EBook Gratis

APRENDIZAJE sqlite

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlite

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sqlite

It is an unofficial and free sqlite ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sqlite.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con sqlite

Versiones

Versión	Cambios principales	Fecha de lanzamiento
3.0		2004-06-18
3.7.11	SELECCIONAR max (x), y	2012-03-20
3.8.3	CTEs	2014-02-11

Examples

Instalación

SQLite es una biblioteca de C que normalmente se compila directamente en la aplicación al descargar el código fuente de la última versión y agregar el archivo `sqlite3.c` al proyecto.

Muchos lenguajes de script (por ejemplo, Perl , Python , Ruby , etc.) y marcos (por ejemplo, Android) tienen soporte para SQLite; esto se hace con una copia integrada de la biblioteca SQLite, que no necesita instalarse por separado.

Para probar SQL, puede ser útil usar el shell de línea de comandos (`sqlite3` o `sqlite3.exe`). Ya se entrega con la mayoría de las distribuciones de Linux; en Windows, descargue los archivos binarios precompilados en el paquete `sqlite-tools` y extráigalos en algún lugar.

Documentación

SQLite ya tiene una extensa documentación , que no debe ser duplicada aquí.

Lea Empezando con sqlite en línea: https://riptutorial.com/es/sqlite/topic/1753/empezando-con-sqlite

Capítulo 2: Comandos de punto de línea de comando

Introducción

El shell de línea de comandos `sqlite3` implementa un conjunto adicional de comandos (que no están disponibles en programas que usan la biblioteca SQLite). Documentación oficial: Comandos especiales para `sqlite3`

Examples

Exportando e importando una tabla como un script SQL

Exportar una base de datos es un proceso simple de dos pasos:

```
sqlite> .output mydatabase_dump.sql
sqlite> .dump
```

Exportar una tabla es bastante similar:

```
sqlite> .output mytable_dump.sql
sqlite> .dump mytable
```

El archivo de salida debe definirse con `.output` antes de usar `.dump` ; de lo contrario, el texto acaba de salir a la pantalla.

Importar es aún más simple:

```
sqlite> .read mytable_dump.sql
```

Lea Comandos de punto de línea de comando en línea:
https://riptutorial.com/es/sqlite/topic/3789/comandos-de-punto-de-linea-de-comando

Capítulo 3: Declaraciones de PRAGMA

Observaciones

La documentación de SQLite tiene una referencia de todas las declaraciones de PRAGMA .

Examples

PRAGMAs con efectos permanentes.

La mayoría de las declaraciones de PRAGMA afectan solo a la conexión de la base de datos actual, lo que significa que deben volver a aplicarse cada vez que se abre la base de datos.

Sin embargo, los siguientes PRAGMA escriben en el archivo de base de datos y pueden ejecutarse en cualquier momento (pero en algunos casos, no dentro de una transacción):

- ID de aplicación
- journal_mode al habilitar o deshabilitar el modo WAL
- schema_version
- version_usuario
- wal_checkpoint

La siguiente configuración de PRAGMA establece las propiedades del archivo de base de datos que no se pueden cambiar después de la creación, por lo que deben ejecutarse antes de la primera escritura real en la base de datos:

- auto_vacuum (también se puede cambiar antes de VACUUM)
- codificación
- legacy_file_format
- page_size (también se puede cambiar antes de VACUUM)

Por ejemplo:

```
-- open a connection to a not-yet-existing DB file
PRAGMA page_size = 4096;
PRAGMA auto_vacuum = INCREMENTAL;
CREATE TABLE t(x);              -- database is created here, with the above settings
```

Lea Declaraciones de PRAGMA en línea: https://riptutorial.com/es/sqlite/topic/5223/declaraciones-de-pragma

Capítulo 4: sqlite3_stmt: Declaración preparada (API de C)

Observaciones

Documentación oficial: Objeto de declaración preparada.

Examples

Ejecutando una declaración

Una declaración se construye con una función como sqlite3_prepare_v2 () .

Un objeto de declaración preparado *debe* limpiarse con sqlite3_finalize () . No olvides esto en caso de error.

Si se usan parámetros , establezca sus valores con las funciones sqlite3_bind_xxx () .

La ejecución real ocurre cuando se llama a sqlite3_step () .

```
const char *sql = "INSERT INTO MyTable(ID, Name) VALUES (?, ?)";
sqlite3_stmt *stmt;
int err;

err = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
if (err != SQLITE_OK) {
    printf("prepare failed: %s\n", sqlite3_errmsg(db));
    return /* failure */;
}

sqlite3_bind_int (stmt, 1, 42);                          /* ID */
sqlite3_bind_text(stmt, 2, "Bob", -1, SQLITE_TRANSIENT); /* name */

err = sqlite3_step(stmt);
if (err != SQLITE_DONE) {
    printf("execution failed: %s\n", sqlite3_errmsg(db));
    sqlite3_finalize(stmt);
    return /* failure */;
}

sqlite3_finalize(stmt);
return /* success */;
```

Lectura de datos desde un cursor

Una consulta SELECT se ejecuta como cualquier otra declaración. Para leer los datos devueltos, llame a sqlite3_step () en un bucle. Vuelve:

- SQLITE_ROW: si los datos para la siguiente fila están disponibles, o

- SQLITE_DONE: si no hay más filas, o
- Cualquier código de error.

Si una consulta no devuelve ninguna fila, el primer paso devuelve SQLITE_DONE.

Para leer los datos de la fila actual, llame a las funciones sqlite3_column_xxx () :

```
const char *sql = "SELECT ID, Name FROM MyTable";
sqlite3_stmt *stmt;
int err;

err = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
if (err != SQLITE_OK) {
    printf("prepare failed: %s\n", sqlite3_errmsg(db));
    return /* failure */;
}

for (;;) {
    err = sqlite3_step(stmt);
    if (err != SQLITE_ROW)
        break;

    int         id   = sqlite3_column_int (stmt, 0);
    const char *name = sqlite3_column_text(stmt, 1);
    if (name == NULL)
        name = "(NULL)";
    printf("ID: %d, Name: %s\n", id, name);
}

if (err != SQLITE_DONE) {
    printf("execution failed: %s\n", sqlite3_errmsg(db));
    sqlite3_finalize(stmt);
    return /* failure */;
}

sqlite3_finalize(stmt);
return /* success */;
```

Ejecutando una declaración preparada varias veces

Después de que se ejecutó una instrucción, una llamada a sqlite3_reset () la devuelve al estado original para que pueda volver a ejecutarse.

Normalmente, mientras que la declaración se mantiene igual, los parámetros se cambian:

```
const char *sql = "INSERT INTO MyTable(ID, Name) VALUES (?, ?)";
sqlite3_stmt *stmt;
int err;

err = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
if (err != SQLITE_OK) {
    printf("prepare failed: %s\n", sqlite3_errmsg(db));
    return /* failure */;
}

for (...) {
    sqlite3_bind_int (stmt, 1, ...);   /* ID */
```

```
    sqlite3_bind_text(stmt, 2, ...);   /* name */

    err = sqlite3_step(stmt);
    if (err != SQLITE_DONE) {
        printf("execution failed: %s\n", sqlite3_errmsg(db));
        sqlite3_finalize(stmt);
        return /* failure */;
    }

    sqlite3_reset(stmt);
}

sqlite3_finalize(stmt);
return /* success */;
```

Lea sqlite3_stmt: Declaración preparada (API de C) en línea:
https://riptutorial.com/es/sqlite/topic/5456/sqlite3-stmt--declaracion-preparada--api-de-c-

Capítulo 5: Tipos de datos

Observaciones

Documentación oficial: tipos de datos en SQLite versión 3.

Examples

Función TYPEOF

```
sqlite> SELECT TYPEOF(NULL);
null
sqlite> SELECT TYPEOF(42);
integer
sqlite> SELECT TYPEOF(3.141592653589793);
real
sqlite> SELECT TYPEOF('Hello, world!');
text
sqlite> SELECT TYPEOF(X'0123456789ABCDEF');
blob
```

Usando booleanos

Para los booleanos, SQLite usa números enteros `0` y `1` :

```
sqlite> SELECT 2 + 2 = 4;
1
sqlite> SELECT 'a' = 'b';
0
sqlite> SELECT typeof('a' = 'b');
integer
```

```
> CREATE TABLE Users ( Name, IsAdmin );
> INSERT INTO Users VALUES ('root', 1);
> INSERT INTO Users VALUES ('john', 0);
> SELECT Name FROM Users WHERE IsAdmin;
root
```

Hacer cumplir los tipos de columna

SQLite utiliza la escritura dinámica e ignora los tipos de columna declarados:

```
> CREATE TABLE Test (
      Col1 INTEGER,
      Col2 VARCHAR(2),       -- length is ignored, too
      Col3 BLOB,
      Col4,                  -- no type required
      Col5 FLUFFY BUNNIES    -- use whatever you want
  );
> INSERT INTO Test VALUES (1, 1, 1, 1, 1);
```

```
> INSERT INTO Test VALUES ('xxx', 'xxx', 'xxx', 'xxx', 'xxx');
> SELECT * FROM Test;
1   1   1   1   1
xxx xxx xxx xxx xxx
```

(Sin embargo, los tipos de columna declarados se utilizan para la afinidad de tipo).

Para imponer tipos, debe agregar una restricción con la función typeof () :

```
CREATE TABLE Tab (
    Col1 TEXT   CHECK (typeof(Col1) = 'text' AND length(Col1) <= 10),
    [...]
);
```

(Si dicha columna debería ser NULLable, debe permitir explícitamente `'null'`).

Tipos de fecha / hora

SQLite no tiene un tipo de datos separado para los valores de fecha u hora.

Cuerdas ISO8601

Las palabras clave integradas `CURRENT_DATE` , `CURRENT_TIME` y `CURRENT_TIMESTAMP` devuelven cadenas en formato ISO8601:

```
> SELECT CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP;
CURRENT_DATE  CURRENT_TIME  CURRENT_TIMESTAMP
------------  ------------  -------------------
2016-07-08    12:34:56      2016-07-08 12:34:56
```

Tales valores también se entienden por todas las funciones de fecha / hora incorporadas :

```
> SELECT strftime('%Y', '2016-07-08');
2016
```

Números del día juliano

Las funciones de fecha / hora incorporadas interpretan los números como días julianos :

```
> SELECT datetime(2457578.02425926);
2016-07-08 12:34:56
```

La función `julianday()` convierte cualquier valor de fecha / hora admitido en un número de día juliano:

```
> SELECT julianday('2016-07-08 12:34:56');
2457578.02425926
```

Marcas de tiempo de Unix

Las funciones de fecha / hora incorporadas pueden interpretar números como marcas de tiempo Unix con el modificador de `unixepoch` :

```
> SELECT datetime(0, 'unixepoch');
1970-01-01 00:00:00
```

La función `strftime()` puede convertir cualquier valor de fecha / hora admitido en una marca de tiempo Unix:

```
> SELECT strftime('%s', '2016-07-08 12:34:56');
1467981296
```

formatos no compatibles

Sería posible almacenar los valores de fecha / hora en cualquier otro formato en la base de datos, pero las funciones de fecha / hora integradas no las analizarán y devolverán NULL:

```
> SELECT time('1:30:00');   -- not two digits

> SELECT datetime('8 Jul 2016');
□
```

Lea Tipos de datos en línea: https://riptutorial.com/es/sqlite/topic/5252/tipos-de-datos

Creditos

S. No	Capítulos	Contributors
1	Empezando con sqlite	CL., Community, e4c5, H. Pauwelyn
2	Comandos de punto de línea de comando	CL., e4c5, James Toomey, Lasse Vågsæther Karlsen, ravenspoint, Thinkeye
3	Declaraciones de PRAGMA	CL., springy76
4	sqlite3_stmt: Declaración preparada (API de C)	CL.
5	Tipos de datos	CL.