Kostenloses eBook

LERNEN

sqlite

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlite

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit sqlite

Versionen

Ausführung	Wesentliche Änderungen	Veröffentlichungsdatum
3,0		2004-06-18
3.7.11	SELECT max (x), y	2012-03-20
3.8.3	CTEs	2014-02-11

Examples

Installation

SQLite ist eine C- Bibliothek, die normalerweise direkt in die Anwendung kompiliert wird, indem der Quellcode der neuesten Version heruntergeladen und die Datei `sqlite3.c` zum Projekt hinzugefügt wird.

Viele Skriptsprachen (z. B. Perl , Python , Ruby usw.) und Frameworks (z. B. Android) unterstützen SQLite. Dies erfolgt mit einer integrierten Kopie der SQLite-Bibliothek, die nicht separat installiert werden muss.

Zum Testen von SQL kann es sinnvoll sein, die Befehlszeilen-Shell (`sqlite3` oder `sqlite3.exe`) zu verwenden. Bei den meisten Linux-Distributionen ist es bereits enthalten. Laden Sie unter Windows die vorkompilierten Binärdateien aus dem Paket `sqlite-tools` herunter und extrahieren Sie sie irgendwo.

Dokumentation

SQLite verfügt bereits über eine umfangreiche Dokumentation , die hier nicht dupliziert werden sollte.

Erste Schritte mit sqlite online lesen: https://riptutorial.com/de/sqlite/topic/1753/erste-schritte-mit-sqlite

Kapitel 2: Befehlszeile Punktbefehle

Einführung

Die Befehlszeilen-Shell `sqlite3` implementiert eine zusätzliche Gruppe von Befehlen (die nicht in Programmen verfügbar sind, die die SQLite-Bibliothek verwenden). Offizielle Dokumentation: Spezielle Befehle an `sqlite3`

Examples

Exportieren und Importieren einer Tabelle als SQL-Skript

Das Exportieren einer Datenbank ist ein einfacher Prozess in zwei Schritten:

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

Das Exportieren einer Tabelle ist ziemlich ähnlich:

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

Die Ausgabedatei muss vor der Verwendung von `.dump` mit `.output` definiert werden. Andernfalls wird der Text nur auf dem Bildschirm ausgegeben.

Das Importieren ist noch einfacher:

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

Befehlszeile Punktbefehle online lesen: https://riptutorial.com/de/sqlite/topic/3789/befehlszeile-punktbefehle

Kapitel 3: Datentypen

Bemerkungen

Offizielle Dokumentation: Datentypen in SQLite Version 3

Examples

TYPEOF-Funktion

```
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
```

Boolean verwenden

Für Booleans verwendet SQLite die Ganzzahlen `0` und `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
```

Erzwingen von Spaltentypen

SQLite verwendet dynamische Typisierung und ignoriert deklarierte Spaltentypen:

```
> 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
```

(Deklarierte Spaltentypen werden jedoch für die Typaffinität verwendet .)

Zur Erzwingung von Typen müssen Sie eine Einschränkung mit der Funktion typeof () hinzufügen:

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

(Wenn eine solche Spalte NULL-fähig sein soll, müssen Sie explizit `'null'` zulassen.)

Datums- / Zeittypen

SQLite hat keinen separaten Datentyp für Datums- oder Zeitwerte.

ISO8601-Zeichenketten

Die integrierten Schlüsselwörter `CURRENT_DATE` , `CURRENT_TIME` und `CURRENT_TIMESTAMP` geben Strings im ISO8601-Format zurück:

```
> 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
```

Solche Werte werden auch von allen eingebauten Datums- / Zeitfunktionen verstanden :

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

Julianische Tageszahlen

Die eingebauten Datums- / Uhrzeitfunktionen interpretieren Zahlen als Julianische Tage :

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

Die `julianday()` -Funktion konvertiert jeden unterstützten Datums- / Zeitwert in eine julianische `julianday()` :

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

Unix-Zeitstempel

Die integrierten Datums- / `unixepoch` können Zahlen mit dem `unixepoch` Modifikator als Unix-Zeitstempel `unixepoch` :

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

Die Funktion `strftime()` kann jeden unterstützten Datums- / Zeitwert in einen Unix-Zeitstempel konvertieren:

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

nicht unterstützte Formate

Es wäre möglich, Datums- / Uhrzeitwerte in einem anderen Format in der Datenbank zu speichern, aber die integrierten Datums- / Uhrzeitfunktionen werden diese nicht analysieren und NULL zurückgeben:

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

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

Datentypen online lesen: https://riptutorial.com/de/sqlite/topic/5252/datentypen

Kapitel 4: PRAGMA-Anweisungen

Bemerkungen

Die SQLite-Dokumentation enthält eine Referenz auf alle PRAGMA-Anweisungen .

Examples

PRAGMAs mit dauerhaften Auswirkungen

Die meisten PRAGMA-Anweisungen wirken sich nur auf die aktuelle Datenbankverbindung aus, dh sie müssen beim Öffnen der Datenbank erneut angewendet werden.

Die folgenden PRAGMAs schreiben jedoch in die Datenbankdatei und können jederzeit ausgeführt werden (in einigen Fällen jedoch nicht innerhalb einer Transaktion):

- application_id
- journal_mode beim Aktivieren oder Deaktivieren des WAL-Modus
- schema_version
- user_version
- wal_checkpoint

Die folgenden PRAGMA-Einstellungen legen Eigenschaften der Datenbankdatei fest, die nach der Erstellung nicht geändert werden können. Sie müssen daher vor dem ersten tatsächlichen Schreiben in die Datenbank ausgeführt werden:

- auto_vacuum (kann auch vor VACUUM geändert werden)
- Codierung
- legacy_file_format
- page_size (kann auch vor VACUUM geändert werden)

Zum Beispiel:

```
-- 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
```

PRAGMA-Anweisungen online lesen: https://riptutorial.com/de/sqlite/topic/5223/pragma-anweisungen

Kapitel 5: sqlite3_stmt: Prepared-Anweisung (C-API)

Bemerkungen

offizielle Dokumentation: Prepared Statement Object

Examples

Anweisung ausführen

Eine Anweisung wird mit einer Funktion wie sqlite3_prepare_v2 () erstellt .

Ein vorbereitetes Anweisungsobjekt *muss* mit sqlite3_finalize () aufgeräumt werden. Vergessen Sie dies im Fehlerfall nicht.

Wenn Parameter verwendet werden, legen Sie ihre Werte mit den Funktionen sqlite3_bind_xxx () fest .

Die eigentliche Ausführung findet statt, wenn sqlite3_step () aufgerufen wird.

```
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 */;
```

Daten aus einem Cursor lesen

Eine SELECT - Abfrage wird ausgeführt wie jede andere Aussage. Um die zurückgegebenen Daten zu lesen, rufen Sie sqlite3_step () in einer Schleife auf. Es gibt zurück:

- SQLITE_ROW: wenn die Daten für die nächste Zeile verfügbar sind, oder
- SQLITE_DONE: wenn keine weiteren Zeilen vorhanden sind, oder
- ein beliebiger Fehlercode

Wenn eine Abfrage keine Zeilen zurückgibt, gibt der allererste Schritt SQLITE_DONE zurück.

Um die Daten aus der aktuellen Zeile zu lesen, rufen Sie die Funktionen sqlite3_column_xxx () auf :

```
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 */;
```

Eine vorbereitete Anweisung mehrmals ausführen

Nachdem eine Anweisung ausgeführt wurde , wird sie durch einen Aufruf von sqlite3_reset () wieder in den ursprünglichen Zustand versetzt, damit sie erneut ausgeführt werden kann.

Während die Anweisung selbst gleich bleibt, werden die Parameter normalerweise geändert:

```
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 */;
```

sqlite3_stmt: Prepared-Anweisung (C-API) online lesen:
https://riptutorial.com/de/sqlite/topic/5456/sqlite3-stmt--prepared-anweisung--c-api-

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit sqlite	CL., Community, e4c5, H. Pauwelyn
2	Befehlszeile Punktbefehle	CL., e4c5, James Toomey, Lasse Vågsæther Karlsen, ravenspoint, Thinkeye
3	Datentypen	CL.
4	PRAGMA-Anweisungen	CL., springy76
5	sqlite3_stmt: Prepared-Anweisung (C-API)	CL.