

Introduction to SQLite

SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. It is the most used database engine in the world. It is an in-process library and its code is publicly available. It is free for use for any purpose, commercial or private. It is basically an embedded SQL database engine. Ordinary disk files can be easily read and write by SQLite because it does not have any separate server like SQL. The SQLite database file format is cross-platform so that anyone can easily copy a database between 32-bit and 64-bit systems. Due to all these features, it is a popular choice as an Application File Format.

History:

It was designed by D. Richard Hipp for the purpose of no administration required for operating a program. in August 2000. As it is very lightweight compared to others like MySql and Oracle, it is called SQLite. Different versions of SQLite are released since 2000.

Installation on Windows:

1. Visit the official website of SQLite for downloading the zip file.
2. Download that zip file.
3. Create a folder in C or D (wherever you want) for storing SQLite by expanding zip file.
4. Open the command prompt and set the path for the location of SQLite folder given in the previous step. After that write "sqlite3" and press enter.

You can also directly open the .exe file from the folder where you have stored the SQLite whole thing.

After clicking on the selected .exe file it will open SQLite application

Installation on Linux:

Open Terminal, type this command and enter password

```
sudo apt-get install sqlite3  
sqlite3
```

It will automatically install and once it asks Do you want to continue (Y/N) type Y and press enter. After successful installation, we can check it by command sqlite3.
installation of sqlite3

Features and Limitation of SQLite

SQLite is defined by the following features:

Serverless: –

SQLite does not require a server process or system to operate database. The SQLite library accesses its storage files directly.

Zero Configuration: –

No server means no setup. SQLite is not required to install any application, configure, and nothing to worry about.

Cross-Platform: –

The entire database instance resides in a single cross-platform file, it is not required any administration.

Self-Contained: –

A single library contains the entire database system, which integrated directly into a host application.

Small Runtime Footprint: –

The default build is less than a megabyte of code and requires only a few megabytes of memory. With some adjustments, both the library size and memory use can be significantly reduced.

Transactional: –

SQLite transactions are follows ACID property, it is allowing safe access from multiple processes or threads.

Full-Featured: –

SQLite supports most of the query language features found in the SQL92 (SQL2) standard.

Highly Reliable: –

The purpose of a database is to keep your data safe and organized. To maintain a high level – reliability of database, the core SQLite library is tested before each release. In full, the standard SQLite test suites consist of over 10 million-unit tests and query tests.

Limitations: –

Foreign key constraints: – Foreign keys are the foundation of referential integrity in relational databases. While SQLite parses them, it currently does not have support for foreign keys. It does support check constraints, and foreign key support is estimated to be completed by sometime in 2006.

Trigger support: –There is some support for triggers but it is not complete. Missing features include FOR EACH STATEMENT triggers (currently all triggers must be FOR EACH ROW), INSTEAD OF triggers on tables (currently INSTEAD OF triggers are only allowed on views), and recursive triggers—triggers that trigger themselves.

ALTER TABLE support: – Only the RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. Other kinds of ALTER TABLE operations such as DROP COLUMN, ALTER COLUMN, and ADD CONSTRAINT are not implemented.

Nested transactions: – SQLite allows only a single transaction to be active at one time. Nested transactions allow for fine-grained control over larger, more complex operations in that parts of a transaction can be defined and rolled back in case of an error rather than the entire transaction.

RIGHT and FULL OUTER JOIN: – LEFT OUTER JOIN is implemented, but RIGHT OUTER JOIN and FULL OUTER JOIN are not implemented. LEFT OUT JOIN can be implemented as a right outer join by simplified reversing the order of the tables and

modify the join constraint. Furthermore, FULL OUTER JOIN can be implemented as a combination of other relational operations supported by SQLite.

Updatable views: – VIEWS in SQLite are read-only. You may not execute a DELETE, INSERT, or UPDATE statement on a view. But you can create a trigger that fires on an attempt to DELETE, INSERT, or UPDATE a view and do what you need in the body of the trigger.

GRANT and REVOKE: – GRANT and REVOKE commands in general are aimed at much higher end systems where there are multiple users who have varying access levels to data in the database.

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](#).

SQLite 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);
```

Code language: SQL (Structured Query Language) (sql)

<code>typeof(100)</code>	<code>typeof(10.0)</code>	<code>typeof('100')</code>	<code>typeof(x'1000')</code>	<code>typeof(NULL)</code>
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
);
```

Code language: SQL (Structured Query Language) (sql)

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');
```

Code language: SQL (Structured Query Language) (sql)

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;
```

Code language: SQL (Structured Query Language) (sql)

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;
```

Code language: SQL (Structured Query Language) (sql)

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.

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.properties:-

- **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;
```

Code language: SQL (Structured Query Language) (sql)

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;
```

Code language: SQL (Structured Query Language) (sql)

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

```
ROLLBACK;
```

Code language: SQL (Structured Query Language) (sql)

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  
VALUES(1,100,'-',1000,datetime('now'));
```

```
INSERT INTO account_changes  
VALUES(2,200,'+',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;
```

Code language: SQL (Structured Query Language) (sql)

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;
```

Code language: SQL (Structured Query Language) (sql)

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

How to create and open Database

.open database1.db

Filtering data

SQLite Select Distinct

Summary: in this tutorial, you will learn how to use the SQLite `SELECT DISTINCT` clause to remove duplicate rows in the result set.

Introduction to SQLite `SELECT DISTINCT` clause

The `DISTINCT` clause is an optional clause of the [SELECT](#) statement.

The `DISTINCT` clause allows you to remove the duplicate rows in the result set.

The following statement illustrates the syntax of the `DISTINCT` clause:

```
SELECT DISTINCT select list
FROM table;
```

In this syntax:

- First, the `DISTINCT` clause must appear immediately after the `SELECT` keyword.
- Second, you place a column or a list of columns after the `DISTINCT` keyword. If you use one column, SQLite uses values in that column to evaluate the duplicate. In case you use multiple columns, SQLite uses the combination of values in these columns to evaluate the duplicate.

SQLite considers `NULL` values as duplicates. If you use the `DISTINCT` clause with a column that has `NULL` values, SQLite will keep one row of a `NULL` value.

In database theory, if a column contains `NULL` values, it means that we do not have the information about that column of particular records or the information is not applicable.

For example, if a customer has a phone number with a `NULL` value, it means we don't have information about the phone number of the customer at the time of recording customer information or the customer may not have a phone number at all.

SQLite `SELECT DISTINCT` examples

We will use the customers table in the [sample database](#) for demonstration.

```
CREATE TABLE COMPANY(  
  ID INT PRIMARY KEY NOT NULL,  
  NAME TEXT NOT NULL,  
  AGE INT NOT NULL,  
  ADDRESS CHAR(50),  
  SALARY REAL,  
  CITY TEXT  
);
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (1, 'Paul', 32, 'California', 20000.00 , 'SURAT');
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (2, 'Allen', 25, 'Texas', 15000.00,'MUMBAI');
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (3, 'Teddy', 23, 'Norway', 20000.00, 'SURAT');
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00, 'MUMBAI');
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (5, 'David', 27, 'Texas', 85000.00, 'SURAT');
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,CITY)  
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00, 'MUMBAI');
```

Suppose you want to know the cities where the customers locate, you can use the SELECT statement to get data from the city column of the customers table as follows:

```
SELECT city  
FROM customers  
ORDER BY city;
```

To remove these duplicate rows, you use the DISTINCT clause as follows:

```
SELECT DISTINCT city  
FROM customers  
ORDER BY city;
```

SQLite SELECT DISTINCT on multiple columns

The following statement finds cities and countries of all customers.

```
SELECT
```

```
    city,  
    country  
FROM  
    customers  
ORDER BY  
    country;
```

The result set contains duplicate city and country e.g., Sao Paulo in Brazil as shown in the screenshot above.

To remove duplicate the city and country, you apply the `DISTINCT` clause to both city and country columns as shown in the following query:

```
SELECT DISTINCT  
    city,  
    country  
FROM  
    customers  
ORDER BY  
    country;
```

As mentioned earlier, SQLite uses the combination of city and country to evaluate the duplicate.

SQLite `SELECT DISTINCT` with `NULL` example

This statement returns the names of companies of customers from the customers table.

```
SELECT company  
FROM customers;
```

Now, if you apply the `DISTINCT` clause to the statement, it will keep only one row with a `NULL` value.

See the following statement:

```
SELECT DISTINCT company  
FROM customers;
```

Note that if you select a list of columns from a table and want to get a unique combination of some columns, you can use the [GROUP BY](#) clause.

SQLite Where

Introduction to SQLite WHERE clause

The WHERE clause is an optional clause of the [SELECT](#) statement. It appears after the FROM clause as the following statement:

```
SELECT
    column list
FROM
    table
WHERE
    search condition;
```

In this example, you add a WHERE clause to the SELECT statement to filter rows returned by the query. When evaluating a SELECT statement with a WHERE clause, SQLite uses the following steps:

1. First, check the table in the FROM clause.
2. Second, evaluate the conditions in the WHERE clause to get the rows that met these conditions.
3. Third, make the final result set based on the rows in the previous step with columns in the SELECT clause.

The search condition in the WHERE has the following form:

```
left expression COMPARISON OPERATOR right expression
```

For example, you can form a search condition as follows:

```
WHERE column 1 = 100;
```

```
WHERE column 2 IN (1,2,3);
```

```
WHERE column 3 LIKE 'An%';
```

```
WHERE column 4 BETWEEN 10 AND 20;
```

Besides the SELECT statement, you can use the WHERE clause in the [UPDATE](#) and [DELETE](#) statements.

SQLite comparison operators

A comparison operator tests if two expressions are the same. The following table illustrates the comparison operators that you can use to construct expressions:

Operator	Meaning

=	Equal to
<> or !=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

SQLite logical operators

Logical operators allow you to test the truth of some expressions. A logical operator returns 1, 0, or a NULL value.

Notice that SQLite does not provide Boolean data type therefore 1 means TRUE, and 0 means FALSE.

The following table illustrates the SQLite logical operators:

Operator	Meaning
ALL	returns 1 if all expressions are 1.
AND	returns 1 if both expressions are 1, and 0 if one of the expressions is 0.
ANY	returns 1 if any one of a set of comparisons is 1.
BETWEEN	returns 1 if a value is within a range.
EXISTS	returns 1 if a subquery contains any rows.
IN	returns 1 if a value is in a list of values.
LIKE	returns 1 if a value matches a pattern
NOT	reverses the value of other operators such as NOT EXISTS, NOT IN, NOT BETWEEN, etc.

OR	returns true if either expression is 1
----	----------------------------------------

SQLite WHERE clause examples

We will use the tracks table in the [sample database](#) to illustrate how to use the WHERE clause.

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

The equality operator (=) is the most commonly used operator. For example, the following query uses the WHERE clause the equality operator to find all the tracks in the album id 1:

```
SELECT
    name,
    milliseconds,
    bytes,
    albumid
FROM
    tracks
WHERE
    albumid = 1;
```

Name	Milliseconds	Bytes	AlbumId
▶ For Those About To Rock (We Salute You)	343719	11170334	1
Put The Finger On You	205662	6713451	1
Let's Get It Up	233926	7636561	1
Inject The Venom	210834	6852860	1
Snowballed	203102	6599424	1
Evil Walks	263497	8611245	1
C.O.D.	199836	6566314	1
Breaking The Rules	263288	8596840	1
Night Of The Long Knives	205688	6706347	1
Spellbound	270863	8817038	1

SQLite compares the values stored in the AlbumId column with a literal value 1 to test if they are equal. Only the rows that satisfy the condition are returned.

When you compare two values, you must ensure that they are the same data type. You should compare numbers with numbers, string with strings, etc.

In case you compare values in different data types e.g., a string with a number, SQLite has to perform implicit data type conversions, but in general, you should avoid doing this.

You use the logical operator to combine expressions. For example, to get tracks of the album 1 that have the length greater than 200,000 milliseconds, you use the following statement:

```
SELECT
    name,
    milliseconds,
    bytes,
    albumid
FROM
    tracks
WHERE
    albumid = 1
AND milliseconds > 250000;
```

Name	Milliseconds	Bytes	AlbumId
▶ For Those About To Rock (We Salute You)	343719	11170334	1
Evil Walks	263497	8611245	1
Breaking The Rules	263288	8596840	1
Spellbound	270863	8817038	1

The statement used two expressions `albumid = 1` and `milliseconds > 250000`. It uses the AND logical operator to combine these expressions.

Name	Milliseconds	Bytes	AlbumId
Evil Walks	263497	8611245	1
Breaking The Rules	263288	8596840	1
Spellbound	270863	8817038	1
Restless and Wild	252051	4331779	3
Bad Boy Boogie	267728	8776140	4
Hell Ain't A Bad Place To Be	254380	8331286	4
Walk On Water	295680	9719579	5
Rag Doll	264698	8675345	5
Dude (Looks Like A Lady)	264855	8679940	5
Eat The Rich	251036	8262039	5
All I Really Want	284891	9375567	6
Head Over Feet	267493	8758008	6
Mary Jane	280607	9163588	6
Wake Up	293485	9703359	6

SQLite WHERE clause with LIKE operator example

Sometimes, you may not remember exactly the data that you want to search. In this case, you perform an inexact search using the [LIKE](#) operator.

For example, to find which tracks composed by Smith, you use the LIKE operator as follows:

```
SELECT
    name,
    albumid,
    composer
FROM
    tracks
WHERE
    composer LIKE '%Smith%'
ORDER BY
    albumid;
```

Name	AlbumId	Composer
Restless and Wild	3	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. Dirksnei
Princess of the Dawn	3	Deaffy & R.A. Smith-Diesel
Killing Floor	19	Adrian Smith
Machine Men	19	Adrian Smith
2 Minutes To Midnight	95	Adrian Smith, Bruce Dickinson
Can I Play With Madness	96	Adrian Smith, Bruce Dickinson/Steve Harris
The Evil That Men Do	96	Adrian Smith, Bruce Dickinson/Steve Harris
The Wicker Man	97	Adrian Smith, Bruce Dickinson/Steve Harris
The Fallen Angel	97	Adrian Smith, Steve Harris
Wildest Dreams	98	Adrian Smith, Steve Harris
New Frontier	98	Adrian Smith, Bruce Dickinson/Nicko McBrain

You get tracks composed by R.A. Smith-Diesel, Adrian Smith, etc.

SQLite WHERE clause with the IN operator example

The [IN](#) operator allows you to check whether a value is in a list of a comma-separated list of values. For example, to find tracks that have media type id is 2 or 3, you use the IN operator as shown in the following statement:

```
SELECT
    name,
    albumid,
    mediatypeid
FROM
    tracks
WHERE
    mediatypeid IN (4, 5);
```

Name	AlbumId	MediaTypeId
► Symphony No. 2, Op. 16 - "The Four Temperam	338	2
24 Caprices, Op. 1, No. 24, for Solo Violin, in A M	339	2
Erlkonig, D.328	341	2
Pini Di Roma (Pinien Von Rom) \ I Pini Della Via /	343	2
String Quartet No. 12 in C Minor, D. 703 "Quartet	344	2
L'orfeo, Act 3, Sinfonia (Orchestra)	345	2
Quintet for Horn, Violin, 2 Violas, and Cello in E F	346	2
Koyaanisqatsi	347	2
Battlestar Galactica: The Story So Far	226	3
Occupation / Precipice	227	3
Exodus. Pt. 1	227	3

SQLite Limit

Introduction to SQLite LIMIT clause

The LIMIT clause is an optional part of the [SELECT](#) statement. You use the LIMIT clause to constrain the number of rows returned by the query.

For example, a SELECT statement may return one million rows. However, if you just need the first 10 rows in the result set, you can add the LIMIT clause to the SELECT statement to retrieve 10 rows.

The following illustrates the syntax of the LIMIT clause.

```
SELECT
    column list
```

```
FROM  
    table  
LIMIT row_count;
```

The `row_count` is a positive integer that specifies the number of rows returned.

For example, to get the first 10 rows in the `tracks` table, you use the following statement:

```
SELECT  
    trackId,  
    name  
FROM  
    tracks  
LIMIT 10;
```

TrackId	Name
1	For Those About To Rock (We Salute You)
2	Balls to the Wall
3	Fast As a Shark
4	Restless and Wild
5	Princess of the Dawn
6	Put The Finger On You
7	Let's Get It Up
8	Inject The Venom
9	Snowballed
10	Evil Walks

If you want to get the first 10 rows starting from the 10th row of the result set, you use `OFFSET` keyword as the following:

```
SELECT  
    column list  
FROM  
    table  
LIMIT row count OFFSET offset;
```

Or you can use the following shorthand syntax of the `LIMIT OFFSET` clause:

```
SELECT  
    column list  
FROM  
    table  
LIMIT offset, row_count;
```

For example, to get 10 rows starting from the 11th row in the `tracks` table, you use the following statement:

```
SELECT
    trackId,
    name
FROM
    tracks
LIMIT 10 OFFSET 10;
```

TrackId	Name
11	C.O.D.
12	Breaking The Rules
13	Night Of The Long Knives
14	Spellbound
15	Go Down
16	Dog Eat Dog
17	Let There Be Rock
18	Bad Boy Boogie
19	Problem Child
20	Overdose

You often find the uses of `OFFSET` in web applications for paginating result sets.

SQLite `LIMIT` and `ORDER BY` clause

You should always use the `LIMIT` clause with the `ORDER BY` clause. Because you want to get a number of rows in a specified order, not in an unspecified order.

The `ORDER BY` clause appears before the `LIMIT` clause in the `SELECT` statement. SQLite sorts the result set before getting the number of rows specified in the `LIMIT` clause.

```
SELECT
    column list
FROM
    table
ORDER BY column 1
LIMIT row count;
```

For example, to get the top 10 biggest tracks by size, you use the following query:

```
SELECT
    trackid,
    name,
    bytes
FROM
    tracks
ORDER BY
    bytes DESC
LIMIT 10;
```


TrackId	Name	Bytes
3224	Through a Looking Glass	1059546140
2820	Occupation / Precipice	1054423946
3236	The Young Lords	587051735
3242	The Man With Nine Lives	577829804
2910	Dave	574325829
3235	The Magnificent Warriors	570152232
3231	The Lost Warrior	558872190
2902	Maternity Leave	555244214
3228	Battlestar Galactica, Pt. 3	554509033
2832	The Woman King	552893447

To get the 5 shortest tracks, you sort the tracks by the length specified by milliseconds column using ORDER BY clause and get the first 5 rows using LIMIT clause.

```
SELECT
    trackid,
    name,
    milliseconds
FROM
    tracks
ORDER BY
    milliseconds ASC
LIMIT 5;
```

TrackId	Name	Milliseconds
2461	É Uma Partida De Futebol	1071
168	Now Sports	4884
170	A Statistic	6373
178	Oprah	6635
3304	Commercial 1	7941

Getting the nth highest and the lowest value

You can use the ORDER BY and LIMIT clauses to get the nth highest or lowest value rows. For example, you may want to know the second-longest track, the third smallest track, etc.

To do this, you use the following steps:

1. First, use ORDER BY to sort the result set in ascending order in case you want to get the nth lowest value, or descending order if you want to get the nth highest value.
2. Second, use the LIMIT OFFSET clause to get the nth highest or the nth lowest row.

The following statement returns the second-longest track in the tracks table.


```
SELECT
    trackid,
    name,
    milliseconds
FROM
    tracks
ORDER BY
    milliseconds DESC
LIMIT 1 OFFSET 1;
```

TrackId	Name	Milliseconds
▶ 3224	Through a Looking Glass	5088838

The following statement gets the third smallest track on the tracks table.

```
SELECT
    trackid,
    name,
    bytes
FROM
    tracks
ORDER BY
    bytes
LIMIT 1 OFFSET 2;
```

TrackId	Name	Bytes
▶ 170	A Statistic	211997

In this tutorial, you have learned how to use SQLite LIMIT clause to constrain the number of rows returned by the query.

SQLite BETWEEN

Summary: in this tutorial, you will learn how to use the SQLite BETWEEN operator to test whether a value is in a range of values.

Introduction to SQLite BETWEEN Operator

The BETWEEN operator is a logical operator that tests whether a value is in range of values. If the value is in the specified range, the BETWEEN operator returns true.

The BETWEEN operator can be used in the [WHERE](#) clause of the [SELECT](#), [DELETE](#), [UPDATE](#), and [REPLACE](#) statements.

The following illustrates the syntax of the SQLite BETWEEN operator:

```
test expression BETWEEN low expression AND high expression
```

In this syntax:

- `test_expression` is an expression to test for in the range defined by `low_expression` and `high_expression`.
- `low_expression` and `high_expression` is any valid expression that specify the low and high values of the range. The `low_expression` should be less than or equal to `high_expression`, or the `BETWEEN` is always returns false.
- The `AND` keyword is a placeholder which indicates the `test_expression` should be within the range specified by `low_expression` and `high_expression`.

Note that the `BETWEEN` operator is inclusive. It returns true when the `test_expression` is less than or equal to `high_expression` and greater than or equal to the value of `low_expression`:

```
test_expression >= low_expression AND test_expression <=
high_expression
```

To specify an exclusive range, you use the greater than (`>`) and less than operators (`<`).

Note that if any input to the `BETWEEN` operator is `NULL`, the result is `NULL`, or unknown to be precise.

To negate the result of the `BETWEEN` operator, you use the `NOT BETWEEN` operator as follows:

```
test_expression NOT BETWEEN low_expression AND high_expression
```

The `NOT BETWEEN` returns true if the value of `test_expression` is less than the value of `low_expression` or greater than the value of `high_expression`:

```
test_expression < low_expression OR test_expression >
high_expression
```

SQLite `BETWEEN` operator examples

invoices
* InvoiceId
CustomerId
InvoiceDate
BillingAddress
BillingCity
BillingState
BillingCountry
BillingPostalCode
Total

SQLite BETWEEN numeric values example

The following statement finds invoices whose total is between 14.96 and 18.86:

```
SELECT
    InvoiceId,
    BillingAddress,
    Total
FROM
    invoices
WHERE
    Total BETWEEN 14.91 and 18.86
ORDER BY
    Total;
```

InvoiceId	BillingAddress	Total
193	Berger Straße 10	14.91
103	162 E Superior Street	15.86
208	Ullevålsveien 14	15.86
306	Klanova 9/506	16.86
313	68, Rue Jouvence	16.86
88	Calle Lira, 198	17.91
89	Rotenturmstraße 4, 1010 Innere Stadt	18.86
201	319 N. Frances Street	18.86

As you can see, the invoices whose total is 14.91 or 18.86 are included in the result set.

SQLite NOT BETWEEN numeric values example

To find the invoices whose total are not between 1 and 20, you use the NOT BETWEEN operator as shown in the following query:

```
SELECT
    InvoiceId,
    BillingAddress,
```

```

Total
FROM
invoices
WHERE
Total NOT BETWEEN 1 and 20
ORDER BY
Total;

```

The following picture shows the output:

InvoiceId	BillingAddress	Total
6	Berger Straße 10	0.99
13	1600 Amphitheatre Parkway	0.99
20	110 Raeburn Pl	0.99
27	5112 48 Street	0.99
34	Praça Pio X, 119	0.99
41	C/ San Bernardo 85	0.99
48	796 Dundas Street West	0.99
55	Grétrystraat 63	0.99
62	3 Chatham Street	0.99
69	319 N. Frances Street	0.99
76	Ullevålsveien 14	0.99
83	9, Place Louis Barthou	0.99
90	801 W 4th Street	0.99
104	Barbarossastraße 19	0.99
111	1 Microsoft Way	0.99
118	421 Bourke Street	0.99
125	Rua da Assunção 53	0.99
132	Qe 7 Bloco G	0.99
139	Celsiusg. 9	0.99
146	230 Elgin Street	0.99
153	Sønder Boulevard 51	0.99
160	Via Degli Scipioni, 43	0.99
167	2211 W Berry Street	0.99
174	Klanova 9/506	0.99
181	68, Rue Jouvence	0.99
188	120 S Orange Ave	0.99
195	Av. Brigadeiro Faria Lima, 2170	0.99
209	627 Broadway	0.99
216	307 Macacha Güemes	0.99
223	Rua dos Campeões Europeus de Viena, 4350	0.99
230	8210 111 ST NW	0.99
237	202 Hoxton Street	0.99
244	194A Chain Lake Drive	0.99
251	Rua Dr. Falcão Filho, 155	0.99
258	Lijnbaansgracht 120bg	0.99
265	1033 N Park Ave	0.99
272	Rilská 3174/6	0.99
279	Porthaninkatu 9	0.99
286	69 Salem Street	0.99
293	Theodor-Heuss-Straße 34	0.99
299	8, Rue de la...	0.99

As clearly shown in the output, the result includes the invoices whose total is less than 1 and greater than 20.

SQLite BETWEEN dates example

The following example finds invoices whose invoice dates are from January 1 2010 and January 31 2010:

```
SELECT
    InvoiceId,
    BillingAddress,
    InvoiceDate,
    Total
FROM
    invoices
WHERE
    InvoiceDate BETWEEN '2010-01-01' AND '2010-01-31'
ORDER BY
    InvoiceDate;
```

InvoiceId	BillingAddress	InvoiceDate	Total
84	68, Rue Jouvence	2010-01-08 00:00:00	1.98
85	Erzsébet krt. 58.	2010-01-08 00:00:00	1.98
86	Via Degli Scipioni, 43	2010-01-09 00:00:00	3.96
87	Celsiusg. 9	2010-01-10 00:00:00	6.94
88	Calle Lira, 198	2010-01-13 00:00:00	17.91
89	Rotenturmstraße 4, 1010 Innere Stadt	2010-01-18 00:00:00	18.86
90	801 W 4th Street	2010-01-26 00:00:00	0.99

SQLite NOT BETWEEN dates example

The following statement finds invoices whose dates are not between January 03, 2009, and December 01, 2013:

```
SELECT
    InvoiceId,
    BillingAddress,
    date(InvoiceDate) InvoiceDate,
    Total
FROM
    invoices
WHERE
    InvoiceDate NOT BETWEEN '2009-01-03' AND '2013-12-01'
ORDER BY
    InvoiceDate;
```

The output is as follows:

InvoiceId	BillingAddress	InvoiceDate	Total
1	Theodor-Heuss-Straße 34	2009-01-01	1.98
2	Ullevålsveien 14	2009-01-02	3.96
406	801 W 4th Street	2013-12-04	1.98
407	69 Salem Street	2013-12-04	1.98
408	319 N. Frances Street	2013-12-05	3.96
409	796 Dundas Street West	2013-12-06	5.94
410	Rua dos Campeões Europeus de Viena, 4350	2013-12-09	8.91
411	Porthaninkatu 9	2013-12-14	13.86
412	12, Community Centre	2013-12-22	1.99

SQLite IN

Summary: in this tutorial, you will learn how to use the SQLite IN operator to determine whether a value matches any value in a list of values or a result of a subquery.

Introduction to the SQLite IN operator

The SQLite IN operator determines whether a value matches any value in a list or a [subquery](#). The syntax of the IN operator is as follows:

```
expression [NOT] IN (value_list|subquery);
```

The expression can be any valid expression or a column of a table.

A list of values is a fixed value list or a result set of a single column returned by a subquery. The returned [type](#) of expression and values in the list must be the same.

The IN operator returns true or false depending on whether the expression matches any value in a list of values or not. To negate the list of values, you use the NOT IN operator.

SQLite IN operator examples

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

The following statement uses the IN operator to query the tracks whose media type id is 1 or 2.

```
SELECT
    TrackId,
    Name,
    MediaTypeId
FROM
    Tracks
WHERE
    MediaTypeId IN (4, 5)
ORDER BY
    Name ASC;
```

TrackId	Name	MediaTypeId
3027	"40"	1
3412	"Eine Kleine Nachtmusik" Serenade In G, K. 525: I. Allegro	2
109	#1 Zero	1
3254	#9 Dream	2
602	'Round Midnight	1
1833	(Anesthesia) Pulling Teeth	1
570	(Da Le) Yaleo	1
3045	(I Can't Help) Falling In Love With You	1

This query uses the OR operator instead of the IN operator to return the same result set as the above query:

```
SELECT
    TrackId,
    Name,
    MediaTypeId
FROM
    Tracks
```

```
WHERE
    MediaTypeId = 1 OR MediaTypeId = 2
ORDER BY
    Name ASC;
```

As you can see from the queries, using the IN operator is much shorter.

If you have a query that uses many OR operators, you can consider using the IN operator instead to make the query more readable.

SQLite IN operator with a subquery example

The following query returns a list of album id of the artist id 12:

```
SELECT albumid
FROM albums
WHERE artistid = 12;
```

AlbumId
16
17

To get the tracks that belong to the artist id 12, you can combine the IN operator with a [subquery](#) as follows:

```
SELECT
    TrackId,
    Name,
    AlbumId
FROM
    Tracks
WHERE
    AlbumId IN (
        SELECT
            AlbumId
        FROM
            Albums
        WHERE
            ArtistId = 12
    );
```


TrackId	Name	AlbumId
149	Black Sabbath	16
150	The Wizard	16
151	Behind The Wall Of Sleep	16
152	N.I.B.	16
153	Evil Woman	16
154	Sleeping Village	16
155	Warning	16
156	Wheels Of Confusion / The Straightener	17
157	Tomorrow's Dream	17
158	Changes	17
159	FX	17
160	Supernaut	17
161	Snowblind	17
162	Cornucopia	17

In this example:

- First, the subquery returns a list of album ids that belong to the artist id 12.
- Then, the outer query return all tracks whose album id matches with the album id list returned by the subquery.

SQLite NOT IN examples

The following statement returns a list of tracks whose genre id is not in a list of (1,2,3).

```
SELECT
    trackid,
    name,
    genreid
FROM
    tracks
WHERE
    genreid NOT IN (1, 2, 3);
```

TrackId	Name	GenreId
99	Your Time Has Come	4
100	Out Of Exile	4
101	Be Yourself	4
102	Doesn't Remind Me	4
103	Drown Me Slowly	4
104	Heaven's Dead	4
105	The Worm	4
106	Man Or Animal	4
107	Yesterday To Tomorrow	4
108	Dandelion	4
109	#1 Zero	4

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;
```

Code language: SQL (Structured Query Language) (sql)

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%'
```

Code language: SQL (Structured Query Language) (sql)

[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'
```

Code language: SQL (Structured Query Language) (sql)

[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%';
```

Code language: SQL (Structured Query Language) (sql)

[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%';
```

Code language: SQL (Structured Query Language) (sql)

[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;
```

Code language: SQL (Structured Query Language) (sql)

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 '\';
```

Code language: SQL (Structured Query Language) (sql)

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  
);
```

Code language: SQL (Structured Query Language) (sql)

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');
```

Code language: SQL (Structured Query Language) (sql)

Then, query data from the t table:

```
SELECT * FROM t;
```

Code language: SQL (Structured Query Language) (sql)

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

Code language: Shell Session (shell)

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%';
```

Code language: SQL (Structured Query Language) (sql)

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 '\';
```

Code language: SQL (Structured Query Language) (sql)

Here is the result set:

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

SQLite IS NULL

Summary: in this tutorial, you will learn how to use the SQLite IS NULL and IS NOT NULL operators to check whether a value is NULL or not.

Introduction to the SQLite IS NULL operator

NULL is special. It indicates that a piece of information is unknown or not applicable.

For example, some songs may not have the songwriter information because we don't know who wrote them.

To store these unknown songwriters along with the songs in a database table, we must use NULL.

NULL is not equal to anything even the number zero, an empty string, and so on.

Especially, NULL is not equal to itself. The following expression returns 0:

```
NULL = NULL
```

Code language: SQL (Structured Query Language) (sql)

This is because two unknown information cannot be comparable.

Let's see the following tracks table from the [sample database](#):

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

The following statement attempts to find tracks whose composers are NULL:

```
SELECT
    Name,
    Composer
FROM
    tracks
WHERE
    Composer = NULL;
```

Code language: SQL (Structured Query Language) (sql)

It returns an empty row without issuing any additional message.

This is because the following expression always evaluates to false:

```
Composer = NULL
```

Code language: SQL (Structured Query Language) (sql)

It's not valid to use the NULL this way.

To check if a value is NULL or not, you use the IS NULL operator instead:

```
{ column | expression } IS NULL;
```

Code language: SQL (Structured Query Language) (sql)

The IS NULL operator returns 1 if the column or expression evaluates to NULL.

To find all tracks whose composers are unknown, you use the IS NULL operator as shown in the following query:

```
SELECT
    Name,
    Composer
FROM
```



```

tracks
WHERE
    Composer IS NULL
ORDER BY
    Name;

```

Code language: SQL (Structured Query Language) (sql)

Here is the partial output:

Name	Composer
"?"	[NULL]
#9 Dream	[NULL]
(I Can't Help) Falling In Love With You	[NULL]
...And Found	[NULL]
...In Translation	[NULL]
.07%	[NULL]
100% HardCore	[NULL]
13 Years Of Grief	[NULL]
16 Toneladas	[NULL]
1° De Julho	[NULL]
A Banda	[NULL]
A Bencao E Outros	[NULL]
A Benihana Christmas, Pts. 1 & 2	[NULL]

SQLite IS NOT NULL operator

The NOT operator negates the IS NULL operator as follows:

```

expression | column IS NOT NULL

```

Code language: SQL (Structured Query Language) (sql)

The IS NOT NULL operator returns 1 if the expression or column is not NULL, and 0 if the expression or column is NULL.

The following example finds tracks whose composers are not NULL:

```

SELECT
    Name,
    Composer
FROM
    tracks
WHERE
    Composer IS NOT NULL
ORDER BY
    Name;

```

SQLite GLOB

Summary: in this tutorial, you will learn how to use the SQLite `GLOB` operator to determine whether a string matches a specific pattern.

Introduction to the SQLite `GLOB` operator

The `GLOB` operator is similar to the [LIKE](#) operator. The `GLOB` operator determines whether a string matches a specific pattern.

Unlike the `LIKE` operator, the `GLOB` operator is **case sensitive** and uses the **UNIX wildcards**. In addition, the `GLOB` patterns do not have escape characters.

The following shows the wildcards used with the `GLOB` operator:

- The asterisk (*) wildcard matches any number of characters.
- The question mark (?) wildcard matches exactly one character.

On top of these wildcards, you can use the list wildcard [] to match one character from a list of characters. For example [xyz] match any single x, y, or z character.

The list wildcard also allows a range of characters e.g., [a-z] matches any single lowercase character from a to z. The [a-zA-Z0-9] pattern matches any single alphanumeric character, both lowercase, and uppercase.

Besides, you can use the character ^ at the beginning of the list to match any character except for any character in the list. For example, the [^0-9] pattern matches any single character except a numeric character.

SQLite `GLOB` examples

The following statement finds tracks whose names start with the string Man. The pattern Man* matches any string that starts with Man.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB 'Man*';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
52	Man In The Box
106	Man Or Animal
257	Manguetown
358	Man With The Woman Head
562	Mangueira
584	Manuel
1397	Man On The Edge
1459	Manifest Destiny
1480	Manic Depression
2046	Manguetown
2859	Man of Science, Man of Faith (Premiere)

The following statement gets the tracks whose names end with Man. The pattern *Man matches any string that ends with Man.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB '*Man';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
31	Blind Man
359	Muffin Man
431	The Invisible Man
760	Hard Lovin' Man
809	Holy Man
821	Ramshackle Man
836	Make Love Like A Man

The following query finds the tracks whose names start with any single character (?), followed by the string ere and then any number of character (*).

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB '?ere*';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
324	Pererê
1132	Serenity
1452	Were Do We Go From Here
1740	Sereia
2198	Jeremy
2479	Here's To The Atom Bomb
2791	Hereditário
3042	Here I Am (Come And Take Me)
3133	Here I Go Again

To find the tracks whose names contain numbers, you can use the list wildcard [0-9] as follows:

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB '*[1-9]*';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
109	#1 Zero
122	20 Flight Rock
132	13 Years Of Grief
343	Communication Breakdown(2)
347	Communication Breakdown(3)
348	I Can't Quit You Baby(2)

Or to find the tracks whose name does not contain any number, you place the character ^ at the beginning of the list:

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB '*[^1-9]*';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
1	For Those About To Rock (We Salute You)
2	Balls to the Wall
3	Fast As a Shark
4	Restless and Wild
5	Princess of the Dawn
6	Put The Finger On You
7	Let's Get It Up
8	Inject The Venom

The following statement finds the tracks whose names end with a number.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB '*[1-9]';
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name
360	Vai-Vai 2001
361	X-9 2001
362	Gavioes 2001
363	Nene 2001
364	Rosas De Ouro 2001
365	Mocidade Alegre 2001
366	Camisa Verde 2001

Set operators

SQLite Union

Summary: in this tutorial, you will learn how to use SQLite UNION operator to combine result sets of two or more queries into a single result set.

Introduction to SQLite UNION operator

Sometimes, you need to combine data from multiple tables into a complete result set. It may be for tables with similar data within the same database or maybe you need to combine similar data from multiple databases.

To combine rows from two or more [queries](#) into a single result set, you use SQLite UNION operator. The following illustrates the basic syntax of the UNION operator:

```
query 1
UNION [ALL]
query 2
UNION [ALL]
query 3
...;
```

Code language: SQL (Structured Query Language) (sql)

Both UNION and UNION ALL operators combine rows from result sets into a single result set. The UNION operator removes eliminate duplicate rows, whereas the UNION ALL operator does not.

Because the UNION ALL operator does not remove duplicate rows, it runs faster than the UNION operator.

The following are rules to union data:

- The number of columns in all queries must be the same.
- The corresponding columns must have compatible data types.
- The column names of the first query determine the column names of the combined result set.
- The [GROUP BY](#) and [HAVING](#) clauses are applied to each individual query, not the final result set.
- The [ORDER BY](#) clause is applied to the combined result set, not within the individual result set.

Note that the difference between UNION and JOIN e.g., [INNER JOIN](#) or [LEFT JOIN](#) is that the JOIN clause combines *columns* from multiple related tables, while UNION combines *rows* from multiple similar tables.

Suppose we have two tables t1 and t2 with the following structures:

```
CREATE TABLE t1(  
  v1 INT  
);
```

```
INSERT INTO t1(v1)  
VALUES (1), (2), (3);
```

```
CREATE TABLE t2(  
  v2 INT  
);  
INSERT INTO t2(v2)  
VALUES (2), (3), (4);
```

Code language: SQL (Structured Query Language) (sql)

The following statement combines the result sets of the t1 and t2 table using the UNION operator:

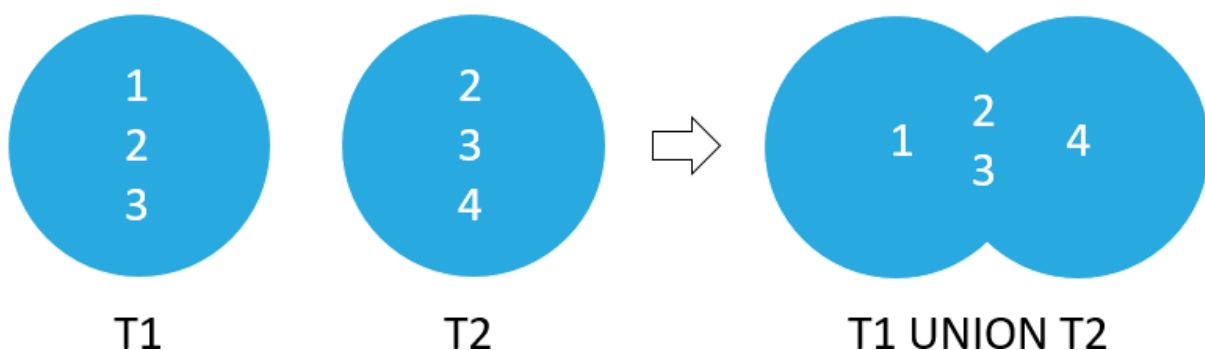
```
SELECT v1 as 'Union'  
FROM t1  
UNION  
SELECT v2  
FROM t2;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

v1
1
2
3
4

The following picture illustrates the UNION operation of t1 and t2 tables:



The following statement combines the result sets of t1 and t2 table using the UNION ALL operator:

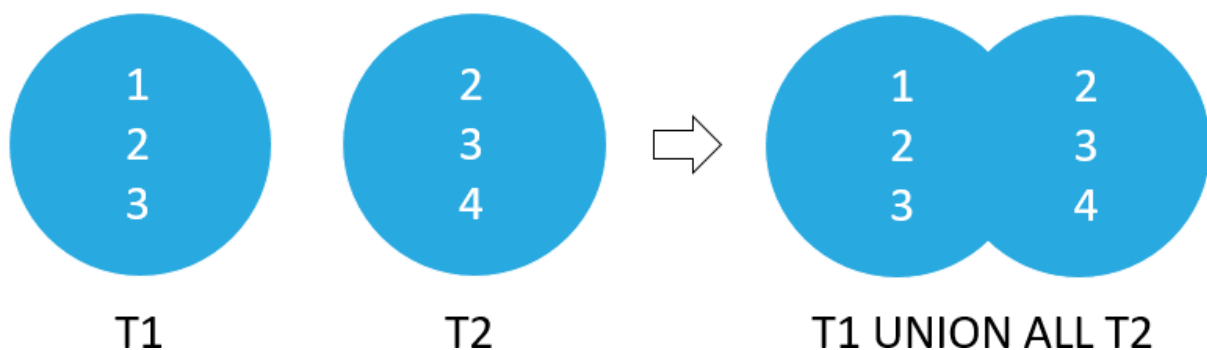
```
SELECT v1
FROM t1
UNION ALL
SELECT v2
FROM t2;
```

Code language: SQL (Structured Query Language) (sql)

The following picture shows the output:

v1
1
2
3
2
3
4

The following picture illustrates the UNION ALL operation of the result sets of t1 and t2 tables:



SQLite UNION examples

Let's take some examples of using the UNION operator.

1) SQLite UNION example

This statement uses the UNION operator to combine names of employees and customers into a single list:

```
SELECT FirstName, LastName, 'Employee' AS Type
FROM employees
UNION
SELECT FirstName, LastName, 'Customer'
FROM customers;
```


Code language: SQL (Structured Query Language) (sql)

Here is the output:

FirstName	LastName	Type
Aaron	Mitchell	Customer
Alexandre	Rocha	Customer
Andrew	Adams	Employee
Astrid	Gruber	Customer
Bjørn	Hansen	Customer
Camille	Bernard	Customer
Daan	Peeters	Customer
Dan	Miller	Customer
Diego	Gutiérrez	Customer
Dominique	Lefebvre	Customer
Eduardo	Martins	Customer
Edward	Francis	Customer
Ellie	Sullivan	Customer
Emma	Jones	Customer
Enrique	Muñoz	Customer

2) SQLite UNION with ORDER BY example

This example uses the UNION operator to combine the names of the employees and customers into a single list. In addition, it uses the ORDER BY clause to sort the name list by first name and last name.

```
SELECT FirstName, LastName, 'Employee' AS Type
FROM employees
UNION
SELECT FirstName, LastName, 'Customer'
FROM customers
ORDER BY FirstName, LastName;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

FirstName 	LastName 	Type 
Aaron	Mitchell	Customer
Alexandre	Rocha	Customer
Andrew	Adams	Employee
Astrid	Gruber	Customer
Bjørn	Hansen	Customer
Camille	Bernard	Customer
Daan	Peeters	Customer
Dan	Miller	Customer
Diego	Gutiérrez	Customer
Dominique	Lefebvre	Customer
Eduardo	Martins	Customer
Edward	Francis	Customer
Ellie	Sullivan	Customer
Emma	Jones	Customer
Enrique	Muñoz	Customer
Fernanda	Ramos	Customer

SQLite Except

Summary: in this tutorial, you will learn how to use the SQLite EXCEPT operator.

Introduction to SQLite EXCEPT operator

SQLite EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

The following shows the syntax of the EXCEPT operator:

```
SELECT select list1
FROM table1
EXCEPT
SELECT select list2
FROM table2
```

Code language: SQL (Structured Query Language) (sql)

This query must conform to the following rules:

- First, the number of columns in the select lists of both queries must be the same.
- Second, the order of the columns and their types must be comparable.

The following statements [create two tables](#) t1 and t2 and [insert](#) some data into both tables:

```
CREATE TABLE t1(
```

```
v1 INT  
);
```

```
INSERT INTO t1 (v1)  
VALUES (1), (2), (3);
```

```
CREATE TABLE t2 (  
v2 INT  
);
```

```
INSERT INTO t2 (v2)  
VALUES (2), (3), (4);
```

Code language: SQL (Structured Query Language) (sql)

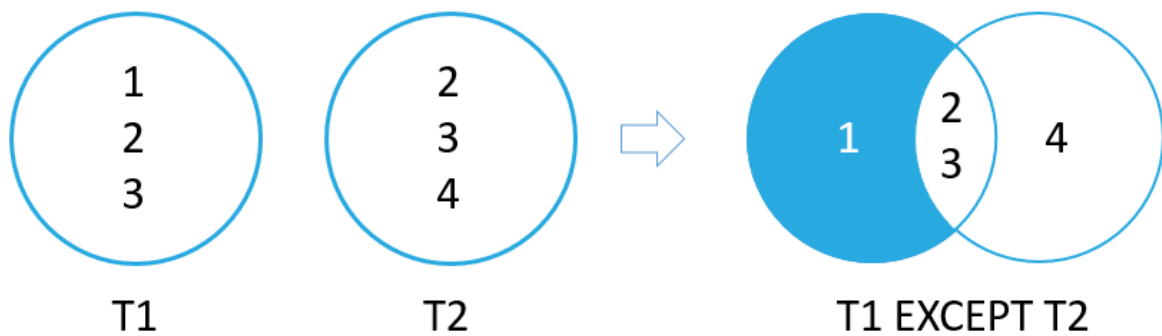
The following statement illustrates how to use the EXCEPT operator to compare result sets of two queries:

```
SELECT v1  
FROM t1  
EXCEPT  
SELECT v2  
FROM t2;
```

Code language: SQL (Structured Query Language) (sql)

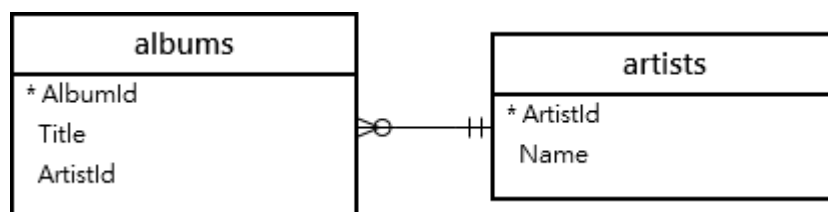
The output is 1.

The following picture illustrates the EXCEPT operation:



SQLite EXCEPT examples

We will use the artists and albums tables from the [sample database](#) for the demonstration.



The following statement finds artist ids of artists who do not have any album in the albums table:

```
SELECT ArtistId
FROM artists
EXCEPT
SELECT ArtistId
FROM albums;
```

Code language: SQL (Structured Query Language) (sql)

The output is as follows:

ArtistId
25
26
28
29
30
31
32
33
34

SQLite Intersect

Summary: in this tutorial, you will learn how to use the SQLite INTERSECT operator.

Introduction to SQLite INTERSECT operator

SQLite INTERSECT operator compares the result sets of two [queries](#) and returns distinct rows that are output by both queries.

The following illustrates the syntax of the INTERSECT operator:

```
SELECT select list1
FROM table1
INTERSECT
SELECT select list2
FROM table2
```

Code language: SQL (Structured Query Language) (sql)

The basic rules for combining the result sets of two queries are as follows:

- First, the number and the order of the columns in all queries must be the same.
- Second, the data types must be comparable.

For the demonstration, we will [create two tables](#) t1 and t2 and [insert some data](#) into both:

```
CREATE TABLE t1 (  
  v1 INT  
);
```

```
INSERT INTO t1(v1)  
VALUES (1), (2), (3);
```

```
CREATE TABLE t2 (  
  v2 INT  
);  
INSERT INTO t2(v2)  
VALUES (2), (3), (4);
```

Code language: SQL (Structured Query Language) (sql)

The following statement illustrates how to use the INTERSECT operator to compare result sets of two queries:

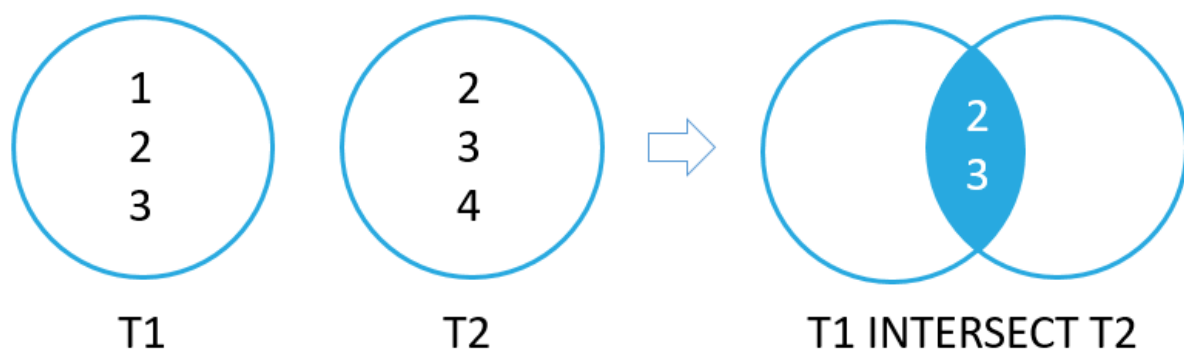
```
SELECT v1  
FROM t1  
INTERSECT  
SELECT v2  
FROM t2;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

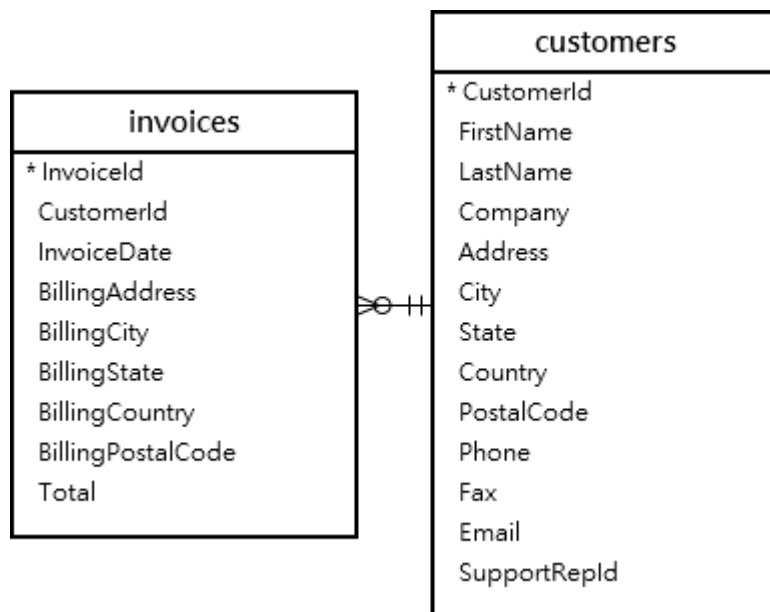
v1
2
3

The following picture illustrates the INTERSECT operation:



SQLite INTERSECT example

For the demonstration, we will use the customers and invoices tables from the [sample database](#).



The following statement finds customers who have invoices:

```
SELECT CustomerId
FROM customers
INTERSECT
SELECT CustomerId
FROM invoices
ORDER BY CustomerId;
```

Code language: SQL (Structured Query Language) (sql)

The following picture shows the partial output:

CustomerId
1
2
3
4
5
6
7
8
9
10

Grouping data

SQLite Group By

Summary: in this tutorial, you will learn how to use SQLite GROUP BY clause to make a set of summary rows from a set of rows.

Introduction to SQLite GROUP BY clause

The GROUP BY clause is an optional clause of the [SELECT](#) statement. The GROUP BY clause a selected group of rows into summary rows by values of one or more columns.

The GROUP BY clause returns one row for each group. For each group, you can apply an aggregate function such as [MIN](#), [MAX](#), [SUM](#), [COUNT](#), or [AVG](#) to provide more information about each group.

The following statement illustrates the syntax of the SQLite GROUP BY clause.

```
SELECT
    column 1,
    aggregate function(column 2)
FROM
    table
GROUP BY
    column 1,
    column 2;
```

Code language: SQL (Structured Query Language) (sql)

The GROUP BY clause comes after the FROM clause of the SELECT statement. In case a statement contains a [WHERE](#) clause, the GROUP BY clause must come after the WHERE clause.

Following the GROUP BY clause is a column or a list of comma-separated columns used to specify the group.

SQLite GROUP BY examples

We use the tracks table from the [sample database](#) for the demonstration.

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

SQLite GROUP BY clause with COUNT function

The following statement returns the album id and the number of tracks per album. It uses the GROUP BY clause to groups tracks by album and applies the [COUNT\(\)](#) function to each group.

```
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	count(TrackId)
1	10
2	1
3	3
4	8
5	15
6	13
7	12
8	14
9	8
10	14
11	12
12	12
13	8
14	13

You can use the [ORDER BY](#) clause to sort the groups as follows:

```
SELECT
    albumid,
    COUNT(trackid)
```



```
FROM
    tracks
GROUP BY
    albumid
ORDER BY COUNT(trackid) DESC;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	COUNT(trackid)
▶ 141	57
23	34
73	30
229	26
230	25
251	25
83	24
231	24
253	24
24	23

SQLite GROUP BY clause with SUM function example

You can use the [SUM](#) function to calculate total per group. For example, to get total length and bytes for each album, you use the SUM function to calculate total milliseconds and bytes.

```
SELECT
    albumid,
    SUM(milliseconds) length,
    SUM(bytes) size
FROM
    tracks
GROUP BY
    albumid;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	length	size
▶ 1	2400415	78270414
2	342562	5510424
3	858088	14613294
4	2453259	80239024
5	4411709	144277453
6	3450925	113150359
7	3249365	105527170
8	2906926	94682869
9	2671407	87412645
10	3927713	94304482
11	3224237	104821979
12	1615722	25479147

SQLite GROUP BY with MAX, MIN, and AVG functions

The following statement returns the album id, album title, maximum length, minimum length, and the average length of tracks in the tracks table.

```
SELECT
    tracks.albumid,
    title,
    min(milliseconds),
    max(milliseconds),
    round(avg(milliseconds),2)
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid
GROUP BY
    tracks.albumid;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	Title	min(milliseconds)	max(milliseconds)	round(avg(milliseconds),2)
1	For Those About To Rock We Salute You	199836	343719	240041.5
2	Balls to the Wall	342562	342562	342562.0
3	Restless and Wild	230619	375418	286029.33
4	Let There Be Rock	215196	369319	306657.38
5	Big Ones	215875	381231	294113.93
6	Jagged Little Pill	176117	491885	265455.77
7	Facelift	152084	387134	270780.42
8	Warner 25 Anos	126511	366837	207637.57
9	Plays Metallica By Four Cellos	221701	436453	333925.88
10	Audioslave	206053	343457	280550.93
11	Out Of Exile	233195	309786	268686.42
12	BackBeat Soundtrack	106266	163265	134643.5
13	The Best Of Billy Cobham	246151	582086	335065.5
14	Alcohol Fueled Brewtality Live! [Disc 1]	235833	555075	312301.46

SQLite GROUP BY multiple columns example

In the previous example, we have used one column in the GROUP BY clause. SQLite allows you to group rows by multiple columns.

For example, to group tracks by media type and genre, you use the following statement:

```
SELECT
    MediaTypeId,
    GenreId,
    COUNT(TrackId)
FROM
    tracks
GROUP BY
    MediaTypeId,
    GenreId;
```

Code language: SQL (Structured Query Language) (sql)

MediaTypeId	GenreId	count(trackid)
1	1	1211
1	2	127
1	3	374
1	4	332
1	5	12
1	6	81
1	7	578
1	8	58
1	9	14
1	10	42
1	11	15
1	12	24
1	13	28
1	14	49

SQLite uses the combination of values of MediaTypeId and GenreId columns as a group e.g., (1,1) and (1,2). It then applies the [COUNT](#) function to return the number of tracks in each group.

SQLite Having

Summary: in this tutorial, you will learn how to use SQLite HAVING clause to specify a filter condition for a group or an aggregate.

Introduction to SQLite HAVING clause

SQLite HAVING clause is an optional clause of the [SELECT](#) statement. The HAVING clause specifies a search condition for a group.

You often use the HAVING clause with the [GROUP BY](#) clause. The GROUP BY clause groups a set of rows into a set of summary rows or groups. Then the HAVING clause filters groups based on a specified condition.

If you use the HAVING clause, you must include the GROUP BY clause; otherwise, you will get the following error:

```
Error: a GROUP BY clause is required before HAVING
Code language: JavaScript (javascript)
```

Note that the HAVING clause is applied after GROUP BY clause, whereas the [WHERE](#) clause is applied before the GROUP BY clause.

The following illustrates the syntax of the HAVING clause:

```

SELECT
    column 1,
    column 2,
    aggregate function (column 3)
FROM
    table
GROUP BY
    column 1,
    column 2
HAVING
    search condition;

```

Code language: SQL (Structured Query Language) (sql)

In this syntax, the HAVING clause evaluates the `search_condition` for each group as a Boolean expression. It only includes a group in the final result set if the evaluation is true.

SQLite HAVING clause examples

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

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

To find the number of tracks for each album, you use GROUP BY clause as follows:

```

SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid;

```

Code language: SQL (Structured Query Language) (sql)

AlbumId	COUNT(trackid)
1	10
2	1
3	3
4	8
5	15
6	13
7	12

To find the numbers of tracks for the album with id 1, we add a HAVING clause to the following statement:

```
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid
HAVING albumid = 1;
```

AlbumId	COUNT(trackid)
1	10

We have referred to the AlbumId column in the HAVING clause.

To find albums that have the number of tracks between 18 and 20, you use the [aggregate function](#) in the HAVING clause as shown in the following statement:

```
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid
HAVING
    COUNT(albumid) BETWEEN 18 AND 20
ORDER BY albumid;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	COUNT(trackid)
21	18
37	20
54	20
55	20
72	18
102	18
115	20

The following statement queries data from `tracks` and `albums` tables using [inner join](#) to find albums that have the total length greater than 60,000,000 milliseconds.

```
SELECT
    tracks.AlbumId,
    title,
    SUM(Milliseconds) AS length
FROM
    tracks
INNER JOIN albums ON albums.AlbumId = tracks.AlbumId
GROUP BY
    tracks.AlbumId
HAVING
    length > 60000000;
```

Code language: SQL (Structured Query Language) (sql)

AlbumId	Title	length
229	Lost, Season 3	70665582
230	Lost, Season 1	64854936
231	Lost, Season 2	63289631
253	Battlestar Galactica (Classic), Season 1	70213784

Sorting rows

SQLite Order By

Summary: in this tutorial, you will learn how to sort a result set of a query using SQLite `ORDER BY` clause.

Introduction to SQLite `ORDER BY` clause

SQLite stores data in the tables in an unspecified order. It means that the rows in the table may or may not be in the order that they were inserted.

If you use the [SELECT](#) statement to query data from a table, the order of rows in the result set is unspecified.

To sort the result set, you add the `ORDER BY` clause to the `SELECT` statement as follows:

```
SELECT
  select list
FROM
  table
ORDER BY
  column 1 ASC,
  column 2 DESC;
```

Code language: SQL (Structured Query Language) (sql)

The `ORDER BY` clause comes after the `FROM` clause. It allows you to sort the result set based on one or more columns in ascending or descending order.

In this syntax, you place the column name by which you want to sort after the `ORDER BY` clause followed by the `ASC` or `DESC` keyword.

- The `ASC` keyword means ascending.
- And the `DESC` keyword means descending.

If you don't specify the `ASC` or `DESC` keyword, SQLite sorts the result set using the `ASC` option. In other words, it sorts the result set in the ascending order by default.

In case you want to sort the result set by multiple columns, you use a comma (,) to separate two columns. The `ORDER BY` clause sorts rows using columns or expressions from left to right. In other words, the `ORDER BY` clause sorts the rows using the first column in the list. Then, it sorts the sorted rows using the second column, and so on.

You can sort the result set using a column that does not appear in the select list of the SELECT clause.

SQLite ORDER BY clause example

Let's take the tracks table in the [sample database](#) for the demonstration.

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

Suppose, you want to get data from name, milliseconds, and album id columns, you use the following statement:

```
SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks;
```

Code language: SQL (Structured Query Language) (sql)

Name	Milliseconds	AlbumId
▶ For Those About To Rock (We Salute You)	343719	1
Balls to the Wall	342562	2
Fast As a Shark	230619	3
Restless and Wild	252051	3
Princess of the Dawn	375418	3
Put The Finger On You	205662	1
Let's Get It Up	233926	1

The SELECT statement that does not use ORDER BY clause returns a result set that is not in any order.

Suppose you want to sort the result set based on AlbumId column in ascending order, you use the following statement:

```
SELECT
    name,
```



```

        milliseconds,
        albumid
FROM
    tracks
ORDER BY
    albumid ASC;

```

Code language: SQL (Structured Query Language) (sql)

Name	Milliseconds	AlbumId
▶ For Those About To Rock (We Salute You)	343719	1
Put The Finger On You	205662	1
Let's Get It Up	233926	1
Inject The Venom	210834	1
Snowballed	203102	1
Evil Walks	263497	1
C.O.D.	199836	1
Breaking The Rules	263288	1
Night Of The Long Knives	205688	1
Spellbound	270863	1
Balls to the Wall	342562	2
Fast As a Shark	230619	3
Restless and Wild	252051	3
Princess of the Dawn	375418	3
Go Down	331180	4
Dog Eat Dog	215196	4



The result set now is sorted by the AlbumId column in ascending order as shown in the screenshot.

SQLite uses ASC by default so you can omit it in the above statement as follows:

```

SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks
ORDER BY
    albumid;

```

Suppose you want to sort the sorted result (by AlbumId) above by the Milliseconds column in descending order. In this case, you need to add the Milliseconds column to the ORDER BY clause as follows:

```

SELECT
    name,
    milliseconds,
    albumid
FROM

```

```

tracks
ORDER BY
    albumid ASC,
    milliseconds DESC;

```

Code language: SQL (Structured Query Language) (sql)

Name	Milliseconds	AlbumId
▶ For Those About To Rock (We Salute You)	343719	1
Spellbound	270863	1
Evil Walks	263497	1
Breaking The Rules	263288	1
Let's Get It Up	233926	1
Inject The Venom	210834	1
Night Of The Long Knives	205688	1
Put The Finger On You	205662	1
Snowballed	203102	1
C.O.D.	199836	1
Balls to the Wall	342562	2
Princess of the Dawn	375418	3
Restless and Wild	252051	3
Fast As a Shark	230619	3
Overdose	369319	4
Let There Be Rock	366654	4

SQLite sorts rows by AlbumId column in ascending order first. Then, it sorts the sorted result set by the Milliseconds column in descending order.

If you look at the tracks of the album with AlbumId 1, you find that the order of tracks changes between the two statements.

SQLite ORDER BY with the column position

Instead of specifying the names of columns, you can use the column's position in the ORDER BY clause.

For example, the following statement sorts the tracks by both albumid (3rd column) and milliseconds (2nd column) in ascending order.

```


SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks
ORDER BY
    3, 2;

```

Code language: SQL (Structured Query Language) (sql)

The number 3 and 2 refers to the AlbumId and Milliseconds in the column list that appears in the SELECT clause.

Name	Milliseconds	AlbumId
► C.O.D.	199836	1
Snowballed	203102	1
Put The Finger On You	205662	1
Night Of The Long Knives	205688	1
Inject The Venom	210834	1
Let's Get It Up	233926	1
Breaking The Rules	263288	1
Evil Walks	263497	1
Spellbound	270863	1
For Those About To Rock (We Salute You)	343719	1
Balls to the Wall	342562	2
Fast As a Shark	230619	3
Restless and Wild	252051	3
Princess of the Dawn	375418	3
Dog Eat Dog	215196	4
Hell Ain't A Bad Place To Be	254380	4



Sorting NULLs

In the database world, NULL is special. It denotes that the information missing or the data is not applicable.

Suppose you want to store the birthday of an artist in a table. At the time of saving the artist's record, you don't have the birthday information.

To represent the unknown birthday information in the database, you may use a special date like 01.01.1900 or an " empty string. However, both of these values do not clearly show that the birthday is unknown.

NULL was invented to resolve this issue. Instead of using a special value to indicate that the information is missing, NULL is used.

NULL is special because you cannot compare it with another value. Simply put, if the two pieces of information are unknown, you cannot compare them.

NULL is even cannot be compared with itself; NULL is not equal to itself so NULL = NULL always results in false.

When it comes to sorting, SQLite considers NULL to be smaller than any other value.

It means that NULLs will appear at the beginning of the result set if you use ASC or at the end of the result set when you use DESC.

SQLite 3.30.0 added the NULLS FIRST and NULLS LAST options to the ORDER BY clause. The NULLS FIRST option specifies that the NULLs will appear at the beginning of the result set while the NULLS LAST option place NULLs at the end of the result set.

The following example uses the ORDER BY clause to sort tracks by composers:

```
SELECT
  TrackId,
  Name,
  Composer
FROM
  tracks
ORDER BY
  Composer;
```

Code language: SQL (Structured Query Language) (sql)

First, you see that NULLs appear at the beginning of the result set because SQLite treats them as the lowest values. When you scroll down the result, you will see other values:

TrackId	Name	Composer
3467	Intro / Stronger Than Me	(Null)
3468	You Sent Me Flying / Cherry	(Null)
3470	I Heard Love Is Blind	(Null)
3478	Slowness	(Null)
3481	A Midsummer Night's Dream, Op.61 Incidental	(Null)
3496	Étude 1, In C Major - Preludio (Presto) - Liszt	(Null)
3497	Erlkonig, D.328	(Null)
3499	Pini Di Roma (Pinien Von Rom) \ I Pini Della Via	(Null)
2107	Iron Man	A. F. Iommi, W. Ward, T. Butler, J. Osbourne
2108	Children Of The Grave	A. F. Iommi, W. Ward, T. Butler, J. Osbourne
2109	Paranoid	A. F. Iommi, W. Ward, T. Butler, J. Osbourne
1908	New Rhumba	A. Jamal
415	Astronomy	A.Bouchard/J.Bouchard/S.Pearlman
2589	Hard To Handle	A.Isbell/A.Jones/O.Redding
15	Go Down	AC/DC

The following example uses the NULLS LAST option to place NULLs after other values:

```
SELECT
  TrackId,
  Name,
  Composer
FROM
```

```

tracks
ORDER BY
  Composer NULLS LAST;

```

Code language: SQL (Structured Query Language) (sql)

If you scroll down the output, you will see that NULLs are placed at the end of the result set:

TrackId	Name	Composer
1052	The Lady Is A Tramp	lorenz hart/richard rodgers
1041	For Once In My Life	orlando murden/ronald miller
1055	Loves Been Good To Me	rod mckuen
817	Lick It Up	roger glover
819	Talk About Love	roger glover
820	Time To Kill	roger glover
821	Ramshackle Man	roger glover
822	A Twist In The Tail	roger glover
824	Solitaire	roger glover
825	One Man's Meat	roger glover
2	Balls to the Wall	(Null)
63	Desafinado	(Null)
64	Garota De Ipanema	(Null)
65	Samba De Uma Nota Só (One Note Samba)	(Null)
66	Por Causa De Você	(Null)
67	Ligia	(Null)
68	Fotografia	(Null)
69	Dindi (Dindi)	(Null)
70	Se Todos Fossem Iguais A Você (Instrumental)	(Null)

More querying techniques

SQLite CASE

Summary: in this tutorial, you will learn about the SQLite CASE expression to add the conditional logic to a query.

The SQLite CASE expression evaluates a list of conditions and returns an expression based on the result of the evaluation.

The CASE expression is similar to the IF-THEN-ELSE statement in other programming languages.

You can use the CASE expression in any clause or statement that accepts a valid expression. For example, you can use the CASE expression in clauses such as [WHERE](#), ORDER BY, [HAVING](#), [SELECT](#) and statements such as [SELECT](#), [UPDATE](#), and [DELETE](#).

SQLite provides two forms of the CASE expression: simple CASE and searched CASE.

SQLite simple CASE expression

The simple CASE expression compares an expression to a list of expressions to return the result. The following illustrates the syntax of the simple CASE expression.

```
CASE case_expression
  WHEN when_expression_1 THEN result_1
  WHEN when_expression_2 THEN result_2
  ...
  [ ELSE result_else ]
END
```

Code language: SQL (Structured Query Language) (sql)

The simple CASE expression compares the `case_expression` to the expression appears in the first WHEN clause, `when_expression_1`, for equality.

If the `case_expression` equals `when_expression_1`, the simple CASE returns the expression in the corresponding THEN clause, which is the `result_1`.

Otherwise, the simple CASE expression compares the `case_expression` with the expression in the next WHEN clause.

In case no `case_expression` matches the `when_expression`, the CASE expression returns the `result_else` in the ELSE clause. If you omit the ELSE clause, the CASE expression returns NULL.

The simple CASE expression uses short-circuit evaluation. In other words, it returns the result and stop evaluating other conditions as soon as it finds a match.

Simple CASE example

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

customers
* CustomerId
FirstName
LastName
Company
Address
City
State
Country
PostalCode
Phone
Fax
Email
SupportRepId

Suppose, you have to make a report of the customer groups with the logic that if a customer locates in the USA, this customer belongs to the domestic group, otherwise the customer belongs to the foreign group.

To make this report, you use the simple CASE expression in the SELECT statement as follows:

```
SELECT customerid,  
       firstname,  
       lastname, country,  
       CASE country  
         WHEN 'USA'  
           THEN 'Domestic'  
         ELSE 'Foreign'  
       END CustomerGroup  
FROM  
  customers  
ORDER BY  
  LastName,  
  FirstName;
```

Code language: SQL (Structured Query Language) (sql)

SQLite searched CASE expression

The searched CASE expression evaluates a list of expressions to decide the result. Note that the simple CASE expression only compares for equality, while the searched CASE expression can use any forms of comparison.

The following illustrates the syntax of the searched CASE expression.

```
CASE
  WHEN bool expression 1 THEN result 1
  WHEN bool expression 2 THEN result 2
  [ ELSE result else ]
END
```

Code language: SQL (Structured Query Language) (sql)

The searched CASE expression evaluates the Boolean expressions in the sequence specified and return the corresponding result if the expression evaluates to true.

In case no expression evaluates to true, the searched CASE expression returns the expression in the ELSE clause if specified. If you omit the ELSE clause, the searched CASE expression returns NULL.

Similar to the simple CASE expression, the searched CASE expression stops the evaluation when a condition is met.

Searched CASE example

We will use the tracks table for the demonstration.

tracks
* TrackId
Name
AlbumId
MediaTypeId
GenreId
Composer
Milliseconds
Bytes
UnitPrice

Suppose you want to classify the tracks based on its length such as less a minute, the track is short; between 1 and 5 minutes, the track is medium; greater than 5 minutes, the track is long.

To achieve this, you use the searched CASE expression as follows:

```
SELECT
```



```

trackid,
name,milliseconds,
CASE
    WHEN milliseconds < 60000 THEN
        'short'
    WHEN milliseconds > 60000 AND milliseconds <
300000 THEN 'medium'
    ELSE
        'long'
    END category
FROM
    Tracks;

```

Code language: SQL (Structured Query Language) (sql)

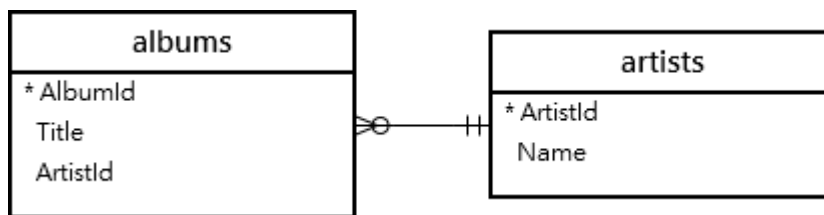
TrackId	Name	category
1	For Those About To Rock (We Salute You)	long
2	Balls to the Wall	long
3	Fast As a Shark	medium
4	Restless and Wild	medium
5	Princess of the Dawn	long
6	Put The Finger On You	medium
7	Let's Get It Up	medium
8	Inject The Venom	medium
9	Snowballed	medium
10	Evil Walks	medium
11	C.O.D.	medium
12	Breaking The Rules	medium
13	Night Of The Long Knives	medium
14	Spellbound	medium

Joining tables

SQLite Join

Summary: in this tutorial, you will learn about various kinds of SQLite joins to query data from two or more tables.

For the demonstration, we will use the artists and albums tables from the [sample database](#).



An artist can have zero or many albums while an album belongs to one artist.

To query data from both artists and albums tables, you can use an [INNER JOIN](#), [LEFT JOIN](#), or [CROSS JOIN](#) clause. Each join clause determines how SQLite uses data from one table to match with rows in another table.

Note that SQLite doesn't directly support the [RIGHT JOIN](#) and [FULL OUTER JOIN](#).

SQLite INNER JOIN

The following statement returns the album titles and their artist names:

```
SELECT
    Title,
    Name
FROM
    albums
INNER JOIN artists
    ON artists.ArtistId = albums.ArtistId limit 5;
```

Code language: SQL (Structured Query Language) (sql)

Here is the partial output:

Title	Name
For Those About To Rock We Salute You	AC/DC
Balls to the Wall	Accept
Restless and Wild	Accept
Let There Be Rock	AC/DC
Big Ones	Aerosmith
Jagged Little Pill	Alanis Morissette
Facelift	Alice In Chains
Warner 25 Anos	Antônio Carlos Jobim
Plays Metallica By Four Cellos	Apocalyptica
Audioslave	Audioslave
Out Of Exile	Audioslave
BackBeat Soundtrack	BackBeat
The Best Of Billy Cobham	Billy Cobham
Alcohol Fueled Brewtality Live! [Disc 1]	Black Label Society
Alcohol Fueled Brewtality Live! [Disc 2]	Black Label Society
Black Sabbath	Black Sabbath
Black Sabbath Vol. 4 (Remaster)	Black Sabbath

In this example, the `INNER JOIN` clause matches each row from the `albums` table with every row from the `artists` table based on the join condition (`artists.ArtistId = albums.ArtistId`) specified after the `ON` keyword.

If the join condition evaluates to true (or 1), the columns of rows from both `albums` and `artists` tables are included in the result set.

This query uses table aliases (`l` for the `albums` table and `r` for `artists` table) to shorten the query:

```
SELECT
    l.Title,
    r.Name
FROM
    albums l
INNER JOIN artists r ON
    r.ArtistId = l.ArtistId limit 5;
```

Code language: SQL (Structured Query Language) (sql)

In case the column names of joined tables are the same e.g., `ArtistId`, you can use the `USING` syntax as follows:

```
SELECT
    Title,
    Name
FROM
    albums
INNER JOIN artists USING (ArtistId);
```

Code language: SQL (Structured Query Language) (sql)

The clause `USING(ArtistId)` is equivalent to the clause `ON artists.ArtistId = albums.ArtistId`.

SQLite LEFT JOIN

This statement selects the artist names and album titles from the artists and albums tables using the `LEFT JOIN` clause:

```
SELECT
  Name,
  Title
FROM
  artists
LEFT JOIN albums ON
  artists.ArtistId = albums.ArtistId
ORDER BY Name limit 5;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

Name	Title
A Cor Do Som	[NULL]
AC/DC	For Those About To Rock We Salute You
AC/DC	Let There Be Rock
Aaron Copland & London Symphony Orchestra	A Copland Celebration, Vol. I
Aaron Goldberg	Worlds
Academy of St. Martin in the Fields & Sir Neville Marriner	The World of Classical Favourites
Academy of St. Martin in the Fields Chamber Ensemble & Sir Neville Marriner	Sir Neville Marriner: A Celebration
Academy of St. Martin in the Fields, John Birch, Sir Neville Marriner	Fauré: Requiem, Ravel: Pavane & Others
Academy of St. Martin in the Fields, Sir Neville Marriner & The English Chamber Orchestra	Bach: Orchestral Suites Nos. 1 - 4
Academy of St. Martin in the Fields, Sir Neville Marriner & The English Chamber Orchestra	[NULL]
Accept	Balls to the Wall
Accept	Restless and Wild
Adrian Leaper & Doreen de Feis	Górecki: Symphony No. 3
Aerosmith	Big Ones
Aerosmith & Sierra Leone's Refugee Allstars	[NULL]
Aisha Duo	Quiet Songs
Alanis Morissette	Jagged Little Pill

The `LEFT JOIN` clause selects data starting from the left table (artists) and matching rows in the right table (albums) based on the join condition (`artists.ArtistId = albums.ArtistId`).

The left join returns all rows from the artists table (or left table) and the matching rows from the albums table (or right table).

If a row from the left table doesn't have a matching row in the right table, SQLite includes columns of the rows in the left table and `NULL` for the columns of the right table.

Similar to the `INNER JOIN` clause, you can use the `USING` syntax for the join condition as follows:

```
SELECT
  Name,
  Title
FROM
  artists
LEFT JOIN albums USING (ArtistId)
ORDER BY
  Name;
```

Code language: SQL (Structured Query Language) (sql)

If you want to find artists who don't have any albums, you can add a [WHERE](#) clause as shown in the following query:

```
SELECT
  Name,
  Title
FROM
  artists
LEFT JOIN albums ON
  artists.ArtistId = albums.ArtistId
WHERE Title IS NULL
ORDER BY Name;
```

Code language: SQL (Structured Query Language) (sql)

This picture shows the partial output:

Name 	Title 
A Cor Do Som	[NULL]
Academy of St. Martin in the Fields, Sir Neville Marriner & William Bennett	[NULL]
Aerosmith & Sierra Leone's Refugee Allstars	[NULL]
Avril Lavigne	[NULL]
Azymuth	[NULL]
Baby Consuelo	[NULL]
Banda Black Rio	[NULL]
Barão Vermelho	[NULL]
Bebel Gilberto	[NULL]
Ben Harper	[NULL]
Big & Rich	[NULL]
Black Eyed Peas	[NULL]
Charlie Brown Jr.	[NULL]
Christina Aguilera featuring BigElf	[NULL]
Corinne Bailey Rae	[NULL]
DJ Dolores & Orchestra Santa Massa	[NULL]

Generally, this type of query allows you to find rows that are available in the left table but don't have corresponding rows in the right table.

Note that LEFT JOIN and LEFT OUTER JOIN are synonyms.

SQLite CROSS JOIN

The CROSS JOIN clause creates a [Cartesian product](#) of rows from the joined tables.

Unlike the INNER JOIN and LEFT JOIN clauses, a CROSS JOIN doesn't have a join condition. Here is the basic syntax of the CROSS JOIN clause:

```
SELECT
    select list
FROM table1
CROSS JOIN table2;
```

Code language: SQL (Structured Query Language) (sql)

The CROSS JOIN combines every row from the first table (table1) with every row from the second table (table2) to form the result set.

If the first table has N rows, the second table has M rows, the final result will have NxM rows.

A practical example of the CROSS JOIN clause is to combine two sets of data for forming an initial data set for further processing. For example, you have a list of products and months, and you want to make a plan when you can sell which products.

The following script creates the products and calendars tables:

```
CREATE TABLE products(  
  product text NOT null  
);  
  
INSERT INTO products(product)  
VALUES ('P1'), ('P2'), ('P3');
```

```
CREATE TABLE calendars(  
  y int NOT NULL,  
  m int NOT NULL  
);  
  
INSERT INTO calendars(y,m)  
VALUES  
  (2019,1),  
  (2019,2),  
  (2019,3),  
  (2019,4),  
  (2019,5),  
  (2019,6),  
  (2019,7),  
  (2019,8),  
  (2019,9),  
  (2019,10),  
  (2019,11),  
  (2019,12);
```




Code language: SQL (Structured Query Language) (sql)

This query uses the CROSS JOIN clause to combine the products with the months:

```
SELECT *  
FROM products  
CROSS JOIN calendars;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

product 	y 	m 
A	2,019	1
A	2,019	2
A	2,019	3
A	2,019	4
A	2,019	5
A	2,019	6
A	2,019	7
A	2,019	8
A	2,019	9
A	2,019	10
A	2,019	11
A	2,019	12
B	2,019	1
B	2,019	2
B	2,019	3
B	2,019	4
B	2,019	5
B	2,019	6
B	2,019	7
B	2,019	8
B	2,019	9
B	2,019	10
B	2,019	11
B	2,019	12
C	2,019	1
C	2,019	2
C	2,019	3
C	2,019	4
C	2,019	5
C	2,019	6
C	2,019	7
C	2,019	8
C	2,019	9
C	2,019	10
C	2,019	11
C	2,019	12

SQLite Inner Join

Summary: this tutorial shows you how to use SQLite inner join clause to query data from multiple tables.

Introduction to SQLite inner join clause

In relational databases, data is often distributed in many related tables. A table is associated with another table using [foreign keys](#).

To [query data](#) from multiple tables, you use INNER JOIN clause. The INNER JOIN clause combines columns from correlated tables.

Suppose you have two tables: A and B.

A has a1, a2, and f columns. B has b1, b2, and f column. The A table links to the B table using a foreign key column named f.

The following illustrates the syntax of the inner join clause:

```
SELECT a1, a2, b1, b2
FROM A
INNER JOIN B on B.f = A.f;
```

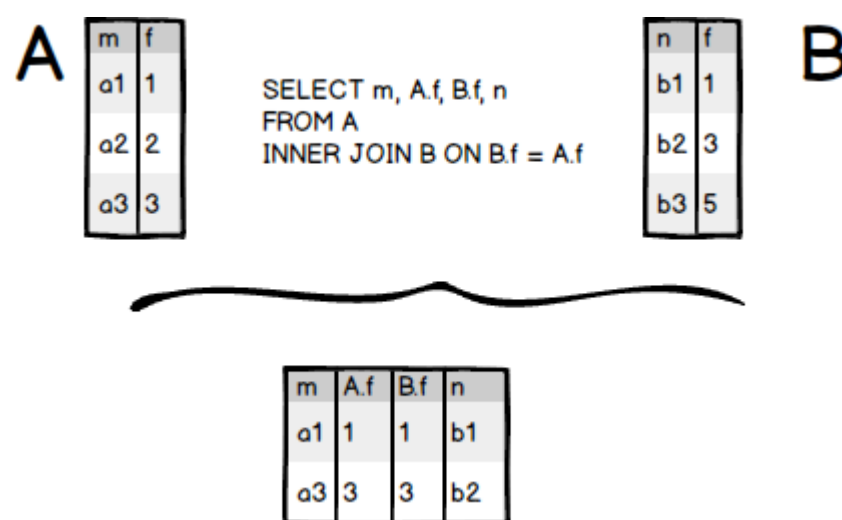
Code language: SQL (Structured Query Language) (sql)

For each row in the A table, the INNER JOIN clause compares the value of the f column with the value of the f column in the B table. If the value of the f column in the A table equals the value of the f column in the B table, it combines data from a1, a2, b1, b2, columns and includes this row in the result set.

In other words, the INNER JOIN clause returns rows from the A table that has the corresponding row in B table.

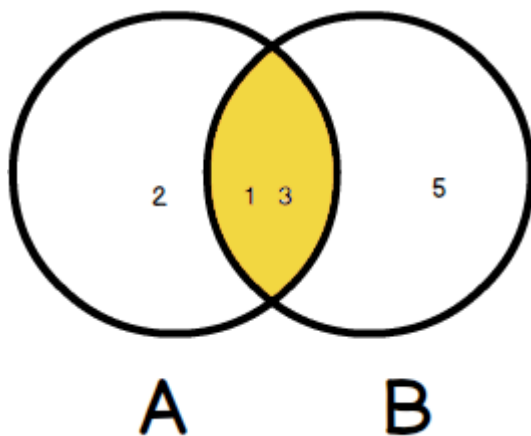
This logic is applied if you join more than 2 tables.

See the following example.



Only the rows in the A table: (a1,1), (a3,3) have the corresponding rows in the B table (b1,1), (b2,3) are included in the result set.

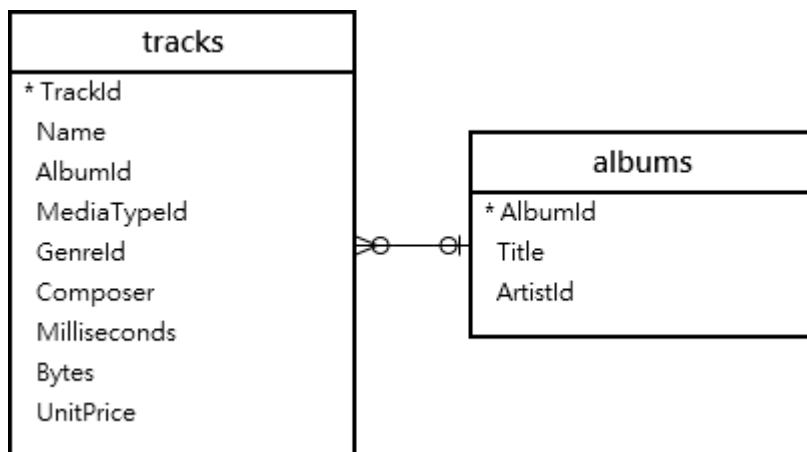
The following diagram illustrates the INNER JOIN clause:



SQLite INNER JOIN examples

Let's take a look at the tracks and albums tables in the [sample database](#).

The tracks table links to the albums table via AlbumId column.



In the tracks table, the AlbumId column is a foreign key. And in the albums table, the AlbumId is the [primary key](#).

To query data from both tracks and albums tables, you use the following statement:

```
SELECT
    trackid,
    name,
    title
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid;
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name	Title
1	For Those About To Rock	For Those About To Rock We Salute You
6	Put The Finger On You	For Those About To Rock We Salute You
7	Let's Get It Up	For Those About To Rock We Salute You
8	Inject The Venom	For Those About To Rock We Salute You
9	Snowballed	For Those About To Rock We Salute You
10	Evil Walks	For Those About To Rock We Salute You
11	C.O.D.	For Those About To Rock We Salute You
12	Breaking The Rules	For Those About To Rock We Salute You
13	Night Of The Long Knives	For Those About To Rock We Salute You
14	Spellbound	For Those About To Rock We Salute You
2	Balls to the Wall	Balls to the Wall
3	Fast As a Shark	Restless and Wild
4	Restless and Wild	Restless and Wild
5	Princess of the Dawn	Restless and Wild

For each row in the tracks table, SQLite uses the value in the albumid column of the tracks table to compare with the value in the albumid of the albums table. If SQLite finds a match, it combines data of rows in both tables in the result set.

You can include the AlbumId columns from both tables in the final result set to see the effect.

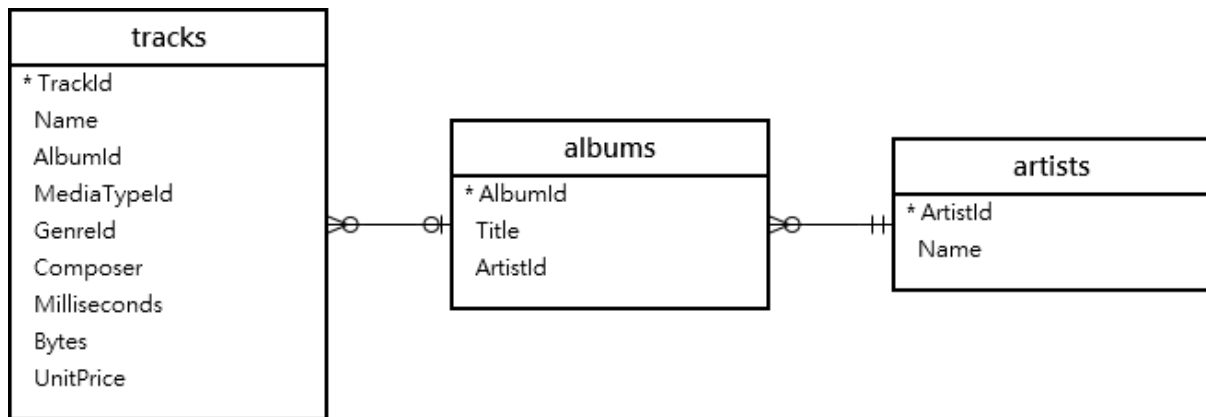
```
SELECT
    trackid,
    name,
    tracks.albumid AS album id tracks,
    albums.albumid AS album id albums,
    title
FROM
    tracks
    INNER JOIN albums ON albums.albumid = tracks.albumid;
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Name	album_id_tracks	album_id_albums	Title
1	For Those About To Rock	1	1	For Those About To Rock We Salute You
6	Put The Finger On You	1	1	For Those About To Rock We Salute You
7	Let's Get It Up	1	1	For Those About To Rock We Salute You
8	Inject The Venom	1	1	For Those About To Rock We Salute You
9	Snowballed	1	1	For Those About To Rock We Salute You
10	Evil Walks	1	1	For Those About To Rock We Salute You
11	C.O.D.	1	1	For Those About To Rock We Salute You
12	Breaking The Rules	1	1	For Those About To Rock We Salute You
13	Night Of The Long Knives	1	1	For Those About To Rock We Salute You
14	Spellbound	1	1	For Those About To Rock We Salute You
2	Balls to the Wall	2	2	Balls to the Wall
3	Fast As a Shark	3	3	Restless and Wild
4	Restless and Wild	3	3	Restless and Wild
5	Princess of the Dawn	3	3	Restless and Wild

SQLite inner join – 3 tables example

See the following tables: tracks albums and artists



One track belongs to one album and one album have many tracks. The tracks table associated with the albums table via albumid column.

One album belongs to one artist and one artist has one or many albums. The albums table links to the artists table via artistid column.

To query data from these tables, you need to use two inner join clauses in the [SELECT](#) statement as follows:

```
SELECT
    trackid,
    tracks.name AS track,
    albums.title AS album,
    artists.name AS artist
FROM
    tracks
    INNER JOIN albums ON albums.albumid = tracks.albumid
    INNER JOIN artists ON artists.artistid = albums.artistid
limit 5;
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Track	Album	Artist
1	For Those About To Rock	For Those About To Rock We Salute You	AC/DC
6	Put The Finger On You	For Those About To Rock We Salute You	AC/DC
7	Let's Get It Up	For Those About To Rock We Salute You	AC/DC
8	Inject The Venom	For Those About To Rock We Salute You	AC/DC
9	Snowballed	For Those About To Rock We Salute You	AC/DC
10	Evil Walks	For Those About To Rock We Salute You	AC/DC
11	C.O.D.	For Those About To Rock We Salute You	AC/DC
12	Breaking The Rules	For Those About To Rock We Salute You	AC/DC
13	Night Of The Long Knives	For Those About To Rock We Salute You	AC/DC
14	Spellbound	For Those About To Rock We Salute You	AC/DC
15	Go Down	Let There Be Rock	AC/DC
16	Dog Eat Dog	Let There Be Rock	AC/DC
17	Let There Be Rock	Let There Be Rock	AC/DC
18	Bad Boy Boogie	Let There Be Rock	AC/DC

You can use a [WHERE clause](#) to get the tracks and albums of the artist with id 10 as the following statement:

```
SELECT
    trackid,
    tracks.name AS Track,
    albums.title AS Album,
    artists.name AS Artist
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid
INNER JOIN artists ON artists.artistid = albums.artistid
WHERE
    artists.artistid = 10;
```

Code language: SQL (Structured Query Language) (sql)

TrackId	Track	Album	Artist
123	Quadrant	The Best Of Billy Cobham	Billy Cobham
124	Snoopy's search-Red baro	The Best Of Billy Cobham	Billy Cobham
125	Spanish moss-"A sound p	The Best Of Billy Cobham	Billy Cobham
126	Moon germs	The Best Of Billy Cobham	Billy Cobham
127	Stratus	The Best Of Billy Cobham	Billy Cobham
128	The pleasant pheasant	The Best Of Billy Cobham	Billy Cobham
129	Solo-Panhandler	The Best Of Billy Cobham	Billy Cobham
130	Do what cha wanna	The Best Of Billy Cobham	Billy Cobham

SQLite CROSS JOIN with a Practical Example

Summary: in this tutorial, you will learn how to use SQLite CROSS JOIN to combine two or more result sets from multiple tables.

Introduction to SQLite CROSS JOIN clause

If you use a [LEFT JOIN](#), [INNER JOIN](#), or CROSS JOIN without the ON or USING clause, SQLite produces the [Cartesian product](#) of the involved tables. The number of rows in the Cartesian product is the product of the number of rows in each involved tables.

Suppose, we have two tables A and B. The following statements perform the cross join and produce a cartesian product of the rows from the A and B tables.

```
SELECT *  
FROM A JOIN B;
```

Code language: SQL (Structured Query Language) (sql)

```
SELECT *  
FROM A  
INNER JOIN B;
```

Code language: SQL (Structured Query Language) (sql)

```
SELECT *  
FROM A  
CROSS JOIN B;
```

Code language: SQL (Structured Query Language) (sql)

```
SELECT *  
FROM A, B;
```

Code language: SQL (Structured Query Language) (sql)

Suppose, the A table has N rows and B table has M rows, the CROSS JOIN of these two tables will produce a result set that contains $N \times M$ rows.

Imagine that if you have the third table C with K rows, the result of the CROSS JOIN clause of these three tables will contain $N \times M \times K$ rows, which may be very huge. Therefore, you should be very careful when using the CROSS JOIN clause.

You use the INNER JOIN and LEFT JOIN clauses more often than the CROSS JOIN clause. However, you will find the CROSS JOIN clause very useful in some cases.

For example, when you want to have a matrix that has two dimensions filled with data completely like members and dates data in a membership database. You want to check the attendants of members for all relevant dates. In this case, you may use the CROSS JOIN clause as the following statement:

```
SELECT name,  
       date  
FROM members  
CROSS JOIN dates;
```

Code language: SQL (Structured Query Language) (sql)

SQLite CROSS JOIN clause example

The following statements create the `ranks` and `suits` tables that store the ranks and suits for a deck of cards and insert the complete data into these two tables.

```
CREATE TABLE ranks (  
  rank TEXT NOT NULL  
);
```

```
CREATE TABLE suits (  
  suit TEXT NOT NULL  
);
```

```
INSERT INTO ranks(rank)  
VALUES ('2'), ('3'), ('4'), ('5'), ('6'), ('7'), ('8'), ('9'), ('10'), (  
'J'), ('Q'), ('K'), ('A');
```

```
INSERT INTO suits(suit)  
VALUES ('Clubs'), ('Diamonds'), ('Hearts'), ('Spades');
```

Code language: SQL (Structured Query Language) (sql)

The following statement uses the `CROSS JOIN` clause to return a complete deck of cards data:

```
SELECT rank,  
  suit  
FROM ranks  
  CROSS JOIN  
  suits  
ORDER BY suit;
```

Code language: SQL (Structured Query Language) (sql)

rank	suit
2	Clubs
3	Clubs
4	Clubs
5	Clubs
6	Clubs
7	Clubs

rank	suit
8	Clubs
9	Clubs
10	Clubs
J	Clubs
Q	Clubs
K	Clubs
A	Clubs
2	Diamonds
3	Diamonds
4	Diamonds
5	Diamonds
6	Diamonds
7	Diamonds
8	Diamonds
9	Diamonds
10	Diamonds
J	Diamonds
Q	Diamonds
K	Diamonds

rank	suit
A	Diamonds
2	Hearts
3	Hearts
4	Hearts
5	Hearts
6	Hearts
7	Hearts
8	Hearts
9	Hearts
10	Hearts
J	Hearts
Q	Hearts
K	Hearts
A	Hearts
2	Spades
3	Spades
4	Spades
5	Spades
6	Spades

rank	suit
7	Spades
8	Spades
9	Spades
10	Spades
J	Spades
Q	Spades
K	Spades
A	Spades

SQLite Self-Join

Summary: in this tutorial, you will learn about a special type of join called SQLite self-join that allows you to join table to itself.

Note that you should be familiar with [INNER JOIN](#) and [LEFT JOIN](#) clauses before going forward with this tutorial.

Introduction to SQLite self-join

The self-join is a special kind of joins that allow you to join a table to itself using either `LEFT JOIN` or `INNER JOIN` clause. You use self-join to create a result set that joins the rows with the other rows within the same table.

Because you cannot refer to the same table more than one in a query, you need to use a table alias to assign the table a different name when you use self-join.

The self-join compares values of the same or different columns in the same table. Only one table is involved in the self-join.

You often use self-join to query parents/child relationship stored in a table or to obtain running totals.

SQLite self-join examples

We will use the employees table in the [sample database](#) for demonstration.

employees	
* EmployeeId	
LastName	
FirstName	
Title	
ReportsTo	
BirthDate	
HireDate	
Address	
City	
State	
Country	
PostalCode	
Phone	
Fax	
Email	

The employees table stores not only employee data but also organizational data. The ReportsTo column specifies the reporting relationship between employees.

If an employee reports to a manager, the value of the ReportsTo column of the employee's row is equal to the value of the EmployeeId column of the manager's row. In case an employee does not report to anyone, the ReportsTo column is NULL.

To get the information on who is the direct report of whom, you use the following statement:

```
SELECT m.firstname || ' ' || m.lastname AS 'Manager',  
       e.firstname || ' ' || e.lastname AS 'Direct report'  
FROM employees e  
INNER JOIN employees m ON m.employeeid = e.reportsto  
ORDER BY manager;
```

Code language: SQL (Structured Query Language) (sql)

Manager	Direct report
Andrew Adams	Nancy Edwards
Andrew Adams	Michael Mitchell
Michael Mitchell	Robert King
Michael Mitchell	Laura Callahan
Nancy Edwards	Jane Peacock
Nancy Edwards	Margaret Park
Nancy Edwards	Steve Johnson

The statement used the `INNER JOIN` clause to join the `employees` to itself.
The `employees` table has two roles: employees and managers.

Because we used the `INNER JOIN` clause to join the `employees` table to itself, the result set does not have the row whose manager column contains a `NULL` value.

Note that the [concatenation operator](#) `||` concatenates multiple strings into a single string. In the example, we use the concatenation operator to form the full names of the employees by concatenating the first name, space, and last name.

In case you want to query the CEO who does not report to anyone, you need to change the `INNER JOIN` clause to `LEFT JOIN` clause in the query above.

Manager	Direct report
(Null)	Andrew Adams
Andrew Adams	Nancy Edwards
Andrew Adams	Michael Mitchell
Michael Mitchell	Robert King
Michael Mitchell	Laura Callahan
Nancy Edwards	Jane Peacock
Nancy Edwards	Margaret Park
Nancy Edwards	Steve Johnson

Andrew Adams is the CEO because he does not report anyone.

You can use the self-join technique to find the employees located in the same city as the following query:

```
SELECT DISTINCT
    e1.city,
    e1.firstName || ' ' || e1.lastname AS fullname
FROM
    employees e1
INNER JOIN employees e2 ON e2.city = e1.city
    AND (e1.firstname <> e2.firstname AND e1.lastname <>
e2.lastname)
ORDER BY
    e1.city;
```

Code language: SQL (Structured Query Language) (sql)

Try It

City	Fullname
Calgary	Jane Peacock
Calgary	Margaret Park
Calgary	Michael Mitchell
Calgary	Nancy Edwards
Calgary	Steve Johnson
Lethbridge	Laura Callahan
Lethbridge	Robert King

The join condition has two expressions:

- `e1.city = e2.city` to make sure that both employees located in the same city
- `e1.firstname <> e2.firstname AND e1.lastname <> e2.lastname` to ensure that `e1` and `e2` are not the same employee with the assumption that there aren't employees who have the same first name and last name.

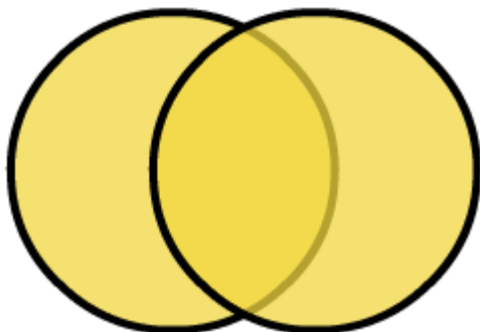
SQLite FULL OUTER JOIN Emulation

Summary: in this tutorial, you will learn how to emulate SQLite full outer join using the UNION and LEFT JOIN clauses.

Introduction to SQL FULL OUTER JOIN clause

In theory, the result of the FULL OUTER JOIN is a combination of a [LEFT JOIN](#) and a RIGHT JOIN. The result set of the full outer join has NULL values for every column of the table that does not have a matching row in the other table. For the matching rows, the FULL OUTER JOIN produces a single row with values from columns of the rows in both tables.

The following picture illustrates the result of the FULL OUTER JOIN clause:



See the following cats and dogs tables.

```
-- create and insert data into the dogs table
CREATE TABLE dogs (
```

```
type TEXT,  
color TEXT  
);
```

```
INSERT INTO dogs(type, color)  
VALUES('Hunting', 'Black'), ('Guard', 'Brown');
```

```
-- create and insert data into the cats table  
CREATE TABLE cats (  
type TEXT,  
color TEXT  
);
```

```
INSERT INTO cats(type, color)  
VALUES('Indoor', 'White'),  
('Outdoor', 'Black');
```

Code language: SQL (Structured Query Language) (sql)

The following statement uses the FULL OUTER JOIN clause to query data from the dogs and cats tables.

```
SELECT *  
FROM dogs  
FULL OUTER JOIN cats  
ON dogs.color = cats.color;
```

Code language: SQL (Structured Query Language) (sql)

The following shows the result of the statement above:

Type	Color	Type	Color
Hunting	Black	Outdoor	Black
Guard	Brown	NULL	NULL
NULL	NULL	Indoor	White

Unfortunately, SQLite does not support the RIGHT JOIN clause and also the FULL OUTER JOIN clause. However, you can easily emulate the FULL OUTER JOIN by using the LEFT JOIN clause.

Emulating SQLite full outer join

The following statement emulates the FULL OUTER JOIN clause in SQLite:

```
SELECT d.type,
```

```

        d.color,
        c.type,
        c.color
FROM dogs d
LEFT JOIN cats c USING(color)
UNION ALL
SELECT d.type,
        d.color,
        c.type,
        c.color
FROM cats c
LEFT JOIN dogs d USING(color)
WHERE d.color IS NULL;

```

Code language: SQL (Structured Query Language) (sql)

How the query works.

- Because SQLite does not support the RIGHT JOIN clause, we use the [LEFT JOIN](#) clause in the second [SELECT](#) statement instead and switch the positions of the cats and dogs tables.
- The [UNION ALL](#) clause retains the duplicate rows from the result sets of both queries.
- The WHERE clause in the second SELECT statement removes rows that already included in the result set of the first SELECT statement.

In this tutorial, you have learned how to use the UNION ALL and LEFT JOIN clauses to emulate the SQLite FULL OUTER JOIN clause.

SQLite Trigger

Summary: this tutorial discusses SQLite trigger, which is a database object fired automatically when the data in a table is changed.

What is an SQLite trigger

An SQLite trigger is a named database object that is executed automatically when an [INSERT](#), [UPDATE](#) or [DELETE](#) statement is issued against the associated table.

When do we need SQLite triggers

You often use triggers to enable sophisticated auditing. For example, you want to log the changes in the sensitive data such as salary and address whenever it changes.

In addition, you use triggers to enforce complex business rules centrally at the database level and prevent invalid transactions.

SQLite `CREATE TRIGGER` statement

To create a new trigger in SQLite, you use the `CREATE TRIGGER` statement as follows:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
  [BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
  ON table_name
  [WHEN condition]
BEGIN
  statements;
END;
```

Code language: SQL (Structured Query Language) (sql)

In this syntax:

- First, specify the name of the trigger after the `CREATE TRIGGER` keywords.
- Next, determine when the trigger is fired such as `BEFORE`, `AFTER`, or [INSTEAD OF](#). You can create `BEFORE` and `AFTER` triggers on a table. However, you can only create an [INSTEAD OF](#) trigger on a view.
- Then, specify the event that causes the trigger to be invoked such as `INSERT`, `UPDATE`, or `DELETE`.
- After that, indicate the table to which the trigger belongs.
- Finally, place the trigger logic in the `BEGIN END` block, which can be any valid SQL statements.

If you combine the time when the trigger is fired and the event that causes the trigger to be fired, you have a total of 9 possibilities:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE
- INSTEAD OF INSERT
- INSTEAD OF DELETE
- INSTEAD OF UPDATE

Suppose you use a `UPDATE` statement to update 10 rows in a table, the trigger that associated with the table is fired 10 times. This trigger is called `FOR EACH ROW` trigger. If the trigger associated with the table is fired one time, we call this trigger a `FOR EACH STATEMENT` trigger.

As of version 3.9.2, SQLite only supports `FOR EACH ROW` triggers. It has not yet supported the `FOR EACH STATEMENT` triggers.

If you use a condition in the `WHEN` clause, the trigger is only invoked when the condition is true. In case you omit the `WHEN` clause, the trigger is executed for all rows.

Notice that if you [drop a table](#), all associated triggers are also deleted. However, if the trigger references other tables, the trigger is not removed or changed if other tables are removed or updated.

For example, a trigger references to a table named `people`, you drop the `people` table or rename it, you need to manually change the definition of the trigger.

You can access the data of the row being inserted, deleted, or updated using the `OLD` and `NEW` references in the form: `OLD.column_name` and `NEW.column_name`.

the `OLD` and `NEW` references are available depending on the event that causes the trigger to be fired.

The following table illustrates the rules.:

Action	Reference
INSERT	NEW is available
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

SQLite triggers examples

Let's [create a new table](#) called leads to store all business leads of the company.

```
CREATE TABLE leads (  
    id integer PRIMARY KEY,  
    first_name text NOT NULL,  
    last_name text NOT NULL,  
    phone text NOT NULL,  
    email text NOT NULL,  
    source text NOT NULL  
);
```

Code language: SQL (Structured Query Language) (sql)

1) SQLite BEFORE INSERT trigger example

Suppose you want to validate the email address before inserting a new lead into the leads table. In this case, you can use a BEFORE INSERT trigger.

First, create a BEFORE INSERT trigger as follows:

```
CREATE TRIGGER validate_email_before_insert_leads  
    BEFORE INSERT ON leads  
BEGIN  
    SELECT  
        CASE  
            WHEN NEW.email NOT LIKE '% @ %. %' THEN  
                RAISE (ABORT, 'Invalid email address')  
            END;  
END;
```

Code language: SQL (Structured Query Language) (sql)

We used the NEW reference to access the email column of the row that is being inserted.

To validate the email, we used the [LIKE](#) operator to determine whether the email is valid or not based on the email pattern. If the email is not valid, the RAISE function aborts the insert and issues an error message.

Second, insert a row with an invalid email into the leads table.

```
INSERT INTO leads (first_name, last_name, email, phone)  
VALUES ('John', 'Doe', 'jjj', '4089009334');
```

Code language: SQL (Structured Query Language) (sql)

SQLite issued an error: "Invalid email address" and aborted the execution of the insert.

Third, insert a row with a valid email.

```
INSERT INTO leads (first name, last name, email, phone,
source)
VALUES ('John', 'Doe', 'john.doe@sqlitetutorial.net',
'4089009334','data');
```

Code language: SQL (Structured Query Language) (sql)

Because the email is valid, the insert statement executed successfully.

```
SELECT
    first name,
    last name,
    email,
    phone
FROM
    leads;
```

Code language: SQL (Structured Query Language) (sql)

first_name	last_name	email	phone
John	Doe	john.doe@sqlitetutorial.net	4089009334

2) SQLite AFTER UPDATE trigger example

The phones and emails of the leads are so important that you can't afford to lose this information. For example, someone accidentally updates the email or phone to the wrong ones or even delete it.

To protect this valuable data, you use a trigger to log all changes which are made to the phone and email.

First, [create a new table](#) called lead_logs to store the historical data.

```
CREATE TABLE lead_logs (
    id INTEGER PRIMARY KEY,
    old_id int,
    new_id int,
    old_phone text,
    new_phone text,
    old_email text,
    new_email text,
    user_action text,
    created_at text
);
```

Code language: SQL (Structured Query Language) (sql)

Second, create an AFTER UPDATE trigger to log data to the lead_logs table whenever there is an update in the email or phone column.

```
CREATE TRIGGER log_contact_after_update
AFTER UPDATE ON leads
WHEN old.phone <> new.phone
OR old.email <> new.email
```

```

BEGIN
    INSERT INTO lead_logs (
        old_id,
        new_id,
        old_phone,
        new_phone,
        old_email,
        new_email,
        user_action,
        created_at
    )
VALUES
    (
        old.id,
        new.id,
        old.phone,
        new.phone,
        old.email,
        new.email,
        'UPDATE',
        DATETIME('NOW')
    ) ;
END;

```

Code language: SQL (Structured Query Language) (sql)

You notice that in the condition in the WHEN clause specifies that the trigger is invoked only when there is a change in either email or phone column.

Third, update the last name of John from Doe to Smith.

```

UPDATE leads
SET
    last_name = 'Smith'
WHERE
    id = 1;

```

Code language: SQL (Structured Query Language) (sql)

The trigger log_contact_after_update was not invoked because there was no change in email or phone.

Fourth, update both email and phone of John to the new ones.

```

UPDATE leads
SET
    phone = '4089998888',
    email = 'john.smith@sqlitetutorial.net'
WHERE
    id = 1;

```

Code language: SQL (Structured Query Language) (sql)

If you check the log table, you will see there is a new entry there.

```
SELECT
  old_phone,
  new_phone,
  old_email,
  new_email,
  user_action
FROM
  lead_logs;
```

Code language: SQL (Structured Query Language) (sql)

old_phone	new_phone	old_email	new_email	user_action
4089009334	4089998888	john.doe@sqlitetutorial.	john.smith@sqlitetutorial.net	UPDATE

You can develop the AFTER INSERT and AFTER DELETE triggers to log the data in the lead_logs table as an exercise.

SQLite DROP TRIGGER statement

To drop an existing trigger, you use the DROP TRIGGER statement as follows:

```
DROP TRIGGER [IF EXISTS] trigger_name;
```

Code language: SQL (Structured Query Language) (sql)

In this syntax:

- First, specify the name of the trigger that you want to drop after the DROP TRIGGER keywords.
- Second, use the IF EXISTS option to delete the trigger only if it exists.

Note that if you drop a table, SQLite will automatically drop all triggers associated with the table.

For example, to remove the validate_email_before_insert_leads trigger, you use the following statement:

```
DROP TRIGGER validate_email_before_insert_leads;
```