SQLite

What is SQLite

SQLite is an SQL database engine, which **doesn't have a separate server process**. SQLite reads and writes directly to disk files. An SQLite database consisting of multiple tables, indices, triggers, and views, is contained within **a single disk file**. The **file format is cross-platform**, meaning it is supported by 32-bit and 64-bit architectures alike, as well as by big-endian vs. little endian architectures.

The size of SQLite with all features enables ranges between $(300\ldots500)\mathrm{kB}$, where $1\mathrm{kB}=2^{10}\mathrm{B}=1{,}024\mathrm{B}$. Because SQLite can run on minimal stack space ($4\mathrm{kB}$) and heap ($100\mathrm{kB}$), it has become a very popular database engine for memory constrained devices such as cellphones, PDAs, and MP3 players. Although there is an inversely proportional relationship between its memory usage and its speed, SQLite can be performant even in low-memory environments.

ACID transactions are database transactions characterized by **atomicity, consistency, isolation, durability.**

An **atomic transaction** is a series of database operations executing as a group, at the same time. These operations cannot be divided apart and executed partially from each other.

A **consistent transaction** obeys the requirement that the transaction can change the data only in allowed ways. Any data written to the database must be valid according to all defined rules, including **constraints**, **cascades**, **triggers**, and any combination thereof.

An **isolated transaction** is a transaction with low visibility to other users and systems. This is necessary to prevent concurrent writes to the same database by different users and systems to generate dirty reads or lost updates. The isolation level of a DB defines when the effects of a transaction become visible to another transaction.

A **durable transaction**, once committed, survives permanently.

SQLite Download

The official SQLite website is sqlite.org. There are versions for Linux, Windows, MacOS X.

SQLite Commands

SQLite Command	Description
.backup ?DB?FILE	Back-up DB (default "main") to **FILE**
.databases	List names and files of attached databases
.dump ?TABLE?	Dump the database in SQL text format. If **TABLE** is specified, only dump tables matching the pattern **TABLE**
.exit	Exit SQLite prompt
.header(s) ON|OFF	Turn display of headers **ON** or **OFF**
.help	Show dot commands
.indices ?TABLE?	Show names of all indices. If **TABLE** is specified, only show indices for tables matching the pattern **TABLE**
.log FILE|off	Turn logging **ON** or **OFF**. **FILE** is either **stderr** or **stdout**
.mode MODE	Set output mode where **MODE** is one of: **csv**—comma-separated values **column**—left-aligned columns **html**—HTML <TABLE> code **insert**—SQL insert statements for TABLE **line**—one value per line **list**—values delimited by .separator string **tabs**—tab-separated values **tcl**—TCL list elements
.nullvalue STRING	Print **STRING** in place of **NULL** values
.output FILENAME	Send output to **FILENAME**
.output stdout	Send output to **stdout** (terminal)

SQLite Command	Description
.print STRING…	Print literal **STRING**
.quit	Exit SQLite prompt
.read FILENAME	Execute SQL in **FILENAME**
.schema ?TABLE?	Show the **CREATE** statements. If **TABLE** is specified, only show tables matching the pattern **TABLE**
.separator STRING	Change separator used by the output mode and **.import** to **STRING**
.show	Show current values for all settings
.stats ON|OFF	Turn stats **ON** or **OFF**
.tables ?PATTERN?	List names of tables matching the pattern **PATTERN**
.width NUM NUM	Set column widths for “column” mode
.timer ON|OFF	Turn the CPU timer measurement **ON** or **OFF**

The following commands were issued for my database, `students.db`:

```
Adrianas-MBP-2:Scripts awise$ sqlite3 students.db
SQLite version 3.8.10.1 2015-05-09 12:14:55
Enter ".help" for usage hints.
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------
0    main             /Users/awise/Python/Scripts/students.db
sqlite> .schema Students
CREATE TABLE Students(ID INTEGER PRIMARY KEY, Name TEXT, Email
TEXT, Grade INT);
sqlite> .dbinfo
database page size:  4096
write format:        1
read format:         1
reserved bytes:      0
file change counter: 52
database page count: 7
freelist page count: 1
schema cookie:       41
schema format:       4
```

```
default cache size:   0
autovacuum top root:  0
incremental vacuum:   0
text encoding:        1 (utf8)
user version:         0
application id:       0
software version:     3008005
number of tables:     5
number of indexes:    0
number of triggers:   0
number of views:      0
schema size:          532
sqlite> .tables
ParticipatesIn  Projects        Situation       Students
Week
sqlite> .fullschema
CREATE TABLE Students(ID INTEGER PRIMARY KEY, Name TEXT, Email
TEXT, Grade INT);
CREATE TABLE ParticipatesIn(StudentID INT, ProjectTitle TEXT,
FOREIGN KEY(StudentID) REFERENCES Students(ID));
CREATE TABLE Projects(Title TEXT, Grade INT, FOREIGN KEY(Title)
REFERENCES ParticipatesIn(ProjectTitle));
CREATE TABLE Situation(StudentID INT, WeekNo INT, FOREIGN
KEY(StudentID) REFERENCES Students(ID), FOREIGN KEY(WeekNo)
REFERENCES Week(Number));
CREATE TABLE Week(WeekNo INTEGER PRIMARY KEY, PDF INT,
WrittenReport INT, Attended INT, Grade INT);
/* No STAT tables available */
```

To format output in column format, the following commands can be issued:

```
sqlite> .header on
sqlite> .mode column
sqlite> .timer on
sqlite> SELECT * FROM Students;
Run Time: real 0.000 user 0.000071 sys 0.000003
sqlite> SELECT * FROM Students;
ID          Name        Email                 Grade
----------  ----------  --------------------  ----------
1           Jeahun AN   hsjaehun@gmail.com    100
2           Qendrim Gj  qgjevukaj@gmail.co    100
3           William Wi  william.widmer15@m    100
4           Yan Zhen L  yanznln@gmail.com     100
5           Richard Hu  richard77927@hotma    100
6           Katherine   katherine.sullivan    100
```

```
7           Shehryar K  shehryar2212@yahoo  100
8           Fazlay Rab  fazlay.rabbi35@myh  100
9           Saad Makhd  saad.makhdumi38@my  100
10          Peter Lena  peter.lenahan84@my  100
11          Manpreet K  mka0019@gmail.com   100
12          Brandon Sh  brandon.shoykhet24  100
13          Carlos Rod  rodriguezCA@gmail.  100
14          Oscar Tong  tong.oscar@gmail.c  100
15          Damian Gli  glina126@gmail.com  100
Run Time: real 0.000 user 0.000183 sys 0.000090
sqlite>
```

Create Database

```
Adrianas-MBP-2:Scripts awise$ sqlite3 testDB.db
SQLite version 3.8.10.1 2015-05-09 12:14:55
Enter ".help" for usage hints.
sqlite> .databases
seq  name             file
---  --------         -------------------------------------
0    main             /Users/awise/Python/Scripts/testDB.db
sqlite>
```

Attach and Detach Database

The ATTACH command selects a particular database from multiple databases. After this, all SQLite statements will be executed for the attached database.

Syntax:

```
ATTACH DATABASE 'DatabaseName' As ‘Alias-Name';
```

Example:

```
sqlite> ATTACH DATABASE 'testDB.db' AS 'TEST';
sqlite> .database
seq  name             file
---  --------------- ------------------------------------
0    main             /Users/awise/Python/Scripts/testDB.db
2    TEST             /Users/awise/Python/Scripts/testDB.db
sqlite>
```

The DETACH command dissociates a database from a database connection, which was previously established with an ATTACH command.

Syntax:

```
sqlite> DETACH DATABASE ‘Alias-Name’;
```

Example:

```
sqlite> DETACH DATABASE 'TEST';
sqlite> .databases
seq  name             file
---  ---------------  ------------------------------------
0    main             /Users/awise/Python/Scripts/testDB.db
```

Create and Drop Table

The CREATE TABLE command will generate a table with a user-specified name in the database, with a user-specified column structure (“schema”).

Syntax:

```
CREATE TABLE database_name.table_name(
   column1 datatype  PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

Example:

```
sqlite> CREATE TABLE Musicians(
   ...> Name TEXT NOT NULL,
   ...> Age INT NOT NULL,
   ...> Address CHAR(50),
   ...> Num_Albums INT
   ...> );
sqlite> CREATE TABLE MemberOf(
   ...> ID INT PRIMARY KEY NOT NULL,
   ...> Band CHAR(50) NOT NULL,
   ...> Musician_ID INT NOT NULL
   ...> );
sqlite> .tables
MemberOf   Musicians
sqlite> .schema Musicians
CREATE TABLE Musicians(
Name TEXT NOT NULL,
Age INT NOT NULL,
Address CHAR(50),
Num_Albums INT
);
sqlite> .schema MemberOf
CREATE TABLE MemberOf(
ID INT PRIMARY KEY NOT NULL,
Band CHAR(50) NOT NULL,
Musician_ID INT NOT NULL
);
```

To delete a table from a database you use the DROP TABLE statement.

Syntax:

```
DROP TABLE database_name.table_name;
```

Example:

```
sqlite> DROP TABLE Musicians;
sqlite> .tables
MemberOf
```

The INSERT Query Statement

This statement is used to populate a table with values.

Syntax:

```
INSERT  INTO  TABLE_NAME  (column1,  column2,  column3,…
columnN) VALUES (value1, value2, value3,…, valueN);
```

If adding values for all columns in the table, the column names need not be listed, but the values must be supplied in the exact order of columns, as specified when the table was created:

```
INSERT  INTO  TABLE_NAME  VALUES  (value1,value2,value3,…
valueN);
```

Example:

```
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(1, 'Lyle PUENTE', 53, 'Crompond, NY', 6);
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(2, 'Tyler JOSEPH', 26, 'Columbus, OH', 4);
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(3, 'Josh DUN', 27, 'Columbus, OH', 3);
```

You can also use another table to populate a given table with data:

```
INSERT  INTO  first_table_name  [(column1,  column2,  ...
columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

The SELECT Query Statement

This statement selects entries from a table or from multiple tables, based on a criterion.

Syntax:

```
SELECT column1, column2,…, columnN FROM table_name;

SELECT * FROM table_name;
```

```
sqlite> .header on
sqlite> .mode column
sqlite> SELECT * FROM Musicians;
ID          Name         Age         Address       Num_Albums
----------  -----------  ----------  ------------  ----------
1           Lyle PUENTE  53          Crompond, NY  6
2           Tyler JOSEP  26          Columbus, OH  4
3           Josh DUN     27          Columbus, OH  3
sqlite>
```

For selecting only a subset of fields from the table:

```
sqlite> SELECT ID, Name, Age FROM Musicians;
ID          Name         Age
----------  -----------  ----------
1           Lyle PUENTE  53
2           Tyler JOSEP  26
3           Josh DUN     27
```

To adjust the column width:

```
sqlite> .width 3 20 3 20 3
sqlite> SELECT * FROM Musicians;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
1    Lyle PUENTE           53   Crompond, NY          6
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
```

To list the tables in the database:

```
sqlite> .width 20
sqlite> SELECT tbl_name FROM sqlite_master WHERE type='table';
tbl_name
--------------------
MemberOf
Musicians
```

Operators

To specify conditions in an SQLite statement, the following types of operators are supported:

- arithmetic operators

- comparison operators
- logical operators
- bitwise operators

Arithmetic Operators

Arithmetic Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Comparison Operators

Comparison Operator	Description
==	equal
=	equal
! =	not equal
<>	not equal
>	strictly greater than
<	strictly less than
>=	greater than or equal to
<=	less than or equal to
! >	not greater than
! <	not less than

Logical Operators

Logical Operator	Description
AND	logical AND
BETWEEN	range of values
EXISTS	checks for row in table
IN	value IN list
NOT IN	value NOT IN list
LIKE	value LIKE wildcard-defined values
GLOB	value like wild-card defined values, case-sensitive
NOT	negate operator
OR	logical OR
IS NULL	compare value to NULL value
IS	same as =
IS NOT	same as ! =
||	concatenates two strings
UNIQUE	searches unique rows

Expressions

There are three types of expressions:

- boolean
- numeric
- date

Boolean Expressions

These have the general syntax:

```
SELECT column1, column2,…, columnN
```

```
FROM table_name
WHERE SINGLE VALUE MATCHING EXPRESSION;
```

Example:

```
sqlite> .width 3 20 3 20 3
sqlite> SELECT * FROM Musicians WHERE Age<=27;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
```

Numeric Expressions

These allow SQL statements to be combined with mathematical expressions, assigning values computed from a table to variables, and displaying the result. For example, the values from a column could be added to create a grand total (the total number of albums for all of our artists in the database).

Syntax:

```
SELECT numerical_expression as OPERATION_NAME
[FROM table_name WHERE CONDITION];
```

Example:

```
sqlite> .width 10
sqlite> SELECT count(*) AS Artists FROM Musicians;
Artists
----------
3
sqlite> .width 15
sqlite> SELECT sum(Num_Albums) AS Total_Albums FROM Musicians;
Total_Albums
---------------
13
sqlite> SELECT avg(Age) AS Average_Age FROM Musicians;
Average_Age
---------------
35.333333333333
```

Date Expressions

These are used to extract current date information that can be used in other database operations.

```
sqlite> .width 20
sqlite> SELECT CURRENT_TIMESTAMP;
CURRENT_TIMESTAMP
--------------------
2015-09-01 19:57:18
```

Table Update and Delete Queries

Table updates are used to modify single field entries in a table row. For example, if we wanted to modify the number of albums for an artist in our `testDB.db` database, we would use an UPDATE statement.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2,…, columnN =
valueN
WHERE [condition];
```

Example:

```
sqlite> UPDATE Musicians
   ...> SET Num_Albums=7
   ...> WHERE Name='Lyle PUENTE’;
sqlite> .width 3 20 3 20 3
sqlite> SELECT * FROM Musicians;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
1    Lyle PUENTE           53   Crompond, NY          7
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
```

The DELETE statement removes an entire row from a table, satisfying a condition:

```
DELETE FROM table_name
```

```
WHERE [condition];
```

The INSERT statement adds an entire row to the table:

```
INSERT INTO TABLE_NAME (column1, column2, column3,…,
columnN)
VALUES (value1, value2, value3,…valueN);
```

Example:

```
sqlite> INSERT INTO Musicians (ID, Name, Age, Address,
Num_Albums)
   ...> VALUES (4, 'Pointhead LARRY', 55, 'Los Angeles, CA', 0);
sqlite> SELECT * FROM Musicians;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
1    Lyle PUENTE           53   Crompond, NY          7
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
4    Pointhead LARRY       55   Los Angeles, CA       0
```

…And now, to remove it:

```
sqlite> DELETE FROM Musicians
   ...> WHERE ID=4;
sqlite> SELECT * FROM Musicians;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
1    Lyle PUENTE           53   Crompond, NY          7
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
```

Pattern Matching with LIKE and GLOB

To allow selection of records (table rows) based on imprecise (partial) data, the SELECT statement can also work with pattern matching.

Syntax:

```
SELECT FROM table_name
```

```
WHERE column LIKE 'XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE '%XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE 'XXXX_'
```

or

```
SELECT FROM table_name
WHERE column LIKE '_XXXX'
```

or

```
SELECT FROM table_name
WHERE column LIKE ‘_XXXX_'
```

The “%” wildcard allows any number of characters in its place, while the “_” wildcard allows exactly one character in its place.

Example:

```
sqlite> SELECT * FROM Musicians
   ...> WHERE Name LIKE '_yle%';
ID    Name                    Age   Address                 Num
---   --------------------    ---   --------------------    ---
1     Lyle PUENTE             53    Crompond, NY            7
2     Tyler JOSEPH            26    Columbus, OH            4
```

The GLOB pattern matching works the same way, with the difference that it is case-sensitive.

The LIMIT Statement

This allows the display of only a limited number of rows from the table, with a given offset.

Syntax:

```
SELECT column1, column2,…, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

Example:

```
sqlite> SELECT * FROM Musicians
   ...> LIMIT 2 OFFSET 1;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
```

The ORDER, GROUP, and HAVING Statements

The ORDER BY statement allows the rearrangement of the table rows to sort them according to an order on one of the fields. For instance, in our `testDB.db`, we could rearrange the artists in increasing order of age.

Syntax:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Example:

```
sqlite> SELECT * FROM Musicians
   ...> ORDER BY Age ASC;
ID   Name                  Age  Address               Num
---  --------------------  ---  --------------------  ---
```

```
2    Tyler JOSEPH          26   Columbus, OH          4
3    Josh DUN              27   Columbus, OH          3
1    Lyle PUENTE           53   Crompond, NY          7
```

The GROUP statement allows aggregation of data pertaining to rows from the same group. For instance, in our `testDB.db`, Tyler JOSEPH and Josh DUN are members of the same band, “Twenty One Pilots”. We could group them by that criterion and sum their respective numbers of albums.

Syntax:

For this purpose, we need to add to the main table of our database, `Musicians`, another field referencing the PK ID from the `MemberOf` table. We do this by creating a FOREIGN KEY field called `Band_ID`. Since we’ve already created our table, we need to:

1. drop the old `Musicians` table
2. re-create the schema, which should now include the foreign key
3. re-insert values for each row, to reference explicitly the `ID` for each band

These steps are shown below:

```
sqlite> DROP TABLE Musicians;
sqlite> DROP TABLE MemberOf;
sqlite> CREATE TABLE MemberOf(
   ...> ID INT PRIMARY KEY NOT NULL,
   ...> Band CHAR(50),
   ...> Since INT);
sqlite> INSERT INTO MemberOf(ID, Band, Since)
   ...> VALUES (1, 'My Brothers Banned', 1996);
sqlite> INSERT INTO MemberOf(ID, Band, Since)
   ...> VALUES (2, 'Twenty One Pilots', 2009);
sqlite> .width 3 20 4
sqlite> SELECT * FROM MemberOf;
ID   Band                  Since
---  --------------------  ----
1    My Brothers Banned    1996
2    Twenty One Pilots     2009
sqlite> CREATE TABLE Musicians(
   ...> ID INT PRIMARY KEY NOT NULL,
   ...> Name CHAR(20),
   ...> Age INT,
   ...> Address CHAR(20),
   ...> Num_Albums INT,
   ...> Band_ID INT,
   ...> FOREIGN KEY(Band_ID) REFERENCES MemberOf(ID)
   ...> );
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums, Band_ID)
   ...> VALUES(1, 'Lyle PUENTE', 53, 'Crompond, NY', 6, 1);
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums, Band_ID)
   ...> VALUES(2, 'Tyler JOSEPH', 26, 'Columbus, OH', 4, 2);
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums, Band_ID)
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums, Band_ID)
   ...> VALUES(3, 'Josh DUN', 27, 'Columbus, OH', 3, 2);
sqlite> SELECT * FROM Musicians;
ID   Name                  Age   Address               Num
Band_ID
---  --------------------  ----  --------------------  ---  ---
1    Lyle PUENTE           53    Crompond, NY          6    1
2    Tyler JOSEPH          26    Columbus, OH          4    2
3    Josh DUN              27    Columbus, OH          3    2
```

Now we're ready to use the GROUP BY statement usefully, to aggregate the number of albums from singers belonging to the same band:

```
sqlite> SELECT sum(Num_Albums) FROM Musicians
   ...> GROUP BY Band_ID
   ...> ORDER BY Age DESC;
sum
---
6
7
sqlite> .width 20 15
sqlite> SELECT Name, sum(Num_Albums) FROM Musicians
   ...> GROUP BY Band_ID
   ...> ORDER BY Age DESC;
Name                  sum(Num_Albums)
--------------------  ---------------
Lyle PUENTE           6
Josh DUN              7
```

If we further want to filter the results of the GROUP BY operation, and show only those that satisfy a condition, we use the HAVING statement. So, after grouping the musicians into bands, we want to know which group has more albums:

```
sqlite> SELECT Name, sum(Num_Albums) FROM Musicians
   ...> GROUP BY Band_ID
   ...> HAVING sum(Num_Albums)>6;
Name                  sum(Num_Albums)
--------------------  ---------------
Josh DUN              7
```

Constraints

Constraints enforce that field values obey user-specified limitations, such as non-null fields, positive valued fields to avoid garbage data, uniqueness of field values where repeats wouldn't make sense, default values where field values are unavailable but not essential.

The following table summarizes these constraints:

Constraint	Description
NOT NULL	a column (field) cannot have a NULL value (usually the PRIMARY KEY)
DEFAULT	provides a default value when none specified (e.g. a minimum of 1 album for each artist, otherwise they wouldn't be in the database!)
UNIQUE	all column (field) values are different (no two rows can have identical entries for a particular column, e.g. no two bands can have the same name for registered trademark purposes)
PRIMARY KEY	unique for each row (across rows), since it is the identifier of each row
CHECK	column (field) values satisfy a certain condition (e.g. the Age cannot be <0)

Joins

Joins allow the display of data from multiple tables to suit cross-reference purposes. There are three types of joins:

1. CROSS JOIN—all the rows of Table 1, each one with all the rows of Table 2
2. INNER JOIN—a refinement of a CROSS JOIN on a condition, requiring a match in Table 2 of a row satisfying that condition

3. OUTER JOIN:

   LEFT OUTER JOIN: all rows of Table 1, even though there are no matches in Table 2 satisfying the condition (e.g. a solo artist, no band)

   RIGHT OUTER JOIN: all rows of Table 2, even though there are no matches Table 1 satisfying the condition

Example CROSS JOIN:

```
sqlite> .width 3 20 30
sqlite> SELECT Musicians.ID, Name, Band FROM
   ...> Musicians CROSS JOIN MemberOf;
ID   Name                  Band
---  --------------------  ------------------------------
1    Lyle PUENTE           My Brothers Banned
1    Lyle PUENTE           Twenty One Pilots
2    Tyler JOSEPH          My Brothers Banned
2    Tyler JOSEPH          Twenty One Pilots
3    Josh DUN              My Brothers Banned
3    Josh DUN              Twenty One Pilots
```

Example INNER JOIN:

```
sqlite> SELECT Musicians.ID, Name, Band FROM
   ...> Musicians INNER JOIN MemberOf
   ...> ON Musicians.Band_ID=MemberOf.ID;
ID   Name                  Band
---  --------------------  ------------------------------
1    Lyle PUENTE           My Brothers Banned
2    Tyler JOSEPH          Twenty One Pilots
3    Josh DUN              Twenty One Pilots
```

Example LEFT OUTER JOIN:

For this example, we would need to create an entry in the `Musicians` table containing a solo artist, belonging to no band. We do this with INSERT:

```
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(4, 'Sara Bareilles', 35, 'Eureka, CA', 2);
sqlite> .width 3 20 3 15 3 7
sqlite> SELECT * FROM Musicians;
ID   Name                  Age  Address          Num  Band_ID
---  --------------------  ---  ---------------  ---  -------
1    Lyle PUENTE           53   Crompond, NY     6    1
```

```
2    Tyler JOSEPH          26   Columbus, OH     4     2
3    Josh DUN              27   Columbus, OH     3     2
4    Sara Bareilles        35   Eureka, CA       2
```

An INNER JOIN would render only those artists for whom there are entries in the MemberOf table:

```
sqlite> .width 3 15 17
sqlite> SELECT Musicians.ID, Name, Band FROM
   ...> Musicians INNER JOIN MemberOf
   ...> ON Musicians.Band_ID=MemberOf.ID;
ID   Name              Band
---  ---------------   -----------------
1    Lyle PUENTE       My Brothers Banned
2    Tyler JOSEPH      Twenty One Pilots
3    Josh DUN          Twenty One Pilots
```

However, a LEFT OUTER JOIN would also include Sara Bareilles, who is a solo artist (for whom there is no entry in table MemberOf):

```
sqlite> SELECT Musicians.ID, Name, Band FROM
   ...> Musicians LEFT OUTER JOIN MemberOf
   ...> ON Musicians.Band_ID=MemberOf.ID;
ID   Name              Band
---  ---------------   -----------------
1    Lyle PUENTE       My Brothers Banned
2    Tyler JOSEPH      Twenty One Pilots
3    Josh DUN          Twenty One Pilots
4    Sara Bareilles
```

Now…, why do we even need a CROSS JOIN?! In trying to answer this question, we need to add three more tables, `Albums`, `Stores`, and `Sales`. By coding a CROSS JOIN of `Albums` with `Stores`, we can generate all (`Album`, `Store`) combinations. Now, if table Sales has columns `Album`, `Store`, `Num_Sold`, we can take the LEFT OUTER JOIN of the CROSS JOIN with `Sales`, to show all the sales per album and store, including the 0 sales, which a GROUP BY `Store` in the `Sales` table wouldn't show.

Unions

A UNION statement will combine the results of two or more SELECT statements from different tables without duplicates. For example, if we had two different tables in our database, one for musicians who are part of a band, and one for solo artists, a UNION statement would be able to merge the two tables without duplicates. For instance, Tyler JOSEPH has a solo album as well, so he'd be part of both tables.

Syntax:

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Example:

```
sqlite> CREATE TABLE SoloArtists(ID INT PRIMARY KEY NOT NULL,
   ...> Name CHAR(20),
   ...> Age INT,
   ...> Address CHAR(20),
   ...> Num_Albums INT,
   ...> Band_ID INT,
   ...> FOREIGN KEY(Band_ID) REFERENCES MemberOf(ID)
   ...> );
sqlite> INSERT INTO SoloArtists(ID, Name, Age, Address,
Num_Albums, Band_ID)
   ...> VALUES(1, 'Lyle PUENTE', 53, 'Crompond, NY', 2, 1);
sqlite> INSERT INTO SoloArtists(ID, Name, Age, Address,
Num_Albums, Band_ID)
   ...> VALUES(2, 'Tyler JOSEPH', 26, 'Columbus, OH', 1, 2);
sqlite> INSERT INTO SoloArtists(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(3, 'Sara BAREILLES', 35, 'Eureka, CA', 2);
sqlite> SELECT * FROM SoloArtists;
ID   Name             Age                Address          Num
Band_ID
---  ---------------  -----------------  ---------------  ---
-------
1    Lyle PUENTE      53                 Crompond, NY     2    1
2    Tyler JOSEPH     26                 Columbus, OH     1    2
```

```
3    Sara BAREILLES   35                Eureka, CA       2
```

Now we want to show that the UNION of tables Musicians and SoloArtists will yield a listing of all artists, without duplicates.

```
sqlite> SELECT ID, Name, Age FROM Musicians
   ...> UNION
   ...> SELECT ID, Name, Age FROM SoloArtists;
ID   Name             Age
---  ---------------  -----------------
1    Lyle PUENTE      53
2    Tyler JOSEPH     26
3    Josh DUN         27
3    Sara BAREILLES   35
4    Sara BAREILLES   35
```

Oops! What happened? We selected a column (field) with unique values, and since Sara BAREILLES appears with different IDs in each table, the rows were listed in the UNION as distinct.

But if we omit the ID in the SELECT statement, we get only de-duplicated rows:

```
sqlite> .width 20 3
sqlite> SELECT Name, Age FROM Musicians
   ...> UNION
   ...> SELECT Name, Age FROM SoloArtists;
Name                  Age
--------------------  ---
Josh DUN              27
Lyle PUENTE           53
Sara BAREILLES        35
Tyler JOSEPH          26
```

Triggers

The TRIGGER command creates events based on database operations. For instance, we could log each new insertion into table Musicians and save that into a new table, which we will name Logs.

Syntax:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
```

```
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Example:

```
sqlite> CREATE TRIGGER audit AFTER INSERT
   ...> ON Musicians
   ...> BEGIN
   ...> INSERT INTO Logs(ID, Date)
   ...> VALUES(new.ID, datetime('now'))
   ...> ;
   ...> END;
Run Time: real 0.004 user 0.000310 sys 0.000834
sqlite> INSERT INTO Musicians(ID, Name, Age, Address,
Num_Albums)
   ...> VALUES(5, 'Robin THICKE', 35, 'Los Angeles, CA', 1);
Run Time: real 0.003 user 0.000193 sys 0.000879
sqlite> SELECT * FROM Musicians;
ID          Name         Age         Address       Num_Albums
Band_ID
----------  -----------  ----------  ------------  ----------
----------
1           Lyle PUENTE  53          Crompond, NY  6           1
2           Tyler JOSEP  26          Columbus, OH  4           2
3           Josh DUN     27          Columbus, OH  3           2
4           Sara BAREIL  35          Eureka, CA    2
5           Robin THICK  35          Los Angeles,  1
Run Time: real 0.001 user 0.000151 sys 0.000081
sqlite> SELECT * FROM Logs;
ID          Date
----------  -------------------
5           2015-09-02 14:05:51
Run Time: real 0.000 user 0.000100 sys 0.000043
```