#Data type:

Different from other database systems, SQLite uses dynamic data 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 and 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.

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.

Q?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.

TYPEOF(val): it is a function which gets the data type of each value stored in a column (here val).

set of rules applies in sorting the data types: (order: NULL, INTEGER and REAL, TEXT, BLOB )

* 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.

Within each specific data type, the data will be sorted based on their value.

* 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.

###################################################################################

#Date and time:

To store the date and time in the format YYYY-MM-DD HH:MM:SS.SSS, TEXT type data for corresponding column should be used.

CREATE TABLE datetime\_text(d1 text, d2 text);

INSERT INTO datetime\_text (d1, d2) VALUES(datetime('now'),datetime('now', 'localtime'));

#remark: datetime('now') gives UTC date and time value.

# to store date and time in the format of [Julian day numbers](https://en.wikipedia.org/wiki/Julian_day), which is the number of days since noon in Greenwich on November 24, 4714 B.C. based on the proleptic Gregorian calendar, REAL type data for corresponding column should be used

CREATE TABLE datetime\_real(d1 real);

INSERT INTO datetime\_real (d1) VALUES(julianday('now'));

#the output is not human readable and the built-in [date()](https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-date-function/) and [time()](https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-time-function/) functions can be used to format a date and time value:

SELECT date(d1), time(d1)FROM datetime\_real;

# the INTEGER can be used to store UNIX time which is the number of seconds since 1970-01-01 00:00:00 UTC.

CREATE TABLE datetime\_int (d1 int);

INSERT INTO datetime\_int (d1) VALUES(strftime('%s','now'));

#the built-in [datetime()](https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-datetime-function/) function can be used to format the output in YYYY-MM-DD HH:MM:SS.SSS format:

SELECT datetime(d1,'unixepoch')FROM datetime\_int;

#####################################################################################

# foreign key constraints at runtime

It can be used to relate two tables, child table to parent table.

SQLite supports foreign key constraint since version 3.6.19. The SQLite library must also be compiled with neither [SQLITE\_OMIT\_FOREIGN\_KEY](https://www.sqlite.org/compile.html#omit_foreign_key) nor [SQLITE\_OMIT\_TRIGGER](https://www.sqlite.org/compile.html#omit_trigger). In order to check whether the current version of SQLite supports foreign key constraints:

PRAGMA foreign\_keys;

The output integer value: 1: enable, 0: disabled, returning nothing means that your SQLite version doesn’t support foreign key constraints. If get 1 which means SQLite is compiled with foreign key constraints, foreign key constraints at runtime can be enabled or disabled using:

PRAGMA foreign\_keys = OFF;

PRAGMA foreign\_keys = ON;

The Foreign key is a column which should be defined in creating of the child table and refers to a column in the parent table (typically the primary key of the parent key). This column is called foreign key in child table and parent key in parent table.

CREATE TABLE suppliers (

supplier\_id INTEGER PRIMARY KEY,

supplier\_name TEXT NOT NULL,

group\_id INTEGER NOT NULL,

FOREIGN KEY (group\_id)

REFERENCES supplier\_groups (group\_id));

#trying to make a change in child table which is not correspond to the relative one in parent table raises error. Eg. Insert row in child table with specific child key where there is no row for that key in the parent table.

Abort due to constraint violation (FOREIGN KEY constraint failed)

#change (delete or update parent keys) in the parent table also can be problematic because the corresponding foreign (child) key in the child table will be affected. The specific ON UPDATE/DELETE action in the CREATE TABLE of the child table can be defined to let SQLite perform in specific way as it finds change in the parent key.

FOREIGN KEY (foreign\_key\_columns)

REFERENCES parent\_table(parent\_key\_columns)

ON UPDATE action

ON DELETE action;

One of these actions can be considered:

1) SET NULL: this action set the child key NULL as the corresponding parent key is deleted or updated.

2) SET DEFAULT: this action set the child key to the default defined value (which is NULL) as the corresponding parent key is deleted or updated.

3) RESTRICT: this action does not allow any change in the parent key and returns error if such changes are requested.

4) NO ACTION: performs like RESTRICT and does not allow any change in the parent key and returns error if such changes are requested.

5) CASCADE: results in the same changes in the corresponding child key, if a change happens in the parent key.

#####################################################################################

#PRIMARY KEY

#consist of one or more column INTEGER:

CREATE TABLE table\_name(column\_1 INTEGER NOT NULL PRIMARY KEY,...);

#or

CREATE TABLE table\_name(

column\_1 INTEGER NOT NULL,

column\_2 INTEGER NOT NULL,

...

PRIMARY KEY(column\_1,column\_2,...));

#if there are more than one child key in the child table, actions on update and/or delete for each child key should be separately defined:

CREATE TABLE country\_languages (

country\_id INTEGER NOT NULL,

language\_id INTEGER NOT NULL,

PRIMARY KEY (country\_id, language\_id),

FOREIGN KEY (country\_id) REFERENCES countries (country\_id)

ON DELETE CASCADE ON UPDATE NO ACTION,

FOREIGN KEY (language\_id) REFERENCES languages (language\_id)

ON DELETE CASCADE ON UPDATE NO ACTION);

######################################################################################rowid:

When you [create a table](https://www.sqlitetutorial.net/sqlite-create-table/) without specifying the WITHOUT ROWID option, SQLite adds an implicit column called rowid that stores 64-bit signed integer. The rowid column is a key that uniquely identifies the rows in the table. Tables that have rowid columns are called rowid tables.

If a table has the primary key that consists of one column, and that column is defined as INTEGER then this primary key column becomes an alias for the rowid column.

#If the rowid value is not specified or a NULL value is used to insert a new row, SQLite automatically assigns the next sequential integer, which is one larger than the largest rowid in the table. The rowid value starts at 1. The maximum value of the rowid column is 9,223,372,036,854,775,807, which is very big. If your data reaches this maximum value and you attempt to insert a new row, SQLite will find an unused integer and uses it. If SQLite cannot find any unused integer, it will issue an SQLITE\_FULL error. On top of that, if you delete some rows and insert a new row, SQLite will try to reuse the rowid values from the deleted rows.

#remark: the attribute AUTOINCREMENT can prevent reuse of a value that has not been used or a value from the previously deleted row. If you don’t have any requirement like this, you should not use the AUTOINCREMENT attribute in the primary key.

#remark: if you assign another integer type such as BIGINT and UNSIGNED INT to the primary key column, this column will not be an alias for the rowid column.

Unsigned Integers (often called "uints") are just like integers (whole numbers) but have the property that they don't have a + or - sign associated with them. Thus they are always non-negative (zero or positive). We use uint's when we know the value we are counting will always be non-negative.

The BigInt data type in SQL Server is the 64-bit representation of an integer. It takes up 8 bytes of storage. It can range from -2^63 (-9,223,372,036,854,775,808) to 2^63 (9,223,372,036,854,775,807). BigInt types in SQL Server are useful for the primary key in very large tables.

#remark: if you declare a column with the INTEGER type and PRIMARY KEY DESC clause, this column will not become an alias for the rowid column:

Because the rowid table organizes its data as a B-tree?, querying and sorting data of a rowid table are very fast. It is faster than using a primary key which is not an alias of the rowid.

In SQL standard, the primary key column must not contain NULL values. It means that the primary key column has an implicit [NOT NULL](https://www.sqlitetutorial.net/sqlite-not-null-constraint/) constraint.

However, to make the current version of SQLite compatible with the earlier version, SQLite allows the primary key column to contain NULL values.

#adding primary key to the existing table that does not have it is not possible (unlike other database systems e.g., MySQL and PostgreSQL) using the [ALTER TABLE](https://www.sqlitetutorial.net/sqlite-alter-table/) statement. The new table with the primary key should be created and the data from the existing table to this new table should be inserted following these steps:

1. set the [foreign key](https://www.sqlitetutorial.net/sqlite-foreign-key/) constraint check off.

2. rename the table to another table name (old\_table)

3. create a new table (table) with exact structure of the table that you have been renamed but this time with the primary key column.

4. copy data from the old\_table to the table.

5. turn on the foreign key constraint check on

PRAGMA foreign\_keys=off;

BEGIN TRANSACTION;

ALTER TABLE table RENAME TO old\_table;

CREATE TABLE table ( ... );

INSERT INTO table SELECT \* FROM old\_table;

COMMIT;

PRAGMA foreign\_keys=on;

#####################################################################################

#UNIQUE constraint: this constraint can be set on the data values of one or more columns in one of below ways:

CREATE TABLE table\_name(

...,

column\_name type UNIQUE,

...);

CREATE TABLE table\_name(

...,

UNIQUE(column\_name));

CREATE TABLE table\_name(

...,

UNIQUE(column\_name1,column\_name2,...));

#####################################################################################

#CHECK constraint:

This constraints allow you to define additional data integrity checks beyond [UNIQUE](https://www.sqlitetutorial.net/sqlite-unique-constraint/) or [NOT NULL](https://www.sqlitetutorial.net/sqlite-not-null-constraint/) to suit your specific application. It could be defined at the column level or the table level.

#column level:

CREATE TABLE table\_name(

...,

column\_name data\_type CHECK(expression),

...

);

#eg.

CREATE TABLE contacts (

contact\_id INTEGER PRIMARY KEY,

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

email TEXT,

phone TEXT NOT NULL CHECK (length(phone) >= 10)

);

#table level:

CREATE TABLE table\_name(

...,

CHECK(expression));

# by defining a CHECK constraint at the table level, whenever a row is inserted into a table or an existing row is updated, the expression associated with each CHECK constraint is evaluated and returned a numeric value 0 or 1. If the result is zero, then a constraint violation occurred. If the result is a non-zero value or NULL, it means no constraint violation occurred.

#eg.

CREATE TABLE products (

product\_id INTEGER PRIMARY KEY,

product\_name TEXT NOT NULL,

list\_price DECIMAL (10, 2) NOT NULL,

discount DECIMAL (10, 2) NOT NULL DEFAULT 0,

CHECK (list\_price >= discount AND discount >= 0 AND list\_price >= 0));

#remark: the expression of a CHECK constraint cannot contain a [subquery](https://www.sqlitetutorial.net/sqlite-subquery/).

# As of version 3.25.2, SQLite does not support adding a CHECK constraint to an existing table but the new exactly the same table as the existing one can be created with the required CHECK constraints.

BEGIN;

-- create a new table

CREATE TABLE new\_table (

[...],

CHECK ([...])

);

-- copy data from old table to the new one

INSERT INTO new\_table SELECT \* FROM old\_table;

-- drop the old table

DROP TABLE old\_table;

-- rename new table to the old one

ALTER TABLE new\_table RENAME TO old\_table;

-- commit changes

COMMIT;

#####################################################################################

#ALTER TABLE:

#using this statement, these actions can be done:

1) Rename a table

ALTER TABLE existing\_table RENAME TO new\_table;

#remarks:

#-The ALTER TABLE only renames a table within a database. It cannot be used to move the table between the attached databases.

#-The database objects such as [indexes](https://www.sqlitetutorial.net/sqlite-index/) and [triggers](https://www.sqlitetutorial.net/sqlite-trigger/) associated with the table will be associated with the new table.

#-If a table is referenced by [views](https://www.sqlitetutorial.net/sqlite-create-view/) or statements in [triggers](https://www.sqlitetutorial.net/sqlite-trigger/), the definition of views and triggers should be changed manually.

2) Add a new column to a table: the new column will be appended at the end of the existing column list.

ALTER TABLE table\_name ADD COLUMN column\_definition;

# some restrictions on the new column:

#-The new column cannot have a [UNIQUE](https://www.sqlitetutorial.net/sqlite-unique-constraint/) or [PRIMARY KEY](https://www.sqlitetutorial.net/sqlite-primary-key/) constraint.

#-If the new column has a [NOT NULL](https://www.sqlitetutorial.net/sqlite-not-null-constraint/) constraint, you must specify a default value for the column other than a NULL value.

#-The new column cannot have a default of CURRENT\_TIMESTAMP, CURRENT\_DATE, and CURRENT\_TIME, or an expression.

#-If the new column is a foreign key and the [foreign key constraint](https://www.sqlitetutorial.net/sqlite-foreign-key/) check is enabled, the new column must accept a default value NULL.

3) [Rename a column](https://www.sqlitetutorial.net/sqlite-rename-column/) (added supported in version 3.20.0)

ALTER TABLE table\_name RENAME COLUMN current\_name TO new\_name;

4) to perform other actions e.g., drop a column, you use the following steps:

-- disable foreign key constraint check

PRAGMA foreign\_keys=off;

-- start a transaction

BEGIN TRANSACTION;

-- Here you can drop column

CREATE TABLE IF NOT EXISTS new\_table(

column\_definition,

...

);

-- copy data from the table to the new\_table

INSERT INTO new\_table(column\_list)

SELECT column\_list

FROM table;

-- drop the table

DROP TABLE table;

-- rename the new\_table to the table

ALTER TABLE new\_table RENAME TO table;

-- commit the transaction

COMMIT;

-- enable foreign key constraint check

PRAGMA foreign\_keys=on;

######################################################################

#remarks on DROP TABLE:

# the DROP TABLE statement deletes the table from the database and the file on disk completely. You will not be able to undo or recover from this action. Therefore, you should perform it with extra caution.

#To remove multiple tables, you need to issue multiple DROP TABLE statements.

#if you want to remove a table in a specific database, you us the [schema\_name.] explicitly. In case that the table has dependent objects such as [triggers](https://www.sqlitetutorial.net/sqlite-trigger/) and [indexes](https://www.sqlitetutorial.net/sqlite-index/), the DROP TABLE statement also removes all the dependent objects.

#The DROP TABLE statement performs an implicit [DELETE](https://www.sqlitetutorial.net/sqlite-delete/) statement before dropping the table. However, the DROP TABLE statement removes the triggers associated with the table before performing the implicit DELETE statement, therefore, the delete triggers will not fire.

#If the [foreign key](https://www.sqlitetutorial.net/sqlite-foreign-key/) constraints enabled and you perform the DROP TABLE statement, before SQLite performs the implicit DELETE statement, it carries the foreign key constraints check. If a foreign key constraint violation occurs (delete the parent table), SQLite issues an error message and will not drop the table.

To remove the parent table without getting that violation error, these steps should be followed:

1. Disable foreign key constraints.
2. Drop the parent table.
3. Update the foreign key in the child table table to NULL values.
4. Enable the foreign key constraints.

#####################################################################################

#VIEW

In database theory, a view is a result set of a stored [query](https://www.sqlitetutorial.net/sqlite-select/). It is the way to pack a query into a named object stored in the database. You can access the data of the underlying tables through a view. The tables that the query in the view definition refers to are called base tables.

A view is useful in some cases:

1) They provide an abstraction layer over tables. Columns can be added or removed in the view without touching the schema of the underlying tables.

2) They can be used to encapsulate complex queries with [joins](https://www.sqlitetutorial.net/sqlite-inner-join/) to simplify the data access.

#view is read only. It means you cannot use [INSERT](https://www.sqlitetutorial.net/sqlite-insert/), [DELETE](https://www.sqlitetutorial.net/sqlite-delete/), and [UPDATE](https://www.sqlitetutorial.net/sqlite-update/) statements to update data in the base tables through the view.

CREATE [TEMP] VIEW [IF NOT EXISTS] view\_name[(column-name-list)] AS select-statement;

#remark: if the TEMP or TEMPORARY option is used, the view to be only visible in the current database connection. The view is called a temporary view and SQLite automatically removes the temporary view whenever the database connection is closed.

# By default, the columns of the view derive from the result set of the SELECT statement. However, you can assign the names of the view columns that are different from the column name of the table

#DROP VIEW:

DROP VIEW [IF EXISTS] [schema\_name.]view\_name;

specify the schema of the view that you want to delete. View Schema defines the design of the database at the view level of the data abstraction. It defines how an end-user will interact with the database system. There are many view schema for a database system.

###################################################################################

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

#INDEX:

#a table as a list of pairs: (rowid, row). Unlike a table, an index has an opposite relationship: (row, rowid). An index is an additional data structure that helps improve the performance of a query.

SQLite uses B-tree for organizing indexes. Note that **B** stands for balanced, B-tree is a balanced tree, not a binary tree.

Q? The B-tree keeps the amount of data at both sides of the tree balanced so that the number of levels that must be traversed to locate a row is always in the same approximate number. In addition, querying using equality (=) and ranges (>, >=, <,<=) on the B-tree indexes are very efficient.

#Each index must be associated with a specific table. An index consists of one or more columns, but all columns of an index must be in the same table. A table may have multiple indexes. The index contains data from the columns that you specify in the index and the corresponding rowid value.

CREATE [UNIQUE] INDEX index\_name ON table\_name(column\_list);

#remark: you want to make sure that values in one or more columns are unique like email and phone, you use the UNIQUE option in the CREATE INDEX statement. The CREATE UNIQUE INDEX creates a new unique index.

CREATE UNIQUE INDEX index\_name ON table\_name(column\_list);

# after creating index for a table, any [query data](https://www.sqlitetutorial.net/sqlite-select/) from the table will be performed based using the index to locate the data. Therefore, if unique index for a table is created, and then we try to insert the same row in the table, we get an error because the unique index has been violated.

We can check if the index is used or not like what is done in below example:

EXPLAIN QUERY PLAN

SELECT

first\_name,

last\_name,

email

FROM

contacts

WHERE

email = 'lisa.smith@sqlitetutorial.net';

#query planner is a software component that determines the best algorithm or query plan to execute an SQL statement.

#remark: If you create an index that consists of one column, SQLite uses that column as the sort key. In case you create an index that has multiple columns, SQLite uses the additional columns as the second, third, … as the sort keys. SQLite [sorts the data](https://www.sqlitetutorial.net/sqlite-order-by/) on the multicolumn index by the first column specified in the CREATE INDEX statement. Then, it sorts the duplicate values by the second column, and so on.

Therefore, the column order is very important when you create a multicolumn index.

To utilize a multicolumn index, the query must contain the condition that has the same column order as defined in the index.

#To find all indexes associated with a table, you use the following command:

PRAGMA index\_list('table\_name');

#To get the information about the columns in an index, you use the following command:

PRAGMA index\_info(index\_name');

# to get all indexes from a database is to query from the sqlite\_master table:

SELECT type, name, tbl\_name, sql FROM sqlite\_master WHERE type= 'index';

#To remove an index from a database:

DROP INDEX [IF EXISTS] index\_name;

#Besides the normal indexes, SQLite allows you to form an index based on expressions involved table columns. This kind of index is called an expression based index.

#eg. Creating this index (not based on the column (company) of table (customers) but is based on the new considered column (LENGTH(company))):

CREATE INDEX customers\_length\_company

ON customers(LENGTH(company));

After creating this index, queries those involve LENGTH(company) are performed using index which is faster through EXPLAIN QUERY PLAN:

EXPLAIN QUERY PLAN

SELECT customerid,

company

FROM customers

WHERE length(company) > 10

ORDER BY length(company) DESC;

The EXPLAIN QUERY PLAN will be used if the expression, which is specified in in the CREATE INDEX statement, appears the same as in the [WHERE](https://www.sqlitetutorial.net/sqlite-where/) clause or [ORDER BY](https://www.sqlitetutorial.net/sqlite-order-by/) clause.

# restrictions on the expression that appears in the CREATE INDEX statement:

1) The expression must refer to the columns of the table that is being indexed only. It cannot refer to the columns of other tables.

2) The expression can only use the deterministic [function](https://www.sqlitetutorial.net/sqlite-functions/) call.

3) The expression cannot use a subquery.

###################################################################################### TRIGGER:

Triggers are used mostly for two main tasks:

1) Enable sophisticated auditing. eg. log the changes (put in another table) in the sensitive data of a table such as salary and address whenever it changes.

2) Enforce complex business rules centrally at the database level and prevent invalid transactions. Eg. Check the validation of data (format of phone numbers or email addresses) before INSERT into the table.

CREATE TRIGGER [IF NOT EXISTS] trigger\_name

[BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]

ON table\_name

[WHEN condition]

BEGIN

statements;

END;

#determine when the trigger is fired such as BEFORE, AFTER, or [INSTEAD OF](https://www.sqlitetutorial.net/sqlite-instead-of-triggers/). You can create BEFORE and AFTER triggers on a table. However, you can only create an [INSTEAD OF](https://www.sqlitetutorial.net/sqlite-instead-of-triggers/) trigger on a view. The event that causes the trigger to be invoked should be specified such as INSERT, UPDATE, or DELETE. After that, indicate the table to which the trigger belongs. The trigger logic in the BEGIN END block should be placed, which can be any valid SQL statements.

# 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.

#if you [drop a table](https://www.sqlitetutorial.net/sqlite-drop-table/), all associated triggers are also deleted. Thus you by drop or rename a table, the definition of the triggers associated by that table should be changed manually.

#Q? 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 |

#DROP TRIGGER:

DROP TRIGGER [IF EXISTS] trigger\_name;

#####################################################################################

#VACUUM

VACUUM can be used to prevent fragmented indices and make a clean smaller size of database and even a backup for the database and it is usefull because:

1) By [drop database](https://www.sqlitetutorial.net/sqlite-drop-table/) objects such as [tables](https://www.sqlitetutorial.net/sqlite-drop-table/), [views](https://www.sqlitetutorial.net/sqlite-create-view/), [indexes](https://www.sqlitetutorial.net/sqlite-index/), and [triggers](https://www.sqlitetutorial.net/sqlite-trigger/) or [delete data from tables](https://www.sqlitetutorial.net/sqlite-delete/), the database file size remains unchanged. Because SQLite just marks the deleted objects as free and reserves it for the future uses. As a result, the size of the database file always grows in size.

2) By [insert](https://www.sqlitetutorial.net/sqlite-insert/) or [delete data](https://www.sqlitetutorial.net/sqlite-delete/) from the tables, the indexes and tables become fragmented, especially for the database that has a high number of inserts, updates, and deletes.

3) [insert](https://www.sqlitetutorial.net/sqlite-insert/), [update](https://www.sqlitetutorial.net/sqlite-update/) and [delete](https://www.sqlitetutorial.net/sqlite-delete/) operations create unused data block within individual database pages. It decreases the number of rows that can be stored in a single page. Therefore, it increases the number of pages to hold a table. Because of this, it increases storage overhead for the table, takes more time to read/write, and decreases the cache performance.

Using VACUUM, SQLite first copies data within a database file to a temporary database. This operation defragments the database objects, ignores the free spaces, and repacks individual pages. Then, the content of the temporary database file will be copied back to the original database file and the original database file is overwritten.

VACUUM command rebuilds the database, therefore it can be used to change some database-specific configuration parameters such as page size, page format, and default encoding. To do this, new values using pragma should be used and then vacuum the database.

The VACUUM command does not change the content of the database except the [rowid](https://www.sqlitetutorial.net/sqlite-autoincrement/) values. If you use INTEGER PRIMARY KEY column, the VACUUM does not change the values of that column. If unaliased rowed is used (Q?), the VACUUM command will reset the rowid values. Besides changing the rowid values, the VACUUM command also builds the index from scratch.

#remark: VACCUM command requires storage to hold the original file and also the copy. Also, it requires exclusive access to the database file. Thus it will not run successfully if the database has a pending SQL statement or an open transaction.

#remark: As of version 3.9.2, you can run the VACUUM command on the main database, not the attached database file.

The auto-vacuum mode is available and it triggers the vacuum process automatically with some limitations.

#full auto-vacuum mode:

PRAGMA auto\_vacuum = FULL;

#incremental vacuum:

PRAGMA auto\_vacuum = INCREMENTAL;

# auto-vacuum mode:

PRAGMA auto\_vacuum = NONE;

VACUUM with an INTO clause

# The VACUUM statement with an INTO clause keeps the original database file unchanged and creates a new database with the file name specified. The new database will contain the same logical content as the original database, but fully vacuumed.

VACUUM schema-name INTO filename;

The filename in the INTO clause can be any SQL expression that evaluates to a string. It must be a path to a file that does not exist or to an empty file, otherwise the VACUUM INTO command will result in an error.

The VACUUM command is very useful for generating backup copies of a live database. It is transactional safe, which the generated database is a consistent snapshot of the original database. However, if an unplanned shutdown or power lose interupts the command, the generated database might be corrupted.

#eg.

VACUUM main INTO 'c:\sqlite\db\chinook\_backup.db';

#####################################################################################

#TRANSACTION:

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.

* **A**tomic: 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.
* **C**onsistent: 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.
* **I**solation: a pending transaction performed by a session must be isolated from other sessions. When a session starts a transaction and executes the [INSERT](https://www.sqlitetutorial.net/sqlite-insert/) or [UPDATE](https://www.sqlitetutorial.net/sqlite-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.

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

BEGIN TRANSACTION;

#the statements of the transaction should go here

COMMIT

#or

ROLLBACK

#remark: COMMIT TRANSACTION or ROLLBACK TRANSACTION work in the same way as COMMIT and ROLLBACK.

#####################################################################################

#full text search:

CREATE VIRTUAL TABLE table\_name

USING FTS5(column1,column2...);

# types, constraints, or [PRIMARY KEY](https://www.sqlitetutorial.net/sqlite-primary-key/) declarationcan not be added in the CREATE VIRTUAL TABLE statement for creating an FTS5 table. If you do so, SQLite will issue an error. Like creating a normal table without specifying the [primary key column](https://www.sqlitetutorial.net/sqlite-primary-key/), SQLite adds an implicit rowid column to the FTS5 table.

Similar to a normal table, [insert data](https://www.sqlitetutorial.net/sqlite-insert/) can be done for virtual table.

After making a virtual table, the full text search FTS5 is possible using a MATCH operator in the [WHERE](https://www.sqlitetutorial.net/sqlite-where/) clause of the [SELECT](https://www.sqlitetutorial.net/sqlite-select/) statement. MATCH, =, or a tabled-value function can be used within WHERE:

#eg. Search for text program in the virtual table posts:

SELECT \*

FROM posts

WHERE posts MATCH 'program';

#or

WHERE posts='program';

#or

WHERE posts('program');

ORDER BY rank;

#remark: NOT, OR, or AND, or \* can be used in the text (within “”) to address more specifically what should be search within FTS5 mode in the virtual table. eg.

SELECT \*

FROM posts

WHERE posts MATCH 'learn NOT text';

WHERE posts MATCH 'learn OR text';

#or

WHERE posts MATCH 'learn AND text';

#or

WHERE posts MATCH 'learn AND text NOT PROGRAM';

## #Q? Built-in auxiliary functions

SQLite provides three built-in auxiliary functions that can be used within full-text queries on the FTS5 table.

* The bm25() returns a value that represents the accuracy of the current match, the lower value means a better match.
* The highlight() auxiliary function returns a copy of the text with search terms surrounded by a specified markup e.g.,<b>search term </b>
* The snippet() selects a short fragment of text in order to maximize the number of search terms it contains.

For example, the following query uses the highlight() function to decorate the search terms using the <b>tag</b>:

SELECT highlight(posts,0, '<b>', '</b>') title,

highlight(posts,1, '<b>', '</b>') body

FROM posts

WHERE posts MATCH 'SQLite'

ORDER BY rank;

#####################################################################################