:
  - ROLLBACK. Deshace todo lo que se había hecho hasta ese momento en la transacción actual.
  - ABORT. Se cancela la ejecución. Los cambios realizados por comandos anteriores se conservan pero se hace un respaldo del estado anterior. Este es el comportamiento por defecto.
  - FAIL. Similar al anterior, solo que no hace respaldo de los datos anteriores.
  - IGNORE. Se ignora la orden actual y se siguen ejecutando los siguientes comandos.
  - REPLACE. Se reemplaza la fila que está causando el conflicto con los valores de la orden actual.
- Las ordenes **ATTACH** y **DETACH** permiten añadir o quitar otro fichero de base de datos a la conexión actual. Las sintaxis son:
  **ATTACH [base_de_datos] fichero.bd AS nombre_base_de_datos.**
  **DETACH [base_de_datos] nombre_base_de_datos.**
- Además, a través del interfaz de C, es posible crear nuevas instrucciones SQL personalizadas.

## *Casos en los que es adecuado el uso de SQLite*

- En **aplicaciones autónomas** que no queramos que dependan de otros procesos, como podría ser un programa para llevar la contabilidad de una pequeña/mediana empresa.
- Para su uso como **formato de almacenamiento** para cualquier aplicación.
- En **pequeños dispositivos** que no disponen de servicio SQL (PDAs, Móviles, etc.) y

con pocos recursos de memoria.
- Bases de datos internas de programas que necesiten manejar gran cantidad de **información temporal** (p. ej. videojuegos, programas de análisis, etc.).

## *Casos en los que se desaconseja el uso de SQLite*

- En **aplicaciones cliente/servidor**. Si tienes muchos programas clientes que acceden a una base de datos almacenada en un servidor, deberías considerar el uso de un motor cliente/servidor.
- En **bases de datos demasiado grandes**. El sistema de transacción de SQLite, necesita almacenar temporalmente 256 bytes de información por cada 1 Mbyte de datos en cada consulta. Si la base de datos es excesivamente grande (varios gigabites), la memoria comienza a convertirse en un problema.
- En **situaciones de alta concurrencia**. Por cada solicitud de acceso a la base de datos, el motor de SQLite, bloquea el fichero entero para evitar problemas de concurrencia. En muchos casos esto no es un problema, ya que cada proceso realiza su consulta rápidamente (sólo unos milisegundos) y termina. Pero hay aplicaciones que emplean alta concurrencia, para las cuales se debería buscar otro tipo de solución.

# Comenzando con SQLite

## *Primeros pasos*

**Descargar la librería**. Es posible obtener la última versión del código fuente o el binario precompilado para Linux, Windows o MacOSX desde la web de SQLite en http://www.sqlite.org/download.html.

En Ubuntu 7.10, si no se tiene ya instalado, se puede bajar el paquete libsqlite3-0 desde el gestor de paquetes Synaptic o tecleando 'sudo aptitude install libsqlite3-0 libsqlite3-0-dev' desde un terminal.

También nos puede interesar el paquete sqlite3, que es la interfaz de sqlite para la linea de comandos.

**Preparar un proyecto SQLite.** Para proporcionar soporte de la librería SQLite3 a un proyecto debe incluirse la cabecera en el fuente:

```
#include <sqlite3.h>
```

Y luego compilarlo con gcc o g++ de la siguiente forma:

```
g++ fuente.c -L/usr/lib -o ejecutable -lsqlite3
```

Donde “fuente.c” es el nombre del fichero fuente del programa a compilar; /usr/lib deberá indicar la ruta hacia donde está instalado el fichero libsqlite3-0.so; y “ejecutable” es el nombre del fichero ejecutable resultante.

## *SQLite para usuarios de PHP / MySQL*

Aquí se ofrece una tabla de referencia rápida de las operaciones más habituales para aquellos programadores familiarizados con PHP y MySQL:

| **Operación** | **PHP/MySQL** | **C / SQLite** |
|---|---|---|
| Conectar con el servidor | mysql_connect() | sqlite3_open() |
| Seleccionar la base de datos | mysql_select_db() | |
| Realizar una consulta | mysql_query() | sqlite3_prepare()<br>sqlite3_exec() |
| Obtener una fila de la consulta | mysql_fetch_array() | sqlite3_step() |
| Liberar el resultado | mysql_free_result() | sqlite3_finalize() |
| Cerrar la conexión con la BD | mysql_close() | sqlite3_close() |

## *Funciones de SQLite básicas para C*

**sqlite3_open()**. Abre una conexión con una base de datos
Sintaxis:

```
int sqlite3_open(const char* fichero, sqlite3 **ppBD);
```

Parámetros:
- **fichero**: El nombre del fichero que contiene la base de datos.
- **ppBD**: Puntero a la base de datos.

**sqlite3_open_v2()**. Abre una conexión con una base de datos (versión extendida).
Igual a sqlite3_open, solo que permite la especificación de opciones (flags) en la conexión.
Sintaxis:

```
int sqlite3_open_v2(const char* fichero, sqlite3 **ppBD, int flags, const char *pVFS);
```

Parámetros:
- **fichero**: El nombre del fichero que contiene la base de datos.
- **ppBD**: Puntero a la base de datos.
- **flags**: Puede contener uno de los siguientes valores:
  - SQLITE_OPEN_READONLY
  - SQLITE_OPEN_READWRITE
  - SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE
- **pVFS**: Puntero al modulo VFS (Sistema de ficheros virtual: Depende del dispositivo) que se utilizará. Si se pone NULL, se empleará el VFS por defecto.

**sqlite3_prepare()**. Prepara una orden SQL para su lectura con sqlite3_step().
Sintaxis:

```
int sqlite3_prepare(
  sqlite3 *db,
  const char *sql,
  int nByte,
  sqlite3_stmt **ppStmt,
  const char **pzTail
);
```

Parámetros:
- **db**: Puntero a la conexión con la BD sobre la que se va a ejecutar la/s orden/es SQL.
- **sql**: Cadena que contiene la/s orden/es SQL
- **nByte**: Longitud de la cadena SQL en bytes
- **ppStmt**: (Salida) aquí se devolverá un puntero al resultado de la orden SQL.
- **pzTail**: (Salida) puntero a la parte no usada de la orden SQL. Esta función sólo ejecuta la primera orden SQL de la variable sql que se pasa por parámetro. De esta forma, si hay más de una orden, pzTail, apuntará al comienzo de la siguiente.

Valor de retorno:
- SQLITE_OK. Si todo ha ido bien.
- Codigo de error. Si ha fallado algo.

**sqlite3_step()**. Esta función se llamará una vez por cada fila que queramos obtener del resultado de una consulta realizada con sqlite3_prepare(). Con ella obtenemos los datos de una fila.
Sintaxis:

```
int sqlite3_step(sqlite3_stmt *);
```

Parámetros:
- **stmt**: Puntero al resultado de la consulta SQL que genera sqlite3_prepare().

**sqlite3_finalize()**. Libera la memoria ocupada por el resultado de una orden SQL cuando ya no es necesaria.

Sintaxis:
```
int sqlite3_finalize(sqlite3_stmt *);
```

Parámetros:
- **stmt**: Puntero al resultado de la consulta SQL que genera sqlite3_prepare().

**sqlite3_close()**. Cierra una conexión con una base de datos y libera la memoria ocupada por el objeto sqlite3 que se reservó al llamar a sqlite3_open().

Sintaxis:
```
int sqlite3_close(sqlite3 *);
```

Parámetros:
- **sqlite3**: Puntero a la conexión con la BD que se va a cerrar.

**sqlite3_exec()**. Evalúa y ejecuta una o varias ordenes SQL en la cadena pasada por parámetro.

Sintaxis:
```
int sqlite3_exec(
  sqlite3*,
  const char *sql,
  int (*callback)(void*, int, char**, char**),
  void*,
  char **errmsg
);
```

Parámetros:
- **sqlite3**: Puntero a la conexión con la BD sobre la que se va a ejecutar la orden SQL.
- **sql**: Cadena que contiene la/s orden/es SQL
- **callback**: Si una o más ordenes especificadas en el segundo parámetro son consultas, la función indicada en este parámetro será invocada por cada fila del resultado de la consulta.
- **void***: Puntero arbitrario que se pasará como primer parámetro a la función “callback”.
- **errmsg**: Si se devuelve un error, el mensaje es escrito en este parametro.

Valor de retorno:
- SQLITE_OK. Si todo ha ido bien.

## *Programa de ejemplo en C/C++*

El siguiente código crea una base de datos con una tabla que almacena el nombre y la edad de alumnos:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sqlite3.h>  /* Incluimos la cabecera de sqlite */

using namespace std;

int main(int argc, char *argv[]){
  sqlite3 *db;                /* Puntero a la base de datos */
  sqlite3_stmt *resultado;    /* Puntero a los resultados de las consultas */
  int msg;                    /* Valor de retorno para las instrucciones */
  string orden;               /* Ordenes SQL */
  const char* siguiente;      /* Puntero a la siguiente orden dentro de una lista SQL */
  char* error;                /* Mensaje de error de una orden */

  /* Crear un fichero prueba.bd con la base de datos */
  msg = sqlite3_open("prueba.bd",&db);
  if (msg!=SQLITE_OK) { cout << "Error al crear la base de datos\n" << endl; exit(1); }

  /* Crear una tabla con valores básicos */
  orden = "DROP TABLE IF EXISTS alumnos;";
  orden+= "CREATE TABLE alumnos (nombre VARCHAR(50), edad NUMERIC(3));";
  orden+= "INSERT INTO alumnos values('Daniel Ponsoda',28);";
  orden+= "INSERT INTO alumnos values('Valentin Carretero',22);";
  orden+= "INSERT INTO alumnos values('Omar Marin',26);";
  msg = sqlite3_exec(db,orden.c_str(),NULL,NULL,&error);
  if (msg!=SQLITE_OK) { cout << error << endl; exit(2); }

  /* Preparar una consulta */
  orden = "SELECT * FROM alumnos ORDER BY nombre;";
  msg = sqlite3_prepare(db,orden.c_str(),orden.length(),&resultado,&siguiente);
  if (msg!=SQLITE_OK) { cout << "Error en la consulta" << endl; exit(3); }

  /* Leer la informacion fila a fila */
  while (sqlite3_step(resultado)==SQLITE_ROW){
    cout << sqlite3_column_text(resultado, 0) << " | ";
    cout << sqlite3_column_int (resultado, 1) << endl;
  }

  /* Cerrar la base de datos */
  sqlite3_close(db);
  return 0;
}
```

Compilación y resultado:

```
  $ g++ -Wall bd.c -L/usr/lib -o bd -lsqlite3
  $ ./bd

  Daniel Ponsoda | 28
  Omar Marin | 26
  Valentin Carretero | 22
```

# Glosario

**Aislamiento.** Asegura que una operación no puede ver ni influir a otras.

**Atomicidad.** Es la propiedad de los SGBD de garantizar que las tareas de una transacción, se realizan todas o ninguna.

**Consistencia.** Garantiza que sólo se ejecutan las operaciones que no rompen la integridad ni otras reglas de la BD.

**Durabilidad.** Una vez realizada una operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

**Embebido.** En un sistema gestor de bases de datos, se refiere a que está contenido dentro de la aplicación y que ésta no depende de un servidor.

# Enlaces

*Nota: Todo el siguiente contenido está en inglés.*

Pagina web oficial - http://www.sqlite.org
Tutorial muy extenso – souptonuts.sourceforge.net/readme_sqlite_tutorial.html
Breve introducción sobre SQLite. - http://en.wikipedia.org/wiki/Sqlite
Extensión de SQLite para PHP – http://www.php.net/sqlite
Extensión de SQLite para Python – http://pysqlite.org
Extensión de SQLite para C++ - http://www.int64.org/sqlite.html
Sencillo gestor de consultas en modo gráfico - http://sqlitebrowser.sourceforge.net
Administrador más completo - http://sqliteadmin.orbmu2k.de/