



FREE eBook

LEARNING

sqlite

Unaffiliated free eBook created from
Stack Overflow contributors.

#sqlite

Table of Contents

About.....	1
Chapter 1: Getting started with sqlite.....	2
Versions.....	2
Examples.....	2
Installation.....	2
Documentation.....	2
Chapter 2: Command line dot-commands.....	3
Introduction.....	3
Examples.....	3
Exporting and importing a table as an SQL script.....	3
Chapter 3: Data types.....	4
Remarks.....	4
Examples.....	4
Date/time types.....	4
ISO8601 strings.....	4
Julian day numbers.....	4
Unix timestamps.....	4
unsupported formats.....	5
TYPEOF function.....	5
Using booleans.....	5
Enforcing column types.....	6
Chapter 4: PRAGMA Statements.....	7
Remarks.....	7
Examples.....	7
PRAGMAs with permanent effects.....	7
Chapter 5: sqlite3_stmt: Prepared Statement (C API).....	8
Remarks.....	8
Examples.....	8
Executing a prepared statement multiple times.....	8

Executing a Statement.....	8
Reading Data from a Cursor.....	9
Credits.....	11

About

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Chapter 1: Getting started with sqlite

Versions

Version	Major Changes	Release Date
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 is a [C](#) library that is typically [compiled](#) directly into the application by [downloading](#) the source code of the latest version, and adding the `sqlite3.c` file to the project.

Many script languages (e.g., [Perl](#), [Python](#), [Ruby](#), etc.) and frameworks (e.g., [Android](#)) have support for SQLite; this is done with a built-in copy of the SQLite library, which does not need to be installed separately.

For testing SQL, it might be useful to use the command-line shell (`sqlite3` or `sqlite3.exe`). It is already shipped with most Linux distributions; on Windows, [download](#) the precompiled binaries in the `sqlite-tools` package, and extract them somewhere.

Documentation

SQLite already has extensive [documentation](#), which should not be duplicated here.

Read [Getting started with sqlite online](#): <http://www.riptutorial.com/sqlite/topic/1753/getting-started-with-sqlite>

Chapter 2: Command line dot-commands

Introduction

The `sqlite3` [command-line shell](#) implements an additional set of commands (which are not available in programs that use the SQLite library). Official documentation: [Special commands to sqlite3](#)

Examples

Exporting and importing a table as an SQL script

Exporting a database is a simple two step process:

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

Exporting a table is pretty similar:

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

The output file needs to be defined with `.output` prior to using `.dump`; otherwise, the text is just output to the screen.

Importing is even simpler:

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

Read [Command line dot-commands](http://www.riptutorial.com/sqlite/topic/3789/command-line-dot-commands) online: <http://www.riptutorial.com/sqlite/topic/3789/command-line-dot-commands>

Chapter 3: Data types

Remarks

official documentation: [Datatypes In SQLite Version 3](#)

Examples

Date/time types

SQLite has no separate data type for date or time values.

ISO8601 strings

The built-in keywords `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` return strings in ISO8601 format:

```
> 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
```

Such values are also understood by all [built-in date/time functions](#):

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

Julian day numbers

The [built-in date/time functions](#) interpret numbers as [Julian days](#):

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

The `julianday()` function converts any supported date/time value into a Julian day number:

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

Unix timestamps

The [built-in date/time functions](#) can interpret numbers as [Unix timestamps](#) with the `unixepoch`

modifier:

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

The `strftime()` function can convert any supported date/time value into a Unix timestamp:

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

unsupported formats

It would be possible to store date/time values in any other format in the database, but the built-in date/time functions will not parse them, and return NULL:

```
> SELECT time('1:30:00');    -- not two digits
> SELECT datetime('8 Jul 2016');
[]
```

TYPEOF function

```
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
```

Using booleans

For booleans, SQLite uses integers 0 and 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
```


Enforcing column types

SQLite uses [dynamic typing](#) and ignores declared column types:

```
> 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
```

(However, declared column types are used for [type affinity](#).)

To enforce types, you have to add a constraint with the [typeof\(\)](#) function:

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

(If such a column should be NULLable, you have to explicitly allow 'null'.)

Read Data types online: <http://www.riptutorial.com/sqlite/topic/5252/data-types>

Chapter 4: PRAGMA Statements

Remarks

The SQLite documentation has a [reference of all PRAGMA statements](#).

Examples

PRAGMAs with permanent effects

Most PRAGMA statements affect only the current database connection, which means that they have to be re-applied whenever the database has been opened.

However, the following PRAGMAs write to the database file, and can be executed at any time (but in some cases, not inside a transaction):

- [application_id](#)
- [journal_mode](#) when enabling or disabling [WAL mode](#)
- [schema_version](#)
- [user_version](#)
- [wal_checkpoint](#)

The following PRAGMA settings set properties of the database file which cannot be changed after creation, so they must be executed before the first actual write to the database:

- [auto_vacuum](#) (can also be changed before [VACUUM](#))
- [encoding](#)
- [legacy_file_format](#)
- [page_size](#) (can also be changed before [VACUUM](#))

For example:

```
-- 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
```

Read [PRAGMA Statements online](http://www.riptutorial.com/sqlite/topic/5223/pragma-statements): <http://www.riptutorial.com/sqlite/topic/5223/pragma-statements>

Chapter 5: sqlite3_stmt: Prepared Statement (C API)

Remarks

official documentation: [Prepared Statement Object](#)

Examples

Executing a prepared statement multiple times

After a statement was [executed](#), a call to [sqlite3_reset\(\)](#) brings it back into the original state so that it can be re-executed.

Typically, while the statement itself stays the same, the parameters are changed:

```
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 */;
```

Executing a Statement

A statement is constructed with a function such as [sqlite3_prepare_v2\(\)](#).

A prepared statement object *must* be cleaned up with [sqlite3_finalize\(\)](#). Do not forget this in case of an error.

If [parameters](#) are used, set their values with the [sqlite3_bind_xxx\(\)](#) functions.

The actual execution happens when [sqlite3_step\(\)](#) is called.

```
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 */;
```

Reading Data from a Cursor

A SELECT query is [executed](#) like any other statement. To read the returned data, call [sqlite3_step\(\)](#) in a loop. It returns:

- **SQLITE_ROW**: if the data for the next row is available, or
- **SQLITE_DONE**: if there are no more rows, or
- any error code.

If a query does not return any rows, the very first step returns **SQLITE_DONE**.

To read the data from the current row, call the [sqlite3_column_xxx\(\)](#) functions:

```
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 */;
```

Read `sqlite3_stmt: Prepared Statement (C API)` online:

<http://www.riptutorial.com/sqlite/topic/5456/sqlite3-stmt--prepared-statement--c-api->

Credits

S. No	Chapters	Contributors
1	Getting started with sqlite	CL. , Community , e4c5 , H. Pauwelyn
2	Command line dot-commands	CL. , e4c5 , James Toomey , Lasse Vågsæther Karlsen , ravenspoint , Thinkeye
3	Data types	CL.
4	PRAGMA Statements	CL. , springy76
5	sqlite3_stmt: Prepared Statement (C API)	CL.