# Conceptos avanzados

Tiempo estimado: 10min

Una vez sabemos cómo abrir el archivo principal de la base de datos y cómo realizar consultas, ya sólo queda ver aspectos avanzados. El objeto de esta lección.

La lección comienza mostrando cómo trabajar con bases de datos en memoria. A continuación, cómo usar conexiones cacheadas para reducir los recursos consumidos por la aplicación. Después, se presenta los modos de ejecución: serializado y paralelizado. Y finalmente, cómo cargar extensiones.

Al finalizar la lección, el estudiante sabrá:

- Cómo crear una base de datos en memoria.
- Cómo cachear conexiones.
- Cómo acceder a conexiones cacheadas.
- Cómo serializar comandos SQL.
- Cómo paralelizar comandos SQL.
- Cómo cargar extensiones.

## Bases de datos en memoria

SQLite permite la creación de bases de datos en memoria, recordemos que una base de datos en memoria (*in-memory database*) nos es más que aquella cuyos datos se encuentran en memoria y no en disco. Estas bases de datos se crean a nivel de conexión, no es posible abrir dos conexiones contra la misma base de datos en memoria. Una vez se cierra la conexión, formal o informalmente, la base de datos se pierde y con ello todos sus datos.

### Apertura de una conexión a base de datos en memoria

Si deseamos abrir una conexión a una *nueva* base de datos en memoria, hay que utilizar también la clase Database, indicando como nombre de archivo el texto :memory:.

Ejemplo:

```
var db = new sqlite.Database(":memory:")
```

También es posible crear una base de datos en memoria como una base de datos secundaria, pero esto no es específico del *driver*, sino de SQLite. Para ello, basta con agregar la base de datos a la conexión mediante la sentencia ATTACH DATABASE:

```
var db = new sqlite.Database("test.db")
db.run("ATTACH DATABASE ':memory:' AS sec")
```

### Copias de seguridad

Actualmente, el *driver* sqlite3 no permite realizar copias de seguridad mediante la API interna proporcionada por la biblioteca de SQLite. No es posible hacer una copia de seguridad en caliente de una base de datos en memoria, ni tampoco realizar una restauración de una base de datos a una base de datos en memoria. Hasta que el *driver* no proporcione esta funcionalidad, el uso de bases de datos en memoria es muy restringido. Una base de datos en memoria perderá su contenido cuando se cierre formal o informalmente la conexión.

## Conexiones cacheadas

Una conexión cacheada (*cached connection*) es una conexión almacenada internamente por el *driver* y que devuelve siempre que se solicita una nueva conexión contra la base de datos. El *driver* mantiene un registro de las conexiones abiertas. Cada vez que se solicita una conexión, comprueba si ya tiene una cacheada para esa base de datos y, si es así, devuelve la instancia Database cacheada, sin abrir

ninguna nueva. Si no la tiene cacheada, crea una nueva y la registra internamente para posteriores peticiones de conexión.

La idea es mantener una única conexión para cada base de datos y devolver siempre la misma para evitar los procesos de apertura y de cierre necesarios que además son muy costosos.

### Apertura de conexiones cacheadas

Este tipo de conexiones se obtienen mediante la clase sqlite3.cached.Database y sólo se pueden utilizar con bases de datos en disco, y no con bases de datos en memoria. Esta clase es idéntica a sqlite3.Database.

Ejemplo:

```
//no cacheada
var db1 = new sqlite.Database("test.db");

//cacheada
var db2 = new sqlite.cached.Database("test.db");
var db3 = new sqlite.cached.Database("test.db"); //db3 === db2
```

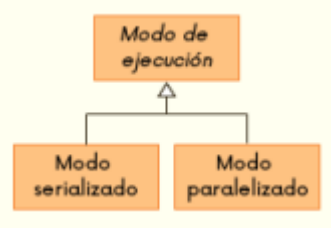
La propiedad sqlite3.cached.objects contiene la lista de las conexiones cacheadas. No se suele utilizar directamente, a menos que se dese quitar de la caché una determinada conexión.

### Cierre de conexiones cacheadas

Es importante tener clara una cosa, las conexiones cacheadas no son conexiones de *pool*. Si se cierra una conexión cacheada, se estará cerrando la conexión y, por lo tanto, cualquier variable o propiedad que la referencie ya *no* podrá ejecutar comandos SQL contra la base de datos porque su conexión se encuentra cerrada. Es más, si se solicita una nueva conexión a la base de datos, la caché devolverá la conexión cacheada y cerrada.

## Modo de ejecución

El modo de ejecución (*execution mode*) indica cómo ejecutar las instrucciones SQL contra la base de datos a través de una misma conexión: serialmente, sin concurrencia; o concurrentemente.




### Modo serializado

Cuando se utiliza el modo serializado (*serialized mode*), sólo una sentencia puede estar en ejecución a la vez, las demás deben esperar a que termine. Este modo se puede aplicar a un conjunto de sentencias o bien a toda la conexión. Se configura mediante el método serialize() de la clase Database.

#### *Modo serializado a nivel de conexión*

Para activar el modo serializado a nivel de toda la conexión, se utiliza la siguiente sobrecarga del método Database.serialize():

```
serialize()
```

Una vez fijado el modo serializado a nivel de conexión, las consultas se irán encolando y la conexión las ejecutará una a una. Hasta que no finalice la que se encuentra en curso, no desencolará la siguiente, y así sucesivamente.

#### *Modo serializado a nivel de función*

También es posible ejecutar un conjunto de sentencias dado de manera serializada sin afectar al modo de ejecución configurado a nivel de conexión. Para ello, también se utiliza el método serialize() de la clase Database:

```
serialize(callback)
```

| Parámetro | Tipo de datos | Descripción |
|---|---|---|
| callback | Function | Función que contiene las sentencias a invocar serialmente: fn(). |

Lo siguiente muestra un sencillo ejemplo ilustrativo:

```
db.serialize(function() {
  db.run("INSERT INTO Tabla VALUES(1, 2)");
  db.run("INSERT INTO Tabla VALUES(3, 4)");
});
```

Es importante tener claro que todos los comandos ejecutados *dentro* de la función adjuntada se ejecutarán en serie, una a una.

### Modo paralelizado

Bajo el modo paralelizado (*parallelized mode*), el *driver* puede ejecutar varias consultas al mismo tiempo dentro de la misma conexión. Al igual que con el modo serializado, se puede configurar a nivel de conexión o de conjunto de sentencias mediante el método Database.parallelize():

```
parallelize()
parallelize(callback)
```

| Parámetro | Tipo de datos | Descripción |
|---|---|---|
| callback | Function | Función que contiene las sentencias a invocar paralelizadamente: fn(). |

La primera sobrecarga lo configura a nivel de conexión. La segunda a nivel de función.

## Carga de extensiones

Recordemos de SQLite que la carga de extensiones (*extension load*) es el proceso mediante el cual se carga una extensión de SQLite. La cual debe estar compilada para la versión del *driver* sqlite3 que estemos usando. Se realiza mediante el método loadExtension() de la clase Database:

```
loadExtension(path)
loadExtension(path, callback)
```

| Parámetro | Tipo de datos | Descripción |
|---|---|---|
| path | String | Ruta al archivo que contiene la extensión a cargar. |
| callback | Function | Función a invocar cuando la extensión se haya cargado: fn(error). |