

SQLite Tutorial

This **SQLite tutorial** teaches you everything you need to know to start using SQLite effectively. You will learn SQLite through extensive hands-on practices.

Getting Started

- [What Is SQLite](#)
- [Download & Install SQLite](#)
- [SQLite Sample Database](#)
- [SQLite Commands](#)

SQLite Tutorial

- [SQLite Select](#)
- [SQLite Order By](#)
- [SQLite Select Distinct](#)
- [SQLite Where](#)
- [SQLite Limit](#)
- [SQLite BETWEEN](#)
- [SQLite IN](#)
- [SQLite Like](#)
- [SQLite IS NULL](#)
- [SQLite GLOB](#)
- [SQLite Join](#)
- [SQLite Inner Join](#)
- [SQLite Left Join](#)
- [SQLite Cross Join](#)
- [SQLite Self-Join](#)
- [SQLite Full Outer Join](#)
- [SQLite Group By](#)
- [SQLite Having](#)
- [SQLite Union](#)
- [SQLite Except](#)
- [SQLite Intersect](#)
- [SQLite Subquery](#)
- [SQLite EXISTS](#)
- [SQLite Case](#)
- [SQLite Insert](#)
- [SQLite Update](#)
- [SQLite Delete](#)
- [SQLite Replace](#)
- [SQLite Transaction](#)

SQLite Data Definition

- [SQLite Data Types](#)
- [SQLite Date & Time](#)
- [SQLite Create Table](#)
- [SQLite Primary Key](#)
- [SQLite Foreign Key](#)
- [SQLite NOT NULL Constraint](#)
- [SQLite UNIQUE Constraint](#)
- [SQLite CHECK Constraint](#)
- [SQLite AUTOINCREMENT](#)
- [SQLite Alter Table](#)
- [SQLite Rename Column](#)
- [SQLite Drop Table](#)
- [SQLite Create View](#)
- [SQLite Drop View](#)
- [SQLite Index](#)
- [SQLite Expression-based Index](#)
- [SQLite Trigger](#)
- [SQLite VACUUM](#)
- [SQLite Transaction](#)
- [SQLite Full-text Search](#)

SQLite Tools

- [SQLite Commands](#)
- [SQLite Show Tables](#)
- [SQLite Describe Table](#)
- [SQLite Dump](#)
- [SQLite Import CSV](#)
- [SQLite Export CSV](#)

SQLite Functions

- [SQLite AVG](#)
- [SQLite COUNT](#)
- [SQLite MAX](#)
- [SQLite MIN](#)
- [SQLite SUM](#)

SQLite Interfaces

- [SQLite PHP](#)
- [SQLite Node.js](#)
- [SQLite Java](#)
- [SQLite Python](#)

If you have been working with other relational database management systems such as MySQL, PostgreSQL, Oracle, Microsoft SQL Server and you hear about SQLite. And you are curious to know more about it.

If your friends recommended you use an SQLite database instead of using a file to manage structured data in your applications. You want to get started with the SQLite immediately to see if you can utilize it for your apps.

If you are just starting out learning SQL and want to use SQLite as the database for your application.

If you are one of the people described above, this SQLite tutorial is for you.

SQLite is an open-source, zero-configuration, self-contained, stand-alone, transaction relational database engine designed to be embedded into an application.

Getting started with SQLite

You should go through this section if this is the first time you have worked with SQLite. Follow these 3-easy steps to get started with SQLite fast.

- First, help you answer the first important question: [what is SQLite?](#) You will have a brief overview of SQLite before working on it.
- Second, show you step by step [how to download and install the SQLite GUI tool](#) on your computer.
- Third, introduce you to an [SQLite sample database](#) and walk you through the steps of using the sample database for practicing.

Basic SQLite tutorial

This section presents basic SQL statements that you can use with SQLite. You will first start querying data from the [sample database](#). If you are already familiar with SQL, you will notice the differences between SQL standard and SQL dialect in SQLite.

Section 1. Simple query

- [Select](#) – query data from a single table using SELECT statement.

Section 2. Sorting rows

- [Order By](#) – sort the result set in ascending or descending order.

Section 3. Filtering data

- [Select Distinct](#) – query *unique* rows from a table using the DISTINCT clause.
- [Where](#) – filter rows of a result set using various conditions.
- [Limit](#) – constrain the number of rows that you want to return. The LIMIT clause helps you get the necessary data returned by a query.
- [Between](#) – test whether a value is in a range of values.
- [In](#) – check if a value matches any value in a list of value or subquery.

- [Like](#) – query data based on pattern matching using wildcard characters: percent sign (%) and underscore (_).
- [Glob](#) – determine whether a string matches a specific UNIX-pattern.
- [IS NULL](#) – check if a value is null or not.

Section 4. Joining tables

- [SQLite join](#) – learn the overview of joins including inner join, left join, and cross join.
- [Inner Join](#) – query data from multiple tables using the inner join clause.
- [Left Join](#) – combine data from multiple tables using the left join clause.
- [Cross Join](#) – show you how to use the cross join clause to produce a cartesian product of result sets of the tables involved in the join.
- [Self Join](#) – join a table to itself to create a result set that joins rows with other rows within the same table.
- [Full Outer Join](#) – show you how to emulate the full outer join in the SQLite using left join and union clauses.

Section 5. Grouping data

- [Group By](#) – combine a set of rows into groups based on specified criteria. The GROUP BY clause helps you summarize data for reporting purposes.
- [Having](#) – specify the conditions to filter the groups summarized by the GROUP BY clause.

Section 6. Set operators

- [Union](#) – combine result sets of multiple queries into a single result set. We also discuss the differences between UNION and UNION ALL clauses.
- [Except](#) – compare the result sets of two queries and returns distinct rows from the left query that are not output by the right query.
- [Intersect](#) – compare the result sets of two queries and returns distinct rows that are output by both queries.

Section 7. Subquery

- [Subquery](#) – introduce you to the SQLite subquery and correlated subquery.
- [Exists](#) operator – test for the existence of rows returned by a subquery.

Section 8. More querying techniques

- [Case](#) – add conditional logic to the query.

Section 9. Changing data

This section guides you on how to update data in the table using insert, update, and delete statements.

- [Insert](#) – insert rows into a table
- [Update](#) – update existing rows in a table.
- [Delete](#) – delete rows from a table.
- [Replace](#) – insert a new row or replace the existing row in a table.

Section 10. Transactions

- [Transaction](#) – show you how to handle transactions.

Section 11. Data definition

In this section, you'll learn how to create database objects such as tables, views, indexes using SQL data definition language.

- [SQLite Data Types](#) – introduce you to the SQLite dynamic types system and its important concepts: storage classes, manifest typing, and type affinity.
- [Create Table](#) – show you how to create a new table in the database.
- [Alter Table](#) – show you how to use modify the structure of an existing table.
- [Rename column](#) – learn step by step how to rename a column of a table.
- [Drop Table](#) – guide you on how to remove a table from the database.
- [VACUUM](#) – show you how to optimize database files.

Section 12. Constraints

- [Primary Key](#) – show you how to define the primary key for a table.
- [NOT NULL constraint](#) – ensure values in a column are not NULL.
- [UNIQUE constraint](#) – ensure values in a column or a group of columns are unique.
- [CHECK constraint](#) – ensure the values in a column meet a specified condition defined by an expression.
- [AUTOINCREMENT](#) – explain how the AUTOINCREMENT column attribute works and why you should avoid using it.

Section 13. Views

- [Create View](#) – introduce you to the view concept and show you how to create a new view in the database.
- [Drop View](#) – show you how to drop a view from its database schema.

Section 14. Indexes

- [Index](#) – teach you about the index and how to utilize indexes to speed up your queries.
- [Index for Expressions](#) – show you how to use the expression-based index.

Section 15. Triggers

- [Trigger](#) – manage triggers in the SQLite database.
- [Create INSTEAD OF triggers](#) – learn about INSTEAD OF triggers and how to create an INSTEAD OF trigger to update data via a view.

Section 16. Full-text search

- [Full-text search](#) – get started with the full-text search in SQLite.

Section 17. SQLite tools

- [SQLite Commands](#) – show you the most commonly used command in the sqlite3 program.
- [SQLite Show Tables](#) – list all tables in a database.

- [SQLite Describe Table](#) – show the structure of a table.
- [SQLite Dump](#) – how to use dump command to backup and restore a database.
- [SQLite Import CSV](#) – how to import CSV file into a table.
- [SQLite Export CSV](#) – how to export an SQLite database to CSV files.

SQLite Resources

If you want to know more information about SQLite, you can go through [a well-organized SQLite resources](#) page that contains links to useful SQLite sites.

SQLite Getting Started

If you haven't worked with the SQLite before, you following these tutorials to get started with SQLite quickly.

[What is SQLite](#)

Provide you with a brief overview of the SQLite and its important features such as serverless, self-contained, zero-configuration, and transactional.

[Download & Install SQLite Tools](#)

Show you step by step how to download and install SQLite tools.

[Introduction to the SQLite Sample Database](#)

Introduce you to an SQLite sample database and provide you with the link to download it for practicing.

[Learn the basic SQLite commands](#)

Introduce you to the most commonly used SQLite3 commands for the sqlite3 program.

What Is MySQL

Summary: this tutorial helps you answer the question: what is MySQL? And give you the reasons why MySQL is the world's most popular open-source database.

To understand MySQL, you first need to understand the database and SQL. If you already know database and SQL, you can jump to the *What is MySQL* section.

Introduction to database

You deal with data every day...

When you want to listen to your favorite songs, you open your playlist from your smartphone. In this case, the playlist is a database.

When you take a photo and upload it to your account on a social network like Facebook, your photo gallery is a database.

When you browse an e-commerce website to buy shoes, clothes, etc., you use the shopping cart database.

Databases are everywhere. So what is a database? By definition, a database is merely a structured collection of data.

The data relating to each other by nature, e.g., a product belonged to a product category and associated with multiple tags. Therefore, we use the term **relational database**.

In the relational database, we model data like products, categories, tags, etc., using tables. A table contains columns and rows. It is like a spreadsheet.

A table may relate to another table using a relationship, e.g., one-to-one and one-to-many relationships.

Because we deal with a significant amount of data, we need a way to define the databases, tables, etc., and process data more efficiently. Besides, we want to turn the data into information.

And this is where SQL comes to play.

SQL – the language of the relational database

SQL stands for the structured query language.

SQL is the standardized language used to access the database.

ANSI/SQL defines the SQL standard. The current version of SQL is SQL:2016. Whenever we refer to the SQL standard, we mean the current SQL version.

SQL contains three parts:

1. Data definition language includes statements that help you define the database and its objects, e.g., tables, [views](#), [triggers](#), [stored procedures](#), etc.
2. Data manipulation language contains statements that allow you to [update](#) and [query data](#).
3. Data control language allows you to [grant the permissions](#) to a user to access specific data in the database.

Now, you understand database and SQL, and it's time to answer the next question...

What is MySQL

MySQL? What?

My is the daughter's name of the [MySQL's co-founder, Monty Widenius](#).

The name of MySQL is the combination of My and SQL, MySQL.

MySQL is a database management system that allows you to manage relational databases. It is open source software backed by Oracle. It means you can use MySQL without paying a dime. Also, if you want, you can change its source code to suit your needs.

Even though MySQL is open source software, you can buy a commercial license version from Oracle to get premium support services.

MySQL is pretty easy to master in comparison with other database software like Oracle Database, or Microsoft SQL Server.

MySQL can run on various platforms UNIX, Linux, Windows, etc. You can install it on a server or even in a desktop. Besides, MySQL is reliable, scalable, and fast.

The official way to pronounce MySQL is *My Ess Que Ell*, not *My Sequel*. However, you can pronounce it whatever you like, who cares?

If you develop websites or web applications, MySQL is a good choice. MySQL is an essential component of the LAMP stack, which includes Linux, Apache, MySQL, and PHP.

How To Download & Install SQLite Tools

Summary: in this tutorial, you will learn step by step on how to download and use the SQLite tools to your computer.

Download SQLite tools

To download SQLite, you open the [download page](#) of the SQLite official website.

1. First, go to the <https://www.sqlite.org> website.
2. Second, open the download page <https://www.sqlite.org/download.html>

SQLite provides various tools for working across platforms e.g., Windows, Linux, and Mac. You need to select an appropriate version to download.

For example, to work with SQLite on Windows, you download the command-line shell program as shown in the screenshot below.

Precompiled Binaries for Windows

sqlite-dll-win32-x86-3290000.zip (474.63 KiB)	32-bit DLL (x86) for SQLite version 3.29.0. (sha1: 00435a36f5e6059287cde2cebb2882669cdba3a5)
sqlite-dll-win64-x64-3290000.zip (788.61 KiB)	64-bit DLL (x64) for SQLite version 3.29.0. (sha1: c88204328d6ee3ff49ca0d58cbbee05243172c3a)

sqlite-tools-win32-x86-3290000.zip (1.71 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha1: f009ff42b8c22886675005e3e57c94d62bca12b3)
------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The downloaded file is in the ZIP format and its size is quite small.

Run SQLite tools

Installing SQLite is simple and straightforward.

1. First, create a new folder e.g., `C:\sqlite`.
2. Second, extract the content of the file that you downloaded in the previous section to the `C:\sqlite` folder. You should see three programs in the `C:\sqlite` folder as shown below:

sqldiff.exe
sqlite3.exe
sqlite3_analyzer.exe

First, open the command line window:



and navigate to the C:\sqlite folder.

```
C:\>cd c:\sqlite
C:\sqlite>
```

Second, type `sqlite3` and press enter, you should see the following output:

```
C:\sqlite>sqlite3
SQLite version 3.29.0 2019-07-10 17:32:03
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Third, you can type the `.help` command from the `sqlite>` prompt to see all available commands in `sqlite3`.

```
sqlite> .help
.archive ...           Manage SQL archives: ".archive --help" for details
.auth ON|OFF           Show authorizer callbacks
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.binary on|off         Turn binary output on or off.  Default OFF
.cd DIRECTORY          Change the working directory to DIRECTORY
...
```

Fourth, to quit the `sqlite>`, you use `.quit` command as follows:

```
sqlite> .quit
```

c:\sqlite>

Install SQLite GUI tool

The sqlite3 shell is excellent...

However, sometimes, you may want to work with the SQLite databases using an intuitive GUI tool.

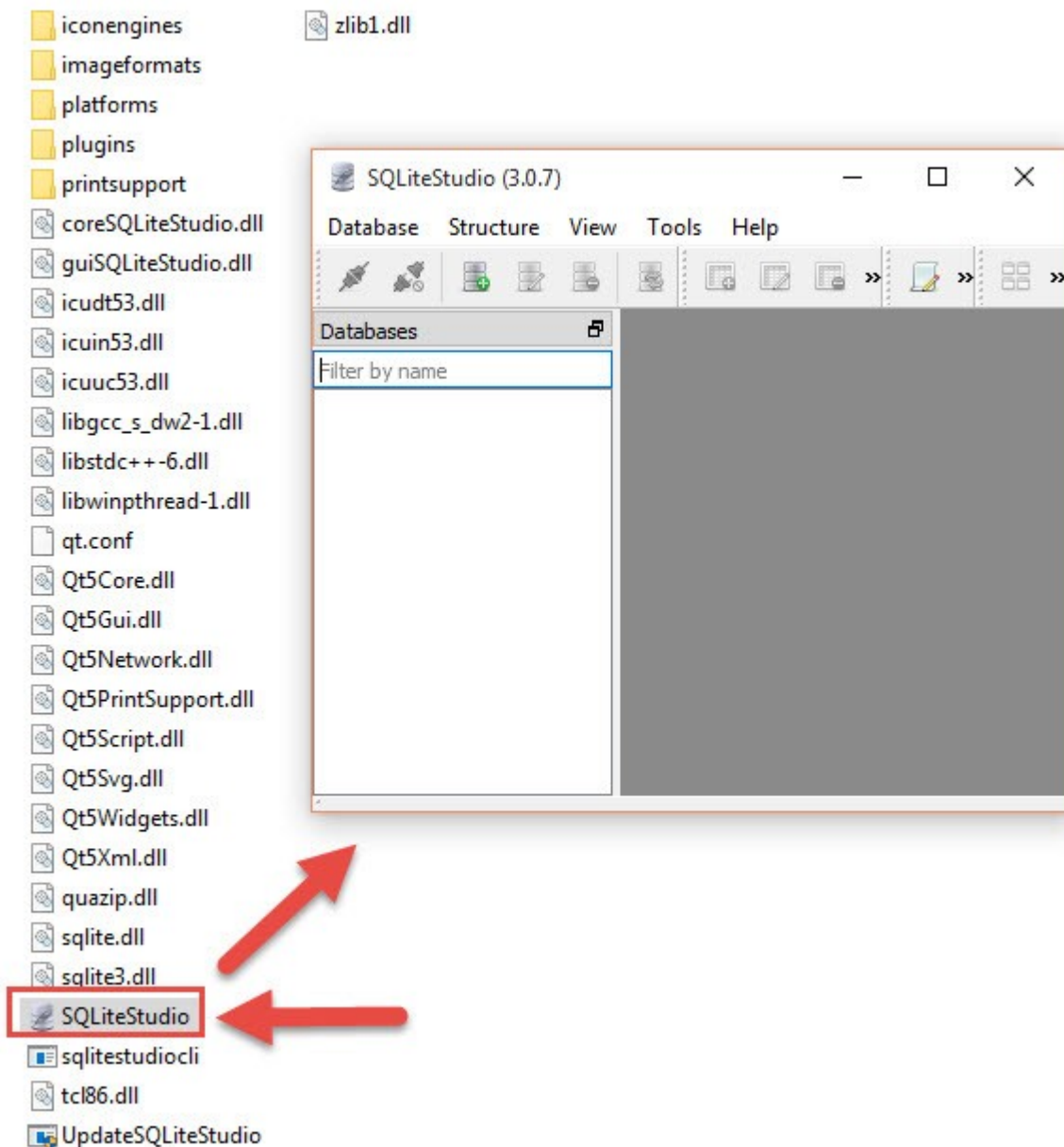
There are many GUI tools for managing SQLite databases available ranging from freeware to commercial licenses.

SQLiteStudio

The SQLiteStudio tool is a free GUI tool for managing SQLite databases. It is free, portable, intuitive, and cross-platform. SQLite tool also provides some of the most important features to work with SQLite databases such as importing, exporting data in various formats including CSV, XML, and JSON.

You can download the SQLiteStudio portable version or installer it by going to the [download page](#). Then, you can extract (or install) the download file to a folder e.g., C:\sqlite\gui\ and launch it.

The following picture illustrates how to launch the SQLiteStudio:



Other SQLite GUI tools

Besides the SQLite Studio, you can use the following free SQLite GUI tools:

- [DBeaver](#) is another free multi-platform database tool. It supports all popular major relational database systems MySQL, PostgreSQL, Oracle, DB2, SQL Server, Sybase.. including SQLite.
- [DB Browser for SQLite](#) – is an open-source tool to manage database files compatible with SQLite.

In this tutorial, you have learned how to download and install SQLite tools on your computer. Now, you should be ready to work with SQLite. If you have any issues with these above steps, feel free to [send us an email](#) to get help.

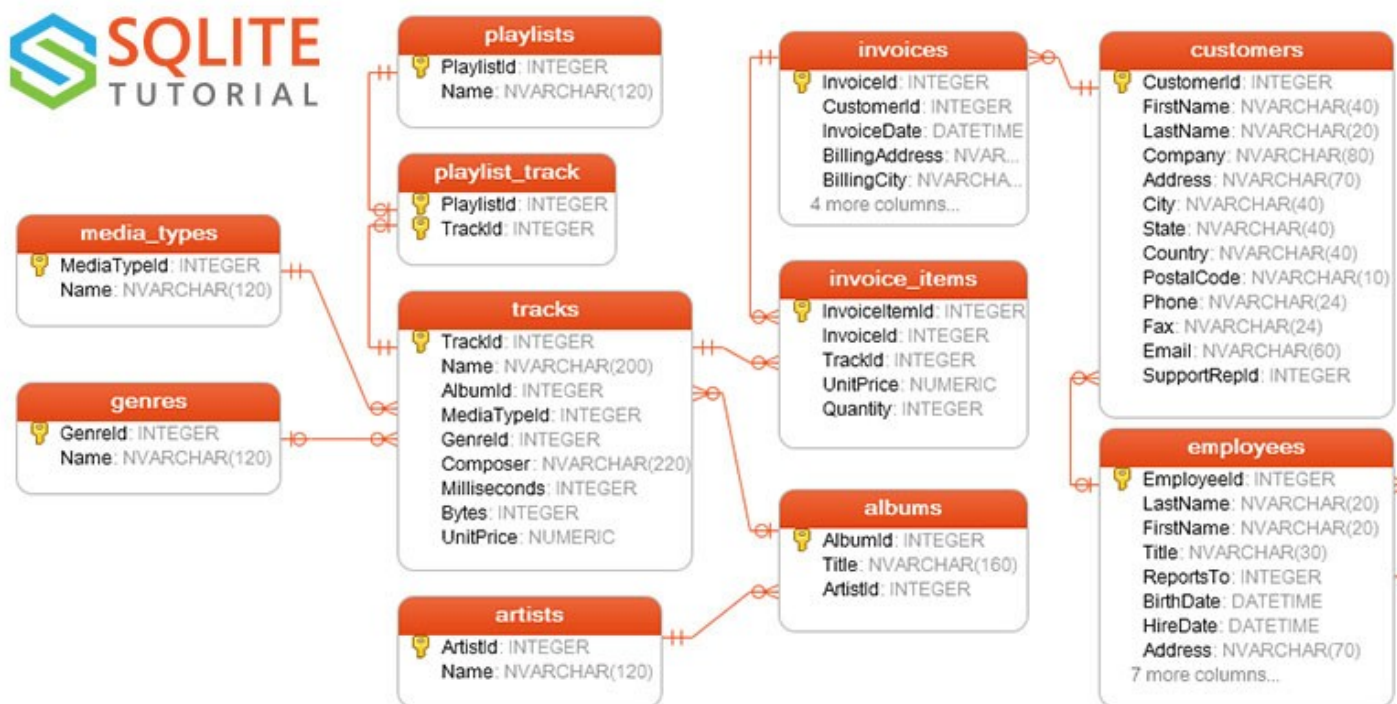
SQLite Sample Database

Summary: in this tutorial, we first introduce you to an SQLite sample database. Then, we will give you the links to download the sample database and its diagram. At the end of the tutorial, we will show you how to connect to the sample database using the sqlite3 tool.

Introduction to chinook SQLite sample database

We provide you with the SQLite sample database named chinook. The chinook sample database is a good database for practicing with SQL, especially SQLite.

The following database diagram illustrates the chinook database tables and their relationships.



Chinook sample database tables

There are 11 tables in the chinook sample database.

- **employees** table stores employees data such as employee id, last name, first name, etc. It also has a field named **ReportsTo** to specify who reports to whom.
- **customers** table stores customers data.
- **invoices** & **invoice_items** tables: these two tables store invoice data. The **invoices** table stores invoice header data and the **invoice_items** table stores the invoice line items data.
- **artists** table stores artists data. It is a simple table that contains only the artist id and name.
- **albums** table stores data about a list of tracks. Each album belongs to one artist. However, one artist may have multiple albums.
- **media_types** table stores media types such as MPEG audio and AAC audio files.

- `genres` table stores music types such as rock, jazz, metal, etc.
- `tracks` table stores the data of songs. Each track belongs to one album.
- `playlists` & `playlist_track` tables: `playlists` table store data about playlists. Each playlist contains a list of tracks. Each track may belong to multiple playlists. The relationship between the `playlists` table and `tracks` table is many-to-many. The `playlist_track` table is used to reflect this relationship.

Download SQLite sample database

You can download the SQLite sample database using the following link.

[Download SQLite sample database](#)

In case you want to have the database diagram for reference, you can download both black&white and color versions in PDF format.

[Download SQLite sample database diagram](#)

[Download SQLite sample database diagram with color](#)

How to connect to SQLite sample database

The sample database file is ZIP format, therefore, you need to extract it to a folder, for example, `C:\sqlite\db`. The name of the file is `chinook.db`

If you don't have zip software installed, you can download a [free zip software such as 7-zip](#).

First, use the command line program and navigate to the SQLite directory where the `sqlite3.exe` file is located:

```
c:\sqlite>
```

Second, use the following command to connect to the `chinook` sample database located in the `db` folder, which is a subfolder of the `sqlite` folder.

```
c:\sqlite>sqlite3 c:\sqlite\db\chinook.db
```

You should see the following command:

```
sqlite>
```

Third, try a simple [command](#) e.g., `.tables` to view all the tables available in the sample database.

```
sqlite> .tables
albums      employees  invoices    playlists
artists     genres     media_types tracks
customers   invoice_items playlist_track
```

In this tutorial, we have introduced you to the chinook SQLite sample database and showed you how to connect to it using the sqlite3 tool.

SQLite Commands

Summary: in this tutorial, we will introduce you to the most commonly used SQLite commands of the sqlite3 command-line program.

The SQLite project delivers a simple command-line tool named sqlite3 (or sqlite3.exe on Windows) that allows you to interact with the SQLite databases using SQL statements and commands.

Connect to an SQLite database

To start the sqlite3, you type the sqlite3 as follows:

```
>sqlite3
SQLite version 3.29.0 2019-07-10 17:32:03
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

By default, an SQLite session uses the in-memory database, therefore, all changes will be gone when the session ends.

To open a database file, you use the `.open FILENAME` command. The following statement opens the `chinook.db` database:

```
sqlite> .open c:\sqlite\db\chinook.db
```

If you want to open a specific database file when you connect to the SQLite database, you use the following command:

```
>sqlite3 c:\sqlite\db\chinook.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite>
```

If you start a session with a database name that does not exist, the sqlite3 tool will create the database file.

For example, the following command creates a database named `sales` in the `C:\sqlite\db\` directory:

```
>sqlite3 c:\sqlite\db\sales.db
SQLite version 3.29.0 2019-07-10 17:32:03
Enter ".help" for usage hints.
sqlite>
```

Show all available commands and their purposes

To show all available commands and their purpose, you use the `.help` command as follows:

```
.help
```

Show databases in the current database connection

To show all databases in the current connection, you use the `.databases` command. The `.databases` command displays at least one database with the name: `main`.

For example, the following command shows all the databases of the current connection:

```
sqlite> .database
seq  name                file
---  -
0    main                 c:\sqlite\db\sales.db
sqlite>
```

To add an additional database in the current connection, you use the statement [ATTACH DATABASE](#). The following statement adds the `chinook` database to the current connection.

```
sqlite> ATTACH DATABASE "c:\sqlite\db\chinook.db" AS chinook;
```

Now if you run the `.database` command again, the `sqlite3` returns two databases: `main` and `chinook`.

```
sqlite> .databases
seq  name                file
---  -
0    main                 c:\sqlite\db\sales.db
2    chinook              c:\sqlite\db\chinook.db
```

Exit sqlite3 tool

To exit the `sqlite3` program, you use the `.exit` command.

```
sqlite>.exit
```

Show tables in a database

To display all the tables in the current database, you use the `.tables` command. The following commands open a new database connection to the `chinook` database and display the tables in the database.

```
>sqlite3 c:\sqlite\db\chinook.db
SQLite version 3.29.0 2019-07-10 17:32:03
```

Enter ".help" for usage hints.

```
sqlite> .tables
```

albums	employees	invoices	playlists
artists	genres	media_types	tracks
customers	invoice_items	playlist_track	

```
sqlite>
```

If you want to find tables based on a specific pattern, you use the `.table` pattern command. The sqlite3 uses the [LIKE](#) operator for pattern matching.

For example, the following statement returns the table that ends with the string `es`.

```
sqlite> .table 'es'
```

employees	genres	invoices	media_types
-----------	--------	----------	-------------

```
sqlite>
```

Show the structure of a table

To display the structure of a table, you use the `.schema TABLE` command. The `TABLE` argument could be a pattern. If you omit it, the `.schema` command will show the structures of all the tables.

The following command shows the structure of the `albums` table.

```
sqlite> .schema albums
```

```
CREATE TABLE "albums"
(
  [AlbumId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  [Title] NVARCHAR(160) NOT NULL,
  [ArtistId] INTEGER NOT NULL,
  FOREIGN KEY ([ArtistId]) REFERENCES "artists" ([ArtistId])
    ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_AlbumArtistId] ON "albums" ([ArtistId]);
sqlite>
```

To show the schema and the content of the `sqlite_stat` tables, you use the `.fullschema` command.

```
sqlite>.fullschema
```

Show indexes

To show all indexes of the current database, you use the `.indexes` command as follows:

```
sqlite> .indexes
```

```
IFK_AlbumArtistId
IFK_CustomerSupportRepId
IFK_EmployeeReportsTo
IFK_InvoiceCustomerId
IFK_InvoiceLineInvoiceId
IFK_InvoiceLineTrackId
IFK_PlaylistTrackTrackId
IFK_TrackAlbumId
IFK_TrackGenreId
IFK_TrackMediaTypeId
```


To show the indexes of a specific table, you use the `.indexes TABLE` command. For example, to show indexes of the `albums` table, you use the following command:

```
sqlite> .indexes albums
IFK_AlbumArtistId
```

To show indexes of the tables whose names end with `es`, you use a pattern of the [LIKE](#) operator.

```
sqlite> .indexes %es
IFK_EmployeeReportsTo
IFK_InvoiceCustomerId
```

Save the result of a query into a file

To save the result of a query into a file, you use the `.output FILENAME` command. Once you issue the `.output` command, all the results of the subsequent queries will be saved to the file that you specified in the `FILENAME` argument. If you want to save the result of the next single query only to the file, you issue the `.once FILENAME` command.

To display the result of the query to the standard output again, you issue the `.output` command without arguments.

The following commands select the `title` from the `albums` table and write the result to the `albums.txt` file.

```
sqlite> .output albums.txt
sqlite> SELECT title FROM albums;
```

Execute SQL statements from a file

Suppose we have a file named `commands.txt` in the `c:\sqlite\` folder with the following content:

```
SELECT albumid, title
FROM albums
ORDER BY title
LIMIT 10;
```

To execute the SQL statements in the `commands.txt` file, you use the `.read FILENAME` command as follows:

```
sqlite> .mode column
```

```

sqlite> .header on
sqlite> .read c:/sqlite/commands.txt
AlbumId      Title
-----
156          ...And Justice For All
257          20th Century Masters -
296          A Copland Celebration,
94           A Matter of Life and D
95           A Real Dead One
96           A Real Live One
285          A Soprano Inspired
139          A TempestadeTempestade
203          A-Sides
160          Ace Of Spades

```

In this tutorial, you have learned many useful commands in the sqlite3 tool to perform various tasks that deal with the SQLite database.

SQLite Select

Summary: in this tutorial, you will learn how to use SQLite SELECT statement to query data from a single table.

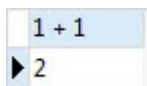
The SELECT statement is one of the most commonly used statements in SQL. The SQLite SELECT statement provides all features of the SELECT statement in SQL standard.

Simple uses of SELECT statement

You can use the SELECT statement to perform a simple calculation as follows:

```
SELECT      1 + 1;
```

[Try It](#)



You can use multiple expressions in the SELECT statement as follows:

```
SELECT
  10 / 5,
  2 * 4 ;
```

[Try It](#)

10 / 5	2 * 4
▶ 2	8

Querying data from a table using the **SELECT** statement

We often use the **SELECT** statement to query data from one or more table. The syntax of the **SELECT** statement is as follows:

```
SELECT DISTINCT column_list
FROM table_list
  JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;
```

The **SELECT** statement is the most complex statement in SQLite. To help easier to understand each part, we will break the **SELECT** statement into multiple easy-to-understand tutorials.

- Use [ORDER BY clause](#) to sort the result set
- Use [DISTINCT](#) clause to query unique rows in a table
- Use [WHERE](#) clause to filter rows in the result set
- Use [LIMIT](#) [OFFSET](#) clauses to constrain the number of rows returned
- Use [INNER JOIN](#) or [LEFT JOIN](#) to query data from multiple tables using join.
- Use [GROUP BY](#) to get the group rows into groups and apply aggregate function for each group.
- Use [HAVING](#) clause to filter groups

In this tutorial, we are going to focus on the simplest form of the **SELECT** statement that allows you to query data from a single table.

```
SELECT column_list
FROM table;
```

Even though the **SELECT** clause appears before the **FROM** clause, SQLite evaluates the **FROM** clause first and then the **SELECT** clause, therefore:

- First, specify the table where you want to get data from in the **FROM** clause. Notice that you can have more than one table in the **FROM** clause. We will discuss it in the subsequent tutorial.
- Second, specify a column or a list of comma-separated columns in the **SELECT** clause.

You use the semicolon (;) to terminate the statement.

SQLite **SELECT** examples

Let's take a look at the **tracks** table in the [sample database](#).

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

The **tracks** table contains columns and rows. It looks like a spreadsheet.

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
1	For Those /	1	1	1	Angus Young, I	343719	11170334	0.99
2	Balls to the 2	2	2	1	(Null)	342562	5510424	0.99
3	Fast As a St	3	2	1	F. Baltes, S. Kau	230619	3990994	0.99
4	Restless an	3	2	1	F. Baltes, R.A. Si	252051	4331779	0.99
5	Princess of	3	2	1	Deaffy & R.A. S	375418	6290521	0.99
6	Put The Fir	1	1	1	Angus Young, I	205662	6713451	0.99
7	Let's Get It	1	1	1	Angus Young, I	233926	7636561	0.99
8	Inject The \	1	1	1	Angus Young, I	210834	6852860	0.99
9	Snowballe	1	1	1	Angus Young, I	203102	6599424	0.99
10	Evil Walks	1	1	1	Angus Young, I	263497	8611245	0.99
11	C.O.D.	1	1	1	Angus Young, I	199836	6566314	0.99
12	Breaking TI	1	1	1	Angus Young, I	263288	8596840	0.99
13	Night Of TI	1	1	1	Angus Young, I	205688	6706347	0.99
14	Snellhoun	1	1	1	Angus Young, I	270863	8817038	0.99

To get data from the **tracks** table such as trackid, track name, composer, and unit price, you use the following statement:

```
SELECT
    trackid,
    name,
```

```

        composer,
        unitprice
FROM
    tracks;

```

[Try It](#)

You specify a list column names, which you want to get data, in the **SELECT** clause and the **tracks** table in the **FROM** clause. SQLite returns the following result:

TrackId	Name	Composer	UnitPrice
1	For Those About To Rock (We	Angus Young, Malcolm Young, Brian Johnson	0.99
2	Balls to the Wall	(Null)	0.99
3	Fast As a Shark	F. Baltes, S. Kaufman, U. Dirksneider & W. Hoffman	0.99
4	Restless and Wild	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. Dirksneider & W. Hoffman	0.99
5	Princess of the Dawn	Deaffy & R.A. Smith-Diesel	0.99
6	Put The Finger On You	Angus Young, Malcolm Young, Brian Johnson	0.99
7	Let's Get It Up	Angus Young, Malcolm Young, Brian Johnson	0.99
8	Inject The Venom	Angus Young, Malcolm Young, Brian Johnson	0.99
9	Snowballed	Angus Young, Malcolm Young, Brian Johnson	0.99
10	Evil Walks	Angus Young, Malcolm Young, Brian Johnson	0.99
11	C.O.D.	Angus Young, Malcolm Young, Brian Johnson	0.99
12	Breaking The Rules	Angus Young, Malcolm Young, Brian Johnson	0.99
13	Night Of The Long Knives	Angus Young, Malcolm Young, Brian Johnson	0.99
14	Spellbound	Angus Young, Malcolm Young, Brian Johnson	0.99

To get data from all columns, you specify the columns of the **tracks** table in the **SELECT** clause as follows:

```

SELECT
    trackid,
    name,
    albumid,
    mediatypeid,
    genreid,
    composer,
    milliseconds,
    bytes,
    unitprice
FROM
    tracks;

```

[Try It](#)

For a table with many columns, the query would be so long that time-consuming to type. To avoid this, you can use the asterisk (*), which is the shorthand for all columns of the table as follows:

```
SELECT * FROM tracks;
```

[Try It](#)

The query is shorter and cleaner now.

However...

You should use the asterisk (*) for the testing purpose only, not in the real application development.

Because...

When you develop an application, you should control what SQLite returns to your application. Suppose, a table has 3 columns, and you use the asterisk (*) to retrieve the data from all three columns.

What if someone removes a column, your application would not be working properly, because it assumes that there are three columns returned and the logic to process those three columns would be broken.

If someone adds more columns, your application may work but it gets more data than needed, which creates more I/O overhead between the database and application.

So try to avoid using the asterisk (*) as a good habit when you use the **SELECT** statement.

In this tutorial, you have learned how to use a simple form of the SQLite **SELECT** statement to query data from a single table.

SQLite LIKE

Summary: in this tutorial, you will learn how to query data based on pattern matching using SQLite **LIKE** operator.

Introduction to SQLite **LIKE** operator

Sometimes, you don't know exactly the complete keyword that you want to query. For example, you may know that your most favorite song contains the word, `elevator` but you don't know exactly the name.

To [query data](#) based on partial information, you use the **LIKE** operator in the [WHERE](#) clause of the [SELECT](#) statement as follows:

```
SELECT      column_list
FROM        table_name
WHERE       column_1 LIKE pattern;
```

Note that you can also use the `LIKE` operator in the `WHERE` clause of other statements such as the `DELETE` and `UPDATE`.

SQLite provides two wildcards for constructing patterns. They are percent sign `%` and underscore `_`:

1. The percent sign `%` wildcard matches any sequence of zero or more characters.
2. The underscore `_` wildcard matches any single character.

The percent sign `%` wildcard examples

The `s%` pattern that uses the percent sign wildcard (`%`) matches any string that starts with `s` e.g., `son` and `so`.

The `%er` pattern matches any string that ends with `er` like `peter`, `clever`, etc.

And the `%per%` pattern matches any string that contains `per` such as `percent` and `peeper`.

The underscore `_` wildcard examples

The `h_nt` pattern matches `hunt`, `hint`, etc. The `__pple` pattern matches `topple`, `supple`, `tipple`, etc.

Note that SQLite `LIKE` operator is case-insensitive. It means `"A" LIKE "a"` is true.

However, for Unicode characters that are not in the ASCII ranges, the `LIKE` operator is case sensitive e.g., `"Ä" LIKE "ä"` is false.

In case you want to make `LIKE` operator works case-sensitively, you need to use the following [PRAGMA](#):

```
PRAGMA case_sensitive_like = true;
```

SQLite `LIKE` examples

We'll use the table `tracks` in the [sample database](#) for the demonstration.

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

To find the tracks whose names start with the `Wild` literal string, you use the percent sign `%` wildcard at the end of the pattern.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name LIKE 'Wild%'
```

[Try It](#)

TrackId	Name
1245	Wildest Dreams
1973	Wild Side
2627	Wild Hearted Son
2633	Wild Flower
2944	Wild Honey

To find the tracks whose names end with `Wild` word, you use `%` wildcard at the beginning of the pattern.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name LIKE '%Wild'
```

[Try It](#)

TrackId	Name
4	Restless and Wild
32	Deuces Are Wild
775	Call Of The Wild
2697	I Go Wild

To find the tracks whose names contain the `Wild` literal string, you use `%` wildcard at the beginning and end of the pattern:

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name LIKE '%wild%';
```

[Try It](#)

TrackId	Name
4	Restless and Wild
32	Deuces Are Wild
775	Call Of The Wild
1245	Wildest Dreams
1869	Where The Wild Things Are
1973	Wild Side
2312	Near Wild Heaven
2627	Wild Hearted Son
2633	Wild Flower
2697	I Go Wild
2930	Who's Gonna Ride Your Wild Horses
2944	Wild Honey

The following statement finds the tracks whose names contain: zero or more characters (`%`), followed by `Br`, followed by a character (`_`), followed by `wn`, and followed by zero or more characters (`%`).

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name LIKE '%Br_wn%';
```

[Try It](#)

SQLite LIKE with ESCAPE clause

If the pattern that you want to match contains % or _, you must use an escape character in an optional ESCAPE clause as follows:

```
column_1 LIKE pattern ESCAPE expression;
```

When you specify the ESCAPE clause, the LIKE operator will evaluate the expression that follows the ESCAPE keyword to a string which consists of a single character, or an escape character.

Then you can use this escape character in the pattern to include literal percent sign (%) or underscore (_). The LIKE operator evaluates the percent sign (%) or underscore (_) that follows the escape character as a literal string, not a wildcard character.

Suppose you want to match the string 10% in a column of a table. However, SQLite interprets the percent symbol % as the wildcard character. Therefore, you need to escape this percent symbol % using an escape character:

```
column_1 LIKE '%10\%%' ESCAPE '\';
```

In this expression, the LIKE operator interprets the first % and last % percent signs as wildcards and the second percent sign as a literal percent symbol.

Note that you can use other characters as the escape character e.g., /, @, \$.

Consider the following example:

First, [create a table](#) t that has one column:

```
CREATE TABLE t(  
  c TEXT  
);
```

Next, [insert](#) some rows into the table t:

```
INSERT INTO t(c)
VALUES('10% increase'),
      ('10 times decrease'),
      ('100% vs. last year'),
      ('20% increase next year');
```

Then, query data from the t table:

```
SELECT * FROM t;

c
-----
10% increase
10 times decrease
100% vs. last year
20% increase next year
```

Fourth, attempt to find the row whose value in the C column contains the 10% literal string:

```
SELECT c
FROM t
WHERE c LIKE '%10%';
```

However, it returns rows whose values in the c column contains 10:

```
c
-----
10% increase
10 times decrease
100% vs. last year
```

Fifth, to get the correct result, you use the **ESCAPE** clause as shown in the following query:

```
SELECT c
FROM t
WHERE c LIKE '%10\%%' ESCAPE '\';
```

Here is the result set:

```
c
-----
10% increase
```

In this tutorial, you have learned how to use SQLite **LIKE** operator to query data based on pattern matching using two wildcard characters percent sign (%) and underscore (_).

SQLite Transaction

Summary: in this tutorial, we will show you how to use the SQLite transaction to ensure the integrity and reliability of the data.

SQLite & ACID

SQLite is a transactional database that all changes and queries are atomic, consistent, isolated, and durable (ACID).

SQLite guarantees all the transactions are ACID compliant even if the transaction is interrupted by a program crash, operation system dump, or power failure to the computer.

- **Atomic:** a transaction should be atomic. It means that a change cannot be broken down into smaller ones. When you commit a transaction, either the entire transaction is applied or not.
- **Consistent:** a transaction must ensure to change the database from one valid state to another. When a transaction starts and executes a statement to modify data, the database becomes inconsistent. However, when the transaction is committed or rolled back, it is important that the transaction must keep the database consistent.
- **Isolation:** a pending transaction performed by a session must be isolated from other sessions. When a session starts a transaction and executes the [INSERT](#) or [UPDATE](#) statement to change the data, these changes are only visible to the current session, not others. On the other hand, the changes committed by other sessions after the transaction started should not be visible to the current session.
- **Durable:** if a transaction is successfully committed, the changes must be permanent in the database regardless of the condition such as power failure or program crash. On the contrary, if the program crashes before the transaction is committed, the change should not persist.

SQLite transaction statements

By default, SQLite operates in auto-commit mode. It means that for each command, SQLite starts, processes, and commits the transaction automatically.

To start a transaction explicitly, you use the following steps:

First, open a transaction by issuing the **BEGIN TRANSACTION** command.

```
BEGIN TRANSACTION;
```

After executing the statement `BEGIN TRANSACTION`, the transaction is open until it is explicitly committed or rolled back.

Second, issue SQL statements to select or update data in the database. Note that the change is only visible to the current session (or client).

Third, commit the changes to the database by using the `COMMIT` or `COMMIT TRANSACTION` statement.

```
COMMIT;
```

If you do not want to save the changes, you can roll back using the `ROLLBACK` or `ROLLBACK TRANSACTION` statement:

```
ROLLBACK;
```

SQLite transaction example

We will create two new tables: `accounts` and `account_changes` for the demonstration.

The `accounts` table stores data about the account numbers and their balances. The `account_changes` table stores the changes of the accounts.

First, create the `accounts` and `account_changes` tables by using the following [CREATE TABLE](#) statements:

```
CREATE TABLE accounts (  
    account_no INTEGER NOT NULL,
```

```

        balance DECIMAL NOT NULL DEFAULT 0,
        PRIMARY KEY(account_no),
        CHECK(balance >= 0)
    );

CREATE TABLE account_changes (
    change_no INT NOT NULL PRIMARY KEY,
    account_no INTEGER NOT NULL,
    flag TEXT NOT NULL,
    amount DECIMAL NOT NULL,
    changed_at TEXT NOT NULL
);

```

Second, [insert](#) some sample data into the `accounts` table.

```



INSERT INTO accounts (account_no, balance)
VALUES (100, 20100);

INSERT INTO accounts (account_no, balance)
VALUES (200, 10100);

```

Third, query data from the `accounts` table:

```
SELECT * FROM accounts;
```

account_no 	balance 
100	20,100
200	10,100

Fourth, transfer 1000 from account 100 to 200, and log the changes to the table `account_changes` in a single transaction.

```

BEGIN TRANSACTION;

UPDATE accounts
    SET balance = balance - 1000
    WHERE account_no = 100;

UPDATE accounts
    SET balance = balance + 1000

```

```

WHERE account_no = 200;

INSERT INTO account_changes(account_no, flag, amount, changed_at)
VALUES(100, '-', 1000, datetime('now'));

INSERT INTO account_changes(account_no, flag, amount, changed_at)
VALUES(200, '+', 1000, datetime('now'));

COMMIT;

```

Fifth, query data from the `accounts` table:

```
SELECT * FROM accounts;
```

account_no	balance
100	19,100
200	11,100

As you can see, balances have been updated successfully.

Sixth, query the contents of the `account_changes` table:

```
SELECT * FROM account_changes;
```

change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:33:01
2	200	+	1,000	2019-08-19 10:33:04

Let's take another example of rolling back a transaction.

First, attempt to deduct 20,000 from account 100:

```

BEGIN TRANSACTION;

UPDATE accounts
  SET balance = balance - 20000
  WHERE account_no = 100;

INSERT INTO account_changes(account_no, flag, amount, changed_at)
VALUES(100, '-', 20000, datetime('now'));

```

SQLite issued an error due to not enough balance:

[SQLITE_CONSTRAINT] Abort due to constraint violation (CHECK constraint failed: accounts)

However, the log has been saved to the `account_changes` table:

```
SELECT * FROM account_changes;
```

change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:48:38
2	200	+	1,000	2019-08-19 10:48:40
3	100	-	20,000	2019-08-19 10:54:07

Second, roll back the transaction by using the `ROLLBACK` statement:

```
ROLLBACK;
```

Finally, query data from the `account_changes` table, you will see that the change no #3 is not there anymore:

```
SELECT * FROM account_changes;
```

change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:48:38
2	200	+	1,000	2019-08-19 10:48:40

In this tutorial, you have learned how to deal with SQLite transactions by using the `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` statements to control the transactions in the SQLite database.

<https://www.sqlitetutorial.net/>

SQLite Foreign Key

Summary: in this tutorial, you will learn how to use the SQLite foreign key constraint to enforce the relationships between related tables.

SQLite foreign key constraint support

SQLite has supported foreign key constraint since version 3.6.19. The SQLite library must also be compiled with neither [SQLITE_OMIT_FOREIGN_KEY](#) nor [SQLITE_OMIT_TRIGGER](#).

To check whether your current version of SQLite supports foreign key constraints or not, you use the following command.

```
PRAGMA foreign_keys;
```

The command returns an integer value: 1: enable, 0: disabled. If the command returns nothing, it means that your SQLite version doesn't support foreign key constraints.

If the SQLite library is compiled with foreign key constraint support, the application can use the `PRAGMA foreign_keys` command to enable or disable foreign key constraints at runtime.

To disable foreign key constraint:

```
PRAGMA foreign_keys = OFF;
```

To enable foreign key constraint:

```
PRAGMA foreign_keys = ON;
```

Introduction to the SQLite foreign key constraints

Let's start with two tables: `suppliers` and `supplier_groups` :

```
CREATE TABLE suppliers (
    supplier_id integer PRIMARY KEY,
    supplier_name text NOT NULL,
    group_id integer NOT NULL
);
```

```
CREATE TABLE supplier_groups (
    group_id integer PRIMARY KEY,
    group_name text NOT NULL
);
```

Assuming that each supplier belongs to one and only one supplier group. And each supplier group may have zero or many suppliers. The relationship between `supplier_groups` and `suppliers` tables is one-to-many. In other words, for each row in the `suppliers` table, there is a corresponding row in the `supplier_groups` table.

Currently, there is no way to prevent you from adding a row to the `suppliers` table without a corresponding row in the `supplier_groups` table.

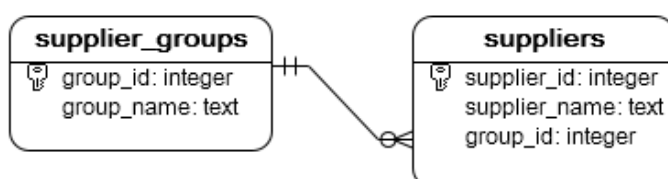
In addition, you may remove a row in the `supplier_groups` table without deleting or updating the corresponding rows in the `suppliers` table. This may leave orphaned rows in the `suppliers` table.

To enforce the relationship between rows in the `suppliers` and `supplier_groups` table, you use the **foreign key constraints**.

To add the foreign key constraint to the `suppliers` table, you change the definition of the `CREATE TABLE` statement above as follows:

```
DROP TABLE suppliers;
```

```
CREATE TABLE suppliers (
    supplier_id INTEGER PRIMARY KEY,
    supplier_name TEXT NOT NULL,
    group_id INTEGER NOT NULL,
    FOREIGN KEY (group_id)
        REFERENCES supplier_groups (group_id)
);
```



The `supplier_groups` table is called a **parent table**, which is the table that a foreign key references. The `suppliers` table is known as a **child table**, which is the table to which the foreign key constraint applies.

The `group_id` column in the `supplier_groups` table is called the **parent key**, which is a column or a set of columns in the parent table that the foreign key constraint references. Typically, the parent key is the primary key of the parent table.

The `group_id` column in the `suppliers` table is called the child key. Generally, the child key references to the [primary key](#) of the parent table.

SQLite foreign key constraint example

First, [insert](#) three rows into the `supplier_groups` table.

```
INSERT INTO supplier_groups (group_name)
VALUES
    ('Domestic'),
    ('Global'),
    ('One-Time');
```

123 group_id	ABC group_name
1	Domestic
2	Global
3	One-Time

Second, insert a new supplier into the `suppliers` table with the supplier group that exists in the `supplier_groups` table.

```
INSERT INTO suppliers (supplier_name, group_id)
VALUES ('HP', 2);
```

This statement works perfectly fine.

Third, attempt to insert a new supplier into the `suppliers` table with the supplier group that does not exist in the `supplier_groups` table.

```
INSERT INTO suppliers (supplier_name, group_id)
VALUES('ABC Inc.', 4);
```

SQLite checked the foreign key constraint, rejected the change, and issued the following error message:

```
[SQLITE_CONSTRAINT] Abort due to constraint violation (FOREIGN KEY constraint failed)
```

SQLite foreign key constraint actions

What would happen if you [delete](#) a row in the `supplier_groups` table? Should all the corresponding rows in the `suppliers` table are also deleted? The same questions to the [update](#) operation.

To specify how foreign key constraint behaves whenever the parent key is deleted or updated, you use the `ON DELETE` or `ON UPDATE` action as follows:

```
FOREIGN KEY (foreign_key_columns)
REFERENCES parent_table(parent_key_columns)
ON UPDATE action
ON DELETE action;
```

SQLite supports the following actions:

- `SET NULL`
- `SET DEFAULT`
- `RESTRICT`
- `NO ACTION`
- `CASCADE`

In practice, the values of the primary key in the parent table do not change therefore the update rules are less important. The more important rule is the `DELETE` rule that specifies the action when the parent key is deleted.

We'll examine each action by the following example

SET NULL

When the parent key changes, delete or update, the corresponding child keys of all rows in the child table set to `NULL`.

First, [drop](#) and [create](#) the table `suppliers` using the `SET NULL` action for the `group_id` foreign key:

```
DROP TABLE suppliers;
```

```
CREATE TABLE suppliers (  
    supplier_id INTEGER PRIMARY KEY,  
    supplier_name TEXT NOT NULL,  
    group_id INTEGER,  
    FOREIGN KEY (group_id)  
    REFERENCES supplier_groups (group_id)  
    ON UPDATE SET NULL  
    ON DELETE SET NULL  
);
```

Second, [insert](#) some rows into the `suppliers` table:

```
INSERT INTO suppliers (supplier_name, group_id)  
VALUES('XYZ Corp', 3);
```

```
INSERT INTO suppliers (supplier_name, group_id)  
VALUES('ABC Corp', 3);
```

Third, delete the supplier group id 3 from the `supplier_groups` table:

```
DELETE FROM supplier_groups  
WHERE group_id = 3;
```

Fourth, [query data](#) from the `suppliers` table.

```
SELECT * FROM suppliers;
```

123 supplier_id	ABC supplier_name	123 group_id
1	XYZ Corp	[NULL]
2	ABC Corp	[NULL]

The values of the `group_id` column of the corresponding rows in the `suppliers` table set to NULL.

SET DEFAULT

The SET DEFAULT action sets the value of the foreign key to the default value specified in the column definition when you [create the table](#).

Because the values in the column `group_id` defaults to NULL, if you delete a row from the `supplier_groups` table, the values of the `group_id` will set to NULL.

After assigning the default value, the foreign key constraint kicks in and carries the check.

RESTRICT

The RESTRICT action does not allow you to change or delete values in the parent key of the parent table.

First, drop and create the `suppliers` table with the RESTRICT action in the foreign key `group_id`:

```
DROP TABLE suppliers;

CREATE TABLE suppliers (
    supplier_id INTEGER PRIMARY KEY,
    supplier_name TEXT NOT NULL,
    group_id INTEGER,
    FOREIGN KEY (group_id)
    REFERENCES supplier_groups (group_id)
    ON UPDATE RESTRICT
    ON DELETE RESTRICT
);
```

Second, insert a row into the table `suppliers` with the `group_id` 1.

```
INSERT INTO suppliers (supplier_name, group_id)
VALUES('XYZ Corp', 1);
```

Third, delete the supplier group with id 1 from the `supplier_groups` table:

```
DELETE FROM supplier_groups
WHERE group_id = 1;
```

SQLite issued the following error:

```
[SQLITE_CONSTRAINT] Abort due to constraint violation (FOREIGN KEY constraint failed)
```

To fix it, you must first delete all rows from the `suppliers` table which has `group_id` 1:

```
DELETE FROM suppliers
WHERE group_id =1;
```

Then, you can delete the supplier group 1 from the `supplier_groups` table:

```
DELETE FROM supplier_groups
WHERE group_id = 1;
```

NO ACTION

The `NO ACTION` does not mean by-pass the foreign key constraint. It has the similar effect as the `RESTRICT`.

CASCADE

The `CASCADE` action propagates the changes from the parent table to the child table when you update or delete the parent key.

First, insert the `supplier` groups into the `supplier_groups` table:

```
INSERT INTO supplier_groups (group_name)
VALUES
    ('Domestic'),
    ('Global'),
    ('One-Time');
```

123 group_id 🔼🔼	ABC group_name 🔼🔼
1	Domestic
2	Global
3	One-Time

Second, drop and create the table `suppliers` with the `CASCADE` action in the foreign key `group_id`:

```
DROP TABLE suppliers;
```

```
CREATE TABLE suppliers (  
    supplier_id INTEGER PRIMARY KEY,  
    supplier_name TEXT NOT NULL,  
    group_id INTEGER,  
    FOREIGN KEY (group_id)  
    REFERENCES supplier_groups (group_id)  
        ON UPDATE CASCADE  
        ON DELETE CASCADE  
);
```

Third, insert some suppliers into the table `suppliers`:

```
INSERT INTO suppliers (supplier_name, group_id)  
VALUES('XYZ Corp', 1);
```

```
INSERT INTO suppliers (supplier_name, group_id)  
VALUES('ABC Corp', 2);
```

123 supplier_id	ABC supplier_name	123 group_id
1	XYZ Corp	1
2	ABC Corp	2

Fourth, update `group_id` of the `Domestic` supplier group to 100:

```
UPDATE supplier_groups  
SET group_id = 100  
WHERE group_name = 'Domestic';
```

Fifth, query data from the table `suppliers`:


```
SELECT * FROM suppliers;
```

supplier_id	supplier_name	group_id
1	XYZ Corp	100
2	ABC Corp	2

As you can see the value in the `group_id` column of the XYZ Corp in the table `suppliers` changed from 1 to 100 when we updated the `group_id` in the `supplier_groups` table. This is the result of `ON UPDATE CASCADE` action.

Sixth, delete supplier group id 2 from the `supplier_groups` table:

```
DELETE FROM supplier_groups  
WHERE group_id = 2;
```

Seventh, query data from the table `suppliers` :

```
SELECT * FROM suppliers;
```

supplier_id	supplier_name	group_id
1	XYZ Corp	100

The supplier id 2 whose `group_id` is 2 was deleted when the supplier group id 2 was removed from the `supplier_groups` table. This is the effect of the `ON DELETE CASCADE` action.

In this tutorial, you have learned about SQLite foreign key constraint and how to use them to enforce the relationship between related tables.

Ferramentas

<https://www.phpliteadmin.org/download/>

<https://sqlitebrowser.org/>

`sudo apt install sqlitebrowser`

<https://github.com/coleifer/sqlite-web>

<http://sqliteadmin.orbmu2k.de/>

SQLite Describe Table

Summary: in this tutorial, you will learn about various ways to show the structure of a table in SQLite.

Getting the structure of a table via the SQLite command-line shell program

To find out the structure of a table via the SQLite command-line shell program, you follow these steps:

First, connect to a database via the SQLite command-line shell program:

```
> sqlite3 c:\sqlite\db\chinook.db
```

Then, issue the following command:

```
.schema table_name
```

For example, to show the statement that created the `albums` table, you use the following command:

```
sqlite> .schema albums
```

Notice that you should not use the semicolon (;).

Here is the output:

```
CREATE TABLE IF NOT EXISTS "albums"
```

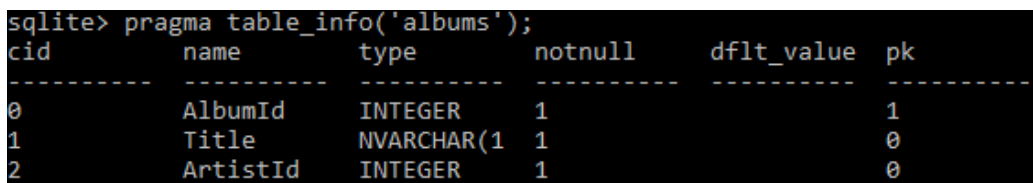
```
(
    [AlbumId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [Title] NVARCHAR(160) NOT NULL,
    [ArtistId] INTEGER NOT NULL,
    FOREIGN KEY ([ArtistId]) REFERENCES "artists" ([ArtistId])
        ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_AlbumArtistId] ON "albums" ([ArtistId]);
```

Another way to show the structure of a table is to use the following PRAGMA command:

```
sqlite> .header on
sqlite> .mode column
sqlite> pragma table_info('albums');
```

Note that the first two commands are used to format the output nicely.

The following picture shows the output:



cid	name	type	notnull	dflt_value	pk
0	AlbumId	INTEGER	1		1
1	Title	NVARCHAR(160)	1		0
2	ArtistId	INTEGER	1		0

Getting the structure of a table using the SQL statement

You can find the structure of a table by querying it from the `sqlite_master` table as follows:

```
SELECT sql
FROM sqlite_master
WHERE name = 'albums';
```

Here is the output:

```
sql
-----
CREATE TABLE "albums"
(
    [AlbumId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
```

```

[Title] NVARCHAR(160) NOT NULL,
[ArtistId] INTEGER NOT NULL,
FOREIGN KEY ([ArtistId]) REFERENCES "artists" ([ArtistId])
ON DELETE NO ACTION ON UPDATE NO ACTION
)

```

In this tutorial, you have learned how to show the structure of a table in SQLite via command-line shell program or SQL statement.

SQLite Data Types

Summary: in this tutorial, you will learn about SQLite data types system and its related concepts such as storage classes, manifest typing, and type affinity.

Introduction to SQLite data types

If you come from other database systems such as [MySQL](#) and [PostgreSQL](#), you notice that they use *static typing*. It means when you declare a column with a specific data type, that column can store only data of the declared data type.

Different from other database systems, SQLite uses *dynamic type system*. In other words, a value stored in a column determines its data type, not the column's data type.

In addition, you don't have to declare a specific data type for a column when you create a table. In case you declare a column with the integer data type, you can store any kind of data types such as text and BLOB, SQLite will not complain about this.

SQLite provides five primitive data types which are referred to as *storage classes*.

Storage classes describe the formats that SQLite uses to store data on disk. A storage class is more general than a data type e.g., INTEGER storage class includes 6 different types of integers. In most cases, you can use storage classes and data types interchangeably.

The following table illustrates 5 storage classes in SQLite:

Storage Class	Meaning
NULL	NULL values mean missing information or unknown.
INTEGER	Integer values are whole numbers (either positive or negative). An integer can have variable sizes such as 1, 2, 3, 4, or 8 bytes.
REAL	Real values are real numbers with decimal values that use 8-byte floats.
TEXT	TEXT is used to store character data. The maximum length of TEXT is unlimited. SQLite supports various character encodings.
BLOB	BLOB stands for a binary large object that can store any kind of data. The maximum size of BLOB is, theoretically, unlimited.

SQLite determines the data type of a value based on its data type according to the following rules:

- If a literal has no enclosing quotes and decimal point or exponent, SQLite assigns the INTEGER storage class.
- If a literal is enclosed by single or double quotes, SQLite assigns the TEXT storage class.

- If a literal does not have quote nor decimal point nor exponent, SQLite assigns REAL storage class.
- If a literal is NULL without quotes, it assigned NULL storage class.
- If a literal has the X'ABCD' or x 'abcd', SQLite assigned BLOB storage class.

SQLite does not support built-in date and time storage classes. However, you can use the TEXT, INT, or REAL to store date and time values. For the detailed information on how to handle date and time values, check it out the [SQLite date and time tutorial](#).

SQLites provides the `typeof()` function that allows you to check the storage class of a value based on its format. See the following example:

```
SELECT
```

```
    typeof(100),
    typeof(10.0),
    typeof('100'),
    typeof(x'1000'),
    typeof(NULL);
```

typeof(100)	typeof(10.0)	typeof('100')	typeof(x'1000')	typeof(NULL)
integer	real	text	blob	null

A single column in SQLite can store mixed data types. See the following example.

First, [create a new table](#) named `test_datatypes` for testing.

```
CREATE TABLE test_datatypes (
    id INTEGER PRIMARY KEY,
    val
);
```

Second, [insert](#) data into the `test_datatypes` table.

```
INSERT INTO test_datatypes (val)
VALUES
    (1),
    (2),
    (10.1),
    (20.5),
```

```
('A'),  
( 'B'),  
(NULL),  
(x'0010'),  
(x'0011');
```

Third, use the `typeof()` function to get the data type of each value stored in the `val` column.

```
SELECT  
    id,  
    val,  
    typeof(val)  
FROM  
    test_datatypes;
```

id	val	typeof(val)
1	1	integer
2	2	integer
3	10.1	real
4	20.5	real
5	A	text
6	B	text
7	(Null)	null
8		blob
9		blob

You may ask how SQLite [sorts](#) data in a column with different storage classes like `val` column above.

To resolve this, SQLite provides the following set of rules when it comes to sorting:

- NULL storage class has the lowest value. It is lower than any other values. Between NULL values, there is no order.
- The next higher storage classes are INTEGER and REAL. SQLite compares INTEGER and REAL numerically.
- The next higher storage class is TEXT. SQLite uses the collation of TEXT values when it compares the TEXT values.
- The highest storage class is the BLOB. SQLite uses the C function `memcmp()` to compare BLOB values.

When you use the [ORDER BY](#) clause to sort the data in a column with different storage classes, SQLite performs the following steps:

- First, group values based on storage class: NULL, INTEGER, and REAL, TEXT, and BLOB.
- Second, sort the values in each group.

The following statement sorts the mixed data in the `val` column of the `test_datatypes` table:

```

SELECT
    id,
    val,
    typeof(val)
FROM
    test_datatypes
ORDER BY val;

```

id	val	typeof(val)
7	(Null)	null
1	1	integer
2	2	integer
3	10.1	real
4	20.5	real
5	A	text
6	B	text
8		blob
9		blob

SQLite manifest typing & type affinity

Other important concepts related to SQLite data types are manifest typing and type affinity:

- Manifest typing means that a data type is a property of a value stored in a column, not the property of the column in which the value is stored. SQLite uses manifest typing to store values of any type in a column.
- Type affinity of a column is the recommended type for data stored in that column. Note that the data type is recommended, not required, therefore, a column can store any type of data.

In this tutorial, you have learned about SQLite data types and some important concepts including storage classes, manifest typing, and type affinity.