Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

Guía rápida de SQLite3

Indice

Pequeña, rápida y de confianza
Elige cualquiera de las tres opciones

Introducción

SQLite es un sencillo sistema de gestión de bases de datos de tipo relacional que está escrito en lenguaje C, que implementa el estándar del lenguaje de consulta SQL-92.

SQlite no funciona como otros gestores que necesitan de un servidor de base de datos ejecutándose en un proceso separado al que se le hacen peticiones. La librería SQLite es tan pequeña que se enlaza con los programas y éstos hacen llamadas directamente a los procedimientos y funciones disponibles para interactuar con las bases de datos; siendo este modo de trabajo más eficiente que el basado en comunicar peticiones a procesos externos.

Una base de datos SQLite se almacena en un sólo archivo y, siempre que sea posible, funcionará completamente en memoria para mejorar su rendimiento. Además, como no tiene dependencias externas es fácilmente portable y convertible.

SQLite tiene licencia GPL, está disponible para las plataformas más extendidas (GNU/Linux,

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

Windows, MacOSX) y cuenta con librerías y drivers para desarrollar bases de datos con los lenguajes de programación más populares (Python, Perl, Java, Ruby, PHP, C, etc.).

Instalar SQLite

En Debian/Ubuntu:
$ sudo apt-get install sqlite3 libsqlite3-dev

En Windows:
Descargar archivos .zip del apartado "**Precompiled Binaries for Windows**" de la página de descargas de SQLite, descomprimir archivos y copiar archivos obtenidos a la ruta C:\Windows\System32.

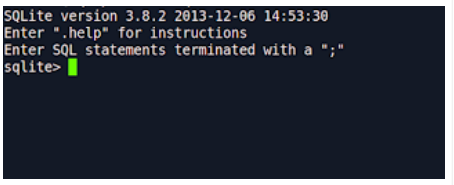
En otros sistemas: Android, Mac OS X, Windows Phone 8

Comandos del gestor

Una vez instalado el gestor de base de datos para iniciar una sesión de trabajo desde la línea de comandos, introducir:

$ sqlite3

A continuación, el sistema comenzará la sesión de trabajo mostrando la siguiente información, y quedando a la espera de recibir comandos:



Comandos más usuales:

- Salir: .exit .quit
- Crear base de datos vacía o abrir existente: .open nombrebase.db
- Listar bases de datos abiertas: .databases
- Listar tablas y vistas de una base de datos abiertai: .tables
- Listar tablas utilizando patrones: .tables "prefijo%" .tables "%sufijo"
- Listar esquema (estructura): .schema nombretabla ó .fullschema
- Listar índices: .indexes ó .indices
- Listar campos de una tabla: PRAGMA table_info("usuarios");
- Ejecutar comandos SQL almacenados en un archivo: .read archivo.sql
- Añadir base datos a sesión actual: ATTACH DATABASE "nueva.db" AS nueva;
- Ayuda: .help

Tablas. Tipos de datos

Una tabla es el objeto principal de una base de datos por ser el lugar donde se almacena la información a gestionar. Una base de datos suele contener un conjunto de tablas y por sus datos, algunas de ellas, pueden estar relacionadas entre sí.

Las tablas se utilizan para organizar la información y se componen de filas y columnas.

Un registro es cada una de las filas en que se divide la tabla y un campo es cada una de las columnas de la tabla que, normalmente, contienen datos de diferentes tipos.

Tipos de datos:

- NULL: Se refiere a los valores nulos o NULL: typeof(NULL) -> null
- INTEGER: Números enteros: typeof(100) -> integer
- REAL: Números reales: typeof(100.10) -> real
- TEXT: Texto: typeof('100') -> text
- BLOB: Binario: typeof(x'100') -> blob

Fechas:

Las fechas en **SQLITE3** se pueden almacenar en un campo como textos, números reales o números enteros.

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

Archivo

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

```
# Fechas y horas como TEXT:
# Formato: ISO8601 "YYYY-MM-DD HH:MM:SS.SSS"

CREATE TABLE IF NOT EXISTS tabla1 (fecha1 text, fecha2 text);
INSERT INTO tabla1 (fecha1, fecha2) VALUES (datetime('now'),
datetime('now', 'localtime'));
SELECT fecha1, typeof(fecha1), fecha2, typeof(fecha2) FROM tabla1;

# Fechas y horas como REAL:
# Formato: juliano

CREATE TABLE IF NOT EXISTS tabla1 (fecha1 real);
INSERT INTO tabla1 (fecha1) VALUES (julianday('now'));
SELECT fecha1 FROM tabla1;

# Fechas y horas como INTEGER:
# Formato: entero

CREATE TABLE IF NOT EXISTS tabla1 (fecha1 int);
INSERT INTO tabla1 (fecha1) VALUES (strftime('%s','now'));
Consulta valor (entero):
SELECT fecha1 FROM tabla1;
Consulta valor (fecha-hora):
SELECT datetime(fecha1, 'unixepoch') FROM tabla1;
```

Crear/Abrir Base de Datos

```
# Crear base de datos vacía (si no existe)
# o abrir base de datos (existente):

$ sqlite3 nombrebase.db

# o bien

sqlite3
.open nombrebase.db
```

Crear tablas

La sentencia **CREATE** se utiliza para crear las tablas de una base de datos. En primer lugar, a cada tabla se le asigna un nombre. Después, se define la lista de campos indicando para cada uno de ellos el tipo de dato que va a contener (**TEXT**, **INTEGER**, **REAL**, **TEXT** y **BLOB**). Además, cada campo puede incluir en su definición algunas restricciones básicas como **NOT NULL** para que no contengan valores nulos o **UNIQUE** para que no se acepte un mismo dato en diferentes filas o registros.

También, para garantizar la integridad de los datos de una base de datos pueden existir campos declarados como **PRIMARY KEY** (claves primarias) o **FOREIGN KEY** (claves externas). Ver apartados correspondientes.

```
# Ejemplo: Crear tabla "usuarios" y "dptos".

CREATE TABLE usuarios (
 id_usu INTEGER PRIMARY KEY AUTOINCREMENT,
 cta_usu TEXT UNIQUE,
 nombre TEXT NOT NULL,
 id_dpto INTEGER,
 ecorreo TEXT NOT NULL,
 FOREIGN KEY (id_dpto) REFERENCES dptos(id_dpto)
);

CREATE TABLE dptos (
 id_dpto INTEGER PRIMARY KEY AUTOINCREMENT,
 denom TEXT NOT NULL UNIQUE
);
```

Operaciones básicas con registros

INSERT

La sentencia **INSERT** permite insertar registros en una tabla.

```
# Insertar un registro:
INSERT INTO tabla1 (campo1, campo2) VALUES (dato1, dato2);

# Insertar múltiples registros:
INSERT INTO tabla1 (campo1, campo2) VALUES (dato11, dato12),
(dato21, dato22);

# Insertar registro con valores por defecto
# (o valores Null) definidos al crear tabla:
CREATE TABLE tabla1 (campo1 INTEGER DEFAULT 1,
campo2 INTEGER DEFAULT 2);
INSERT INTO tabla1 DEFAULT VALUES;

# Insertar registros con los datos de un SELECT:
INSERT INTO tabla2 SELECT campo1, campo2 FROM tabla1;
```

SELECT

La sentencia **SELECT** permite consultar los datos almacenados en una base de datos.

```
# Sintáxis de SELECT:

# SELECT DISTINCT lista_campos
# FROM lista_tablas_o_vista
#    JOIN tabla ON condición_de_unión
# WHERE filtro_a_nivel_de_fila
# ORDER BY campo
# LIMIT número OFFSET offset
# GROUP BY campo
# HAVING filtro_a_nivel_de_agrupamiento;

# Claúsulas:

# ORDER BY campo: permite ordenar la consulta por
# nombre o número de campo.

# DISTINCT. En una consulta muestra filas únicas.

# WHERE. Se utiliza para filtrar la consulta a
# nivel de filas.

# LIMIT núm. OFFSET offset. Limita núm. de filas devueltas.

# GROUP BY campo. Agrupa de filas por campo (resumen).
# Devuelve una fila por cada grupo. A cada grupo se le
# puede aplicar una función (MIN, MAX, SUM, COUNT, AVG)
# para obtener más información del grupo.

# HAVING. Aplica filtro (a una agrupación o en una
# agregación).

# INNER JOIN o LEFT JOIN. Permite consultar filas de
# múltiples tablas.

# CASE. Calcula campo a partir de una o más condiciones

# Operadores de comparación:
=, <>, !=, <, >, <= , >=

# Operadores:
# 0, 1, NULL - 0 equivale a FALSE y 1 a TRUE
# ALL, AND, OR, ANY, BETWEEN, EXISTS, IN, LIKE,
# NOT (NOT EXISTS, NOT IN, NOT BETWEEN, etc.)

# Ejemplos de SELECT:
SELECT campo1, campo2 FROM tabla1;
SELECT * FROM tabla1;
SELECT campo1, campo2 FROM tabla1 ORDER BY campo1 ASC, campo2 DESC;
SELECT campo1, campo2 FROM tabla1 ORDER BY campo1 ASC;
SELECT campo1, campo2 FROM tabla1 ORDER BY 2,1;
SELECT DISTINCT campo1 FROM tabla1;
SELECT campo1, campo2 FROM tabla1 WHERE campo1 = 100;
```

```
SELECT campo1, campo2 FROM tabla1 WHERE campo2 IN (1,2,3);
SELECT campo1, campo2 FROM tabla1 WHERE campo3 IN (SELECT campo3
FROM tabla2 WHERE campo4 = 20);
SELECT campo1, campo2, campo3 FROM tabla1 WHERE campo3
NOT IN (1, 2, 3);
SELECT campo1, campo2, campo3 FROM tabla1 WHERE campo3 LIKE 'An%';
SELECT campo1, campo2, campo3 FROM tabla1 WHERE campo1
BETWEEN 10 AND 20;
SELECT campo1, campo2 FROM tabla1 LIMIT 10;
SELECT campo1, campo2 FROM tabla1 LIMIT 10 OFFSET 10;
SELECT campo1, campo2 FROM tabla1 WHERE campo2 LIKE '%Br';
SELECT campo1, campo2 FROM tabla1 WHERE campo2 LIKE '%Br_wn%';
SELECT campo1, campo2 FROM tabla1 WHERE campo2 GLOB 'Man*';
SELECT campo1, campo2 FROM tabla1 WHERE campo2 GLOB '*Man';
SELECT campo1, campo2 FROM tabla1 WHERE campo2 GLOB '?ere*';
SELECT campo1, campo2 FROM tabla1 WHERE campo2 GLOB '*[1-9]*';
SELECT campo1, COUNT(campo2) FROM tabla1 GROUP BY campo3;
SELECT campo1, COUNT(campo2) FROM tabla1 GROUP BY campo2
HAVING campo3 = 1;
SELECT campo1, campo2 CASE WHEN campo3 < 3 THEN '1'
WHEN campo3 > 3 AND campo3 < 6 THEN '2'
ELSE '3' END valor FROM tabla1;
```

```
# Seleccionar registros coincidentes de ambas
# tablas y los que no coincidan de tabla1:
SELECT campo1, tabla1.campo2, tabla2.campo2, campo3 FROM tabla1
LEFT JOIN tabla2 ON tabla2.campo2 = tabla1.campo2

# Seleccionar registros coincidentes de ambas tablas:
SELECT campo1, tabla1.campo2, tabla2.campo2, campo3 FROM tabla1
INNER JOIN tabla2 ON tabla2.campo2 = tabla1.campo2

# Seleccionar registros coincidentes y no
# coincidentes de ambas tablas:
SELECT * FROM tabla1 FULL OUTER JOIN tabla2 ON
tabla1.campo1 = tabla2.campo1;

# Combinar dos o más conjuntos de resultados:
SELECT campo1, campo2 FROM tabla1 CROSS JOIN tabla2 ORDER BY campo2;

# Combinar conjuntos de resultados de dos o
# más consultas en un único conjunto de
# resultados (UNION y UNION ALL: UNION elimina
# filas duplicadas y UNION ALL no):
consulta_1 UNION [ALL] consulta_2 UNION [ALL] consulta_3 ...;
SELECT campo1, campo2 FROM tabla1 UNION SELECT campo1, campo2
FROM tabla2 ORDER BY campo1, campo2;
```

UPDATE

La sentencia **UPDATE** se utiliza para actualizar datos en los registros o filas de una tabla.

```
# Actualizar un campo en registros que cumplan una condición:
UPDATE tabla1 SET campo1 = 'dato1' WHERE campo2 = 3;

# Actualizar varios campos en registros que cumplan una condición:
UPDATE tabla1 SET campo1 = 'dato1', campo2 = 'dato2' WHERE campo3 = 3;

# Actualizar limitando el número de registros:
UPDATE tabla1 SET campo1 = 'dato1' ORDER BY campo2 LIMIT 1;

# En este caso se limita la actualización a
# 1 registro, que será el primero que resulte de
# la ordenación que establece la claúsula ORDER BY

# Actualizar todos los registros:
UPDATE tabla1 SET campo1 = 'dato1';
```

DELETE

La sentencia **DELETE** sirve para borrar registros de las tablas de una base de datos.

```
# Borrar los registros que cumplan una condición:
DELETE FROM tabla1 WHERE campo1 > 10;

# Borrar todos los registros:
DELETE FROM tabla1;

# Borrar limitando el número de registros:
DELETE FROM tabla1 WHERE campo1 > 10 ORDER BY campo2 LIMIT 2;
```

REPLACE

Se utiliza para insertar una nueva fila o para reemplazar (borrar una fila existente e insertar una nueva) en una tabla.

```
# Cuando sucede una excepción por una
# restricción UNIQUE KEY o PRIMARY:

# En primer lugar, se elimina la fila existente
# que produce la excepción.
# Y después, se inserta la nueva fila.

# Ejemplo

# tabla tabla1:
# campo1, campo2, campo3
# 1, cadena1, 120000
# 2, cadena2, 100000
# 3, cadena3, 150000

# Insertar un nuevo registro con REPLACE
# (no se produce excepción):

# Normalmente, se crea un índice con clave única
# para asegurar que no hay duplicados.

CREATE UNIQUE INDEX indice1 ON tabla1 (campo2);

# En el siguiente ejemplo como no existe un
# registro que contenga la clave "cadena4" REPLACE
# insertará un nuevo registro

REPLACE INTO tabla1 (campo2, campo3) VALUES ('cadena4', 140000);

# campo1, campo2, campo3
# 1, cadena1, 120000
# 2, cadena2, 100000
# 3, cadena3, 150000
# 4, cadena4, 140000

# Actualizar registro con REPLACE
# (se produce excepción):

# En el ejemplo que sigue como existe un
# registro con la clave "cadena1", REPLACE
# borra el registro actual e inserta uno nuevo:

REPLACE INTO tabla1 (campo2, campo3) VALUES ('cadena1', 170000);

# campo1, campo2, campo3
# 2, cadena2, 100000
# 3, cadena3, 150000
# 4, cadena4, 140000
# 5, cadena1, 170000
```

Declarar una clave primaria (Primary Key)

Una **clave primaria** es un campo o combinación de campos que identifica de forma única a cada fila de una tabla.

```
# Declarar clave primaria de campo único:

CREATE TABLE tabla1 (
 campo1 INTEGER PRIMARY KEY,
 campo2 TEXT NOT NULL
);
```

```
# En SQL estándar la columna de clave principal
# no debe contener valores NULL. Esto significa
# que la columna de clave principal tiene una
# restricción NOT NULL implícita. Sin embargo,
# para hacer la versión actual de SQLite compatible
# con las versiones anteriores, SQLite permite
# que la columna de clave principal pueda
# contener valores nulos.

# Declarar clave primaria de múltiples campos:

CREATE TABLE tabla1 (
 campo1 INTEGER,
 campo2 INTEGER,
 PRIMARY KEY (campo1, campo2),
 ...
);

# La columna rowid:

# Cuando se crea una tabla sin especificar
# la opción WITHOUT ROWID, SQLite añade una
# columna llamada rowid que almacena el
# número de fila (entero de 64 bits).
# La columna rowid es una clave que identifica
# de forma unívoca cada fila dentro de la tabla.

SELECT rowid, campo1, campo2 FROM tabla1;

# Añadir clave primaria a una tabla existente:

# SQLite no permite utilizar la sentencia
# ALTER TABLE para agregar una clave principal
# a una tabla existente como puede hacerse
# en MySQL o ProstgreSQL.

# Sin embargo, se puede renombrar la tabla,
# crear una nueva tabla con la misma estructura
# y después importar la tabla renombrada:

PRAGMA foreign_keys=off;
BEGIN TRANSACTION;
ALTER TABLE tabla1 RENAME TO tabla1_anterior;
CREATE TABLE tabla1 (
 campo1 integer PRIMARY KEY,
 campo2 text NOT NULL
);
INSERT INTO tabla1 SELECT * FROM tabla1_anterior;
DROP TABLE tabla1_anterior;
COMMIT;
PRAGMA foreign_keys=on;
PRAGMA table_info(tabla1);

# Definir una clave primaria autoincrementada:

CREATE TABLE tabla1 (
 campo1 INTEGER PRIMARY KEY AUTOINCREMENT,
 campo2 TEXT NOT NULL UNIQUE
);

INSERT INTO tabla1 (campo2) VALUES ('dato1'), ('dato2');
```

Declarar una clave externa (Foreign Key)

Una **clave externa** es un campo (o varios) que señalan la clave primaria de otra tabla. El propósito de la clave externa es asegurar la integridad referencial de los datos. La **integridad referencial** es un sistema de reglas que utilizan la mayoría de las bases de datos relacionales para asegurar que los registros de las tablas relacionadas son válidos; y para que no se borren o cambien datos de forma accidental produciendo errores (de integridad).

```
# Crear tablas con claves externas:

# Cuando se crea una tabla la claúsula
# FOREIGN KEYS se utiliza para declarar
# la clave externa.

# En la siguiente ejemplo la clave externa
```

```
# "campo3" de "tabla1" está vinculada con
# la clave primaria "campo1" de "tabla2":

CREATE TABLE tabla1 (
 campo1 INTEGER PRIMARY KEY AUTOINCREMENT,
 campo2 TEXT NOT NULL UNIQUE,
 campo3 INTEGER,
 FOREIGN KEY (campo3) REFERENCES tabla2(campo1)
);

CREATE TABLE tabla2 (
 campo1 INTEGER PRIMARY KEY AUTOINCREMENT,
 campo2 TEXT UNIQUE,
 campo3 TEXT NOT NULL,
);

# Declarar acciones de clave externa:

# Las acciones se utilizan para controlar
# qué hacer en caso de borrado o actualización
# de una clave externa.

FOREIGN KEY (clave_externa)
 REFERENCES tabla_padre(clave_padre)
 ON UPDATE acción
 ON DELETE acción;

# Ejemplo:

CREATE TABLE tabla1 (
 campo1 INTEGER PRIMARY KEY AUTOINCREMENT,
 campo2 TEXT UNIQUE,
 campo3 TEXT NOT NULL,
 campo4 INTEGER,
 FOREIGN KEY (campo4) REFERENCES tabla2(campo1) ON DELETE CASCADE ON
UPDATE NO ACTION
);

# Acciones:

# SET NULL. Cuando existen cambios en clave
# primaria, eliminación o actualización,
# las claves secundarias (externas) correspondientes
# de todas las filas se establecen con el valor NULL.

# SET DEFAULT. En este caso las claves externas
# se establecen con el valor por defecto definido
# en el momento de la creación de la tabla.

# RESTRICT. Establece el comportamiento por defecto,
# aplicando las restricciones de clave externa.

# CASCADE. Los cambios en la clave primaria se
# propagarán a todas las claves secundarias que
# dependan de ella.

# NO ACTION. No se realizará ninguna acción.

# Verificar si SQLite soporta restricciones
# de clave externa (Las restricciones de clave
# externa son soportadas a partir de la versión 3.6.19)

# Comprobar si están activas las restricciones:
# PRAGMA foreign_keys;

# Desactivar restricciones:
# PRAGMA foreign_keys=OFF;

# Activar restricciones de clave externa:
# PRAGMA foreign_keys=ON;
```

Cambiar la estructura de una tabla (ALTER TABLE)

```
# Renombrar una tabla:

ALTER TABLE tabla1 RENAME TO tabla1_anterior;
```

```
# Añadir campo o columna:

ALTER TABLE tabla1 ADD COLUMN campo4 INTEGER;

# La nueva columna no puede ser UNIQUE o
# PRIMARY KEY. Si la nueva columna tiene una
# restricción NOT NULL es necesario especificar
# un valor predeterminado que no sea el
# valor NULL:

CREATE TABLE tabla1 (... campo4 INTEGER DEFAULT 1, ...);
```

Borrar una tabla

```
# Borrar una tabla:

DROP TABLE tabla1;

# Si hay relaciones con otras tablas y las
# restricciones están activas no será posible
# borrar la tabla.
# Aparecerá el mensaje: "constraint failed"

# Se podría borrar desactivando antes las
# restricciones. Después, el campo que relacionaba
# la tabla maestra con la auxiliar se puede
# establecer con el valor NULL:

PRAGMA foreign_keys = OFF;
DROP TABLE tabla_auxiliar;
UPDATE tabla_maestra SET campo4 = NULL;
PRAGMA foreign_keys = ON;
```

Operaciones con Vistas

Una **vista** es una consulta que se presenta como una tabla (virtual) que se construye a partir de los datos de una o más tablas relacionadas de una base de datos.

Las vistas tienen la misma estructura que una tabla normal (filas y columnas) con la diferencia de que sólo almacenan la definición de una consulta, no los datos.

```
# A partir de las siguientes tablas:

tabla1: nombre, codigo
tabla2: codigo, descripcion

# Crear vista vinculando registros de ambas
# tablas por el campo "codigo"

CREATE VIEW IF NOT EXISTS nombre_vista AS
SELECT descripcion, tabla1.codigo, tabla2.codigo, nombre
FROM tabla1
INNER JOIN tabla2 ON tabla2.codigo = tabla1.codigo;

# Consultar todos los registros de una vista:

SELECT * FROM nombre_vista;

# Consultar algunos registros de una vista:

SELECT * FROM nombre_vista WHERE nombre = "Carlos";

# Borrar vista:

DROP VIEW IF EXISTS nombre_vista;
```

Operaciones con Índices

Un **índice** permite consultar datos con más rapidez, acelerar la operación de ordenación y hacer cumplir las restricciones únicas.

Cada índice debe estar asociado con una tabla específica. Un índice se compone de una o más columnas pero todas las columnas de un índice deben estar en la misma tabla. Una tabla puede tener varios índices.

```
# Crear índices:

CREATE INDEX nombre_indice ON tabla1(campo_indice);

# Crear índices con claves únicas:

# La opción UNIQUE es opcional y sirve para
# asegurarse de que el valor de la columna
# es único:

CREATE UNIQUE INDEX nombre_indice ON tabla1(campo2);

# Comprobar si se está utilizando el índice:

EXPLAIN QUERY PLAN SELECT * FROM tabla1 WHERE campo2 = "as@as.es";

# Crear indice multicampo:

CREATE INDEX indice1 ON tabla1 (campo2, campo3);

# Si se consulta la tabla por alguno o
# los dos campos, con la cláusula WHERE,
# se utilizará el índice de múltiples campos.

# Borrar índice:

DROP INDEX IF EXISTS indice1;

# Índices basados en expresiones:

# Además de los índices normales SQLite
# permite construir un índice basado en
# expresiones en las cuales se utilizan los
# campos de una tabla. Estos índice se usan
# para mejorar el rendimiento de las consultas.

# Crear índice con la longitud de un campo:
CREATE INDEX indice_longitud ON tabla1(LENGTH(campo2));

# Crear índice con el resultado de multiplicar dos campos:
CREATE INDEX indice_total ON tabla1(campo3*campo4);
```

Operaciones con Triggers

Un **trigger** es un objeto de base de datos que se ejecuta automáticamente cuando se realiza una operación (**INSERT**, **UPDATE** o **DELETE**) en una tabla.

```
# Sintaxis:

CREATE TRIGGER [IF NOT EXISTS] nombre_trigger
   [BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
   ON nombre_tabla
   [WHEN condition]
BEGIN
 declaraciones;
END;

# Crear trigger para validar un campo
# antes de insertar un registro:

CREATE TRIGGER validar_dato BEFORE INSERT ON tabla1
BEGIN
 SELECT
 CASE
 WHEN NEW.campo4 NOT LIKE '%_@__%.__%' THEN
 RAISE (ABORT,'Dirección de correo incorrecta')
 END;
END;

# Crear Trigger para insertar un registro
# en una tabla cuando se modifiquen datos en otra:
```

```
CREATE TRIGGER guardar_cambios AFTER UPDATE ON tabla1
WHEN old.campo1 <> new.campo1 OR old.campo2 <> new.campo2
BEGIN
 INSERT INTO tabla_cambios ( old_campo1, new_campo1, old_campo2,
 new_campo2, tipo_operacion, fecha_cambio )
VALUES
 ( old.campo1, new.campo1, old.campo2, new.campo2, 'UPDATE',
   DATETIME('NOW'));
END;

# Borrar Trigger:

DROP TRIGGER IF EXISTS guardar_cambios;
```

Optimizar una base de datos

Las siguientes operaciones permiten optimizar una base de datos:

```
# Optimizar la base de datos abierta:

VACUUM;

# Optimización automática completa:

PRAGMA auto_vacuum = FULL;

# Optimización automática incremental:

PRAGMA auto_vacuum = INCREMENTAL;

# Desactivar modo automático de optimización:

PRAGMA auto_vacuum = NONE;
```

Garantizar fiabilidad en las transacciones

Las **transacciones** se utilizan para garantizar la integridad y fiabilidad de los datos. Cuando se inicia una transacción esta permanece abierta hasta que se confirma o deshace una operación de forma explícita. Después de iniciar la transacción se ejecutan los comandos SQL para seleccionar o actualizar datos. Finalmente, se confirman los cambios con la sentencia **COMMIT** o COMMIT TRANSACTION; o se deshacen con **ROLLBACK** o ROLLBACK TRANSACTION

```
# Iniciar transacción:

BEGIN TRANSACTION;

# Confirmar transacción:

COMMIT;

# Deshacer transacción:

ROLLBACK;

# Ejemplo:

BEGIN TRANSACTION;
UPDATE tabla1 SET campo1 = 0 WHERE campo2 = 41;
UPDATE tabla1 SET campo1 = 0 WHERE campo2 = 28;
INSERT INTO tabla1 (campo1, campo2, campo3) VALUES (10, 31, 41);
INSERT INTO tabla1 (campo1, campo2, campo3) VALUES (50, 90, 28);
COMMIT;
```

Exportar una consulta (SELECT)

```
# Exportar una consulta en columnas de
# longitud fija:

.output tabla1.txt
```

```
SELECT * FROM vista_tabla1;
.mode column

# Exportar consulta en formato CSV
# (valores separados por comas):

.output tabla1.csv
SELECT * FROM tabla1 ORDER BY campo2 LIMIT 10;
.separator ,
.mode csv
```

Salvar y restaurar una base de datos

```
# Salvar comandos SQL y datos necesarios
# para reconstruir una base de datos:

sqlite3 basedatos.bd
.output basedatos.sql
.dump

# Restaurar base de datos:

sqlite3 nueva_basedatos.bd
.read basedatos.sql

# Salvar comandos SQL y datos necesarios
# para reconstruir una tabla:

sqlite3 basedatos.bd
.output tabla1.sql
.dump tabla1

# Salvar estructura de las tablas,
# índices y vistas (sin datos):

.output basedatos_estructura.sql
.schema

# Salvar comandos SQL para insertar datos:

.mode insert
.output tabla1_datos.sql
SELECT * FROM tabla1;
```

Importar y exportar datos en formato CSV

```
# Importar datos a una tabla existente:

.mode csv
.import tabla1.csv tabla1

# Para borrar los datos de la tabla:

DELETE FROM tabla1;

# Importar datos a una tabla creando
# previamente la tabla:

DROP TABLE IF EXISTS tabla1;
CREATE TABLE tabla1 (
 campo1 INTEGER PRIMARY KEY AUTOINCREMENT,
 campo2 TEXT UNIQUE,
 campo3 TEXT NOT NULL);
.mode csv
.import tabla1.csv tabla1

# Exportar datos con cabecera:

sqlite3 basedatos.db
.output tabla1_datos_cabecera.csv
.headers on
.mode csv
```

```
SELECT * FROM tabla1;

# Exportar datos con cabecera desde la línea de comandos:

sqlite3 -header -csv basedatos.db "SELECT * FROM tabla1;" > tabla1.csv
```

Caso práctico 1: operaciones con base de datos (redlocal.db)

En este caso práctico se crea una base de datos con cuatro tablas en las que se declaran claves primarias y externas. También, se crean índices y vistas y se realizan operaciones con registros habilitando las restricciones de clave externa para garantizar la integridad de los datos.

```
# Iniciar sqlite3:

sqlite3

# Crear base de datos:

.open redlocal.db

# Crear tablas:

CREATE TABLE equipos (
 id_equipo INTEGER PRIMARY KEY AUTOINCREMENT,
 cta_equipo TEXT NOT NULL UNIQUE,
 id_usu INTEGER,
 alta TEXT NOT NULL,
 FOREIGN KEY (id_usu) REFERENCES usuarios(id_usu)
);

CREATE TABLE usuarios (
 id_usu INTEGER PRIMARY KEY AUTOINCREMENT,
 cta_usu TEXT UNIQUE,
 nombre TEXT NOT NULL,
 id_dpte INTEGER,
 ecorreo TEXT NOT NULL,
 FOREIGN KEY (id_dpte) REFERENCES dptes(id_dpte)
);

CREATE TABLE dptes (
 id_dpte INTEGER PRIMARY KEY AUTOINCREMENT,
 denom TEXT NOT NULL UNIQUE
);

CREATE TABLE accesos (
 id_acceso INTEGER PRIMARY KEY AUTOINCREMENT,
 id_equipo INTEGER,
 id_usu INTEGER,
 ult_acceso TEXT NOT NULL,
 FOREIGN KEY (id_equipo) REFERENCES equipos(id_equipo),
 FOREIGN KEY (id_usu) REFERENCES usuarios(id_usu)
);

# Habilitar las restricciones de clave externa:

PRAGMA foreign_keys=ON;

# Crear índices:

CREATE UNIQUE INDEX ind_equipos ON equipos(cta_equipo);
CREATE UNIQUE INDEX ind_usuarios ON usuarios(cta_usu);
CREATE UNIQUE INDEX ind_dptes ON dptes(denom);

# Listar esquema de una tabla e índices:

.schema dptes

# Listar índices

.indices

# Insertar algunos registros:

INSERT INTO dptes (denom) VALUES ('Tenis'), ('Baloncesto'),
('Badminton'), ('Taekwondo');

INSERT INTO usuarios (id_usu, cta_usu, nombre, id_dpte, ecorreo)
VALUES (1, "rnadal", "Rafa Nadal", 1, "rafa.nadal.lorenzo@sqlite.es");
```

```
INSERT INTO usuarios (cta_usu, nombre, id_dpto, ecorreo)
VALUES ("pgasol", "Pau Gasol", 2, "pau.gasol@sqlite.es");
INSERT INTO usuarios (cta_usu, nombre, id_dpte, ecorreo)
VALUES ("cmartin", "Carolina Martin", 3, "c.martin@sqlite.es");
INSERT INTO usuarios (cta_usu, nombre, id_dpte, ecorreo)
VALUES ("ecalvo", "Eva Calvo", 4, "eva.c@sqlite.es");
INSERT INTO usuarios (cta_usu, nombre, id_dpte, ecorreo)
VALUES ("mlopez", "Marc Lopez", 1, "marc.lopez@sqlite.es");

INSERT INTO equipos (cta_equipo, id_usu, alta)
VALUES ('PC01', 2, datetime('now', 'localtime'));
INSERT INTO equipos (cta_equipo, id_usu, alta)
VALUES ('PC02', 1, datetime('now', 'localtime'));
INSERT INTO equipos (cta_equipo, id_usu, alta)
VALUES ('PC03', 3, datetime('now', 'localtime'));

INSERT INTO accesos (id_equipo, id_usu, ult_acceso)
VALUES (1, 2, datetime('now', 'localtime'));
INSERT INTO accesos (id_equipo, id_usu, ult_acceso)
VALUES (2, 1, datetime('now', 'localtime'));

# Crear vistas

CREATE VIEW vista_equipos1 AS SELECT cta_equipo, alta,
nombre, ecorreo FROM equipos INNER JOIN usuarios
ON usuarios.id_usu = equipos.id_usu;

CREATE VIEW vista_equipos2 AS SELECT cta_equipo, alta,
nombre, ecorreo, denom FROM equipos INNER JOIN usuarios
ON usuarios.id_usu = equipos.id_usu INNER JOIN dptes
ON usuarios.id_dpte = dptes.id_dpte;

# Seleccionar registros de una vista:

SELECT * FROM vista_equipos1;
SELECT * FROM vista_equipos2;

# Listar tablas y vistas:

.tables

# Comprobar si una consulta utiliza un índice:

EXPLAIN QUERY PLAN SELECT * FROM usuarios WHERE cta_usu = "mlopez";

# 0|0|0|SEARCH TABLE usuarios USING INDEX ind_usuarios (cta_usu=?)
```

Caso práctico 2: operaciones con base de datos (contactos.db)

En este caso práctico se crea una base de datos con tres tablas en las que se declaran claves primarias y externas. También, se crean índices y vistas y se realizan operaciones con registros habilitando las restricciones de clave externa para garantizar la integridad de los datos.

También, se realizan operaciones sobre la base de datos para comprobar la acción "ON DELETE CASCADE" utilizada en la declaración de las claves externas.

```
# Crear base de datos:

sqlite3
.open contactos.db

# Crear tablas con clave primaria:

CREATE TABLE contactos (
 contacto_id integer PRIMARY KEY,
 nombre text NOT NULL,
 apellido text NOT NULL,
 email text NOT NULL UNIQUE,
 movil text NOT NULL UNIQUE
);

CREATE TABLE grupos (
 grupo_id integer PRIMARY KEY,
 nombre text NOT NULL
);

# Crear tabla con clave primaria formada por
# dos campos que son claves externas:
```

```
CREATE TABLE contactos_grupos (
 contacto_id integer,
 grupo_id integer,
 PRIMARY KEY (contacto_id, grupo_id),
 FOREIGN KEY (contacto_id) REFERENCES contactos (contacto_id)
 ON DELETE CASCADE ON UPDATE NO ACTION,
 FOREIGN KEY (grupo_id) REFERENCES grupos (grupo_id)
 ON DELETE CASCADE ON UPDATE NO ACTION
);

# Realizar operaciones sobre la base de
# datos para comprobar la acción
# "ON DELETE CASCADE" utilizada al declarar
# las claves externas:

# Insertar registros

INSERT INTO grupos (grupo_id, nombre) VALUES (1, "amigos");
INSERT INTO grupos (grupo_id, nombre) VALUES (2, "trabajo");
INSERT INTO grupos (grupo_id, nombre) VALUES (3, "asociacion");

INSERT INTO contactos (contacto_id, nombre, apellido, email, movil)
VALUES (1, "Luis", "Carranza", "lc@correo.es","123 234 345");
INSERT INTO contactos (contacto_id, nombre, apellido, email, movil)
VALUES (2, "Clara", "Campoamor", "clara@correo.es","455 555 655");
INSERT INTO contactos (contacto_id, nombre, apellido, email, movil)
VALUES (3, "Dolores", "Fuertes", "fuertes@correo.es","155 155 155");

INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (1, 1);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (2, 1);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (1, 3);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (3, 3);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (1, 2);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (2, 2);
INSERT INTO contactos_grupos (contacto_id, grupo_id) VALUES (3, 2);

# Borrar un registro de la tabla auxiliar

DELETE FROM grupos WHERE grupo_id = 3;

SELECT * FROM grupos;

# 1|amigos
# 2|trabajo

# Seleccionar todos los registros:

SELECT * FROM contactos_grupos;

# 1|1
# 2|1
# 1|2
# 2|2
# 3|2

# Todos los registros de la clave externa
# suprimida han sido borrados.

# Crear vista:

CREATE VIEW vista_contactos AS SELECT contactos.nombre AS nombrec,
apellido, email, grupos.nombre AS nombreg FROM contactos
INNER JOIN contactos_grupos ON
contactos.contacto_id = contactos_grupos.contacto_id INNER JOIN
grupos ON grupos.grupo_id = contactos_grupos.grupo_id;

# Consultar todos los registros de la vista:

SELECT * FROM vista_contactos;

# Luis|Carranza|lc@correo.es|amigos
# Clara|Campoamor|clara@correo.es|amigos
# Luis|Carranza|lc@correo.es|trabajo
# Clara|Campoamor|clara@correo.es|trabajo
# Dolores|Fuertes|fuertes@correo.es|trabajo

# Consultar todos los registros de los "amigos":

SELECT * FROM vista_contactos WHERE nombreg = "amigos";
```

```
# Luis|Carranza|lc@correo.es|amigos
# Clara|Campoamor|clara@correo.es|amigos
```

Ir al inicio de la página actual

Inicio

Suscribirse a: Entradas (Atom)