This page highlights some of the characteristics of SQLite that are unusual and which make SQLite different from many other SQL database engines.

Zero-Configuration

SQLite does not need to be "installed" before it is used. There is no "setup" procedure. There is no server process that needs to be started, stopped, or configured. There is no need for an administrator to create a new database instance or assign access permissions to users. SQLite uses no configuration files. Nothing needs to be done to tell the system that SQLite is running. No actions are required to recover after a system crash or power failure. There is nothing to troubleshoot.

SQLite just works.

Other more familiar database engines run great once you get them going. But doing the initial installation and configuration can be intimidatingly complex.

Serverless

Most SQL database engines are implemented as a separate server process. Programs that want to access the database communicate with the server using some kind of interprocess communication (typically TCP/IP) to send requests to the server and to receive back results. SQLite does not work this way. With SQLite, the process that wants to access the database reads and writes directly from the database files on disk. There is no intermediary server process.

There are advantages and disadvantages to being serverless. The main advantage is that there is no separate server process to install, setup, configure, initialize, manage, and troubleshoot. This is one reason why SQLite is a "zero-configuration" database engine. Programs that use SQLite require no administrative support for setting up the database engine before they are run. Any program that is able to access the disk is able to use an SQLite database.

On the other hand, a database engine that uses a server can provide better protection from bugs in the client application - stray pointers in a client cannot corrupt memory on the server. And because a server is a single persistent process, it is able control database access with more precision, allowing for finer grain locking and better concurrency.

Most SQL database engines are client/server based. Of those that are serverless, SQLite is the only one that this author knows of that allows multiple applications to access the same database at the same time.

Single Database File

An SQLite database is a single ordinary disk file that can be located anywhere in the directory hierarchy. If SQLite can read the disk file then it can read anything in the database. If the disk file and its directory are writable, then SQLite can change anything in the database. Database files can easily be copied onto a USB memory stick or emailed for sharing.

Other SQL database engines tend to store data as a large collection of files. Often these files are in a standard location that only the database engine itself can access. This makes the data more secure, but also makes it harder to access. Some SQL database engines provide the option of writing directly to disk and bypassing the filesystem all together. This provides added performance, but at the cost of considerable setup and maintenance complexity.

Stable Cross-Platform Database File

The SQLite file format is cross-platform. A database file written on one machine can be copied to and used on a different machine with a different architecture. Big-endian or little-endian, 32-bit or 64-bit does not matter. All machines use the same file format. Furthermore, the developers have pledged to keep the file format stable and backwards compatible, so newer versions of SQLite can read and write older database files.

Most other SQL database engines require you to dump and restore the database when moving from one platform to another and often when upgrading to a newer version of the software.

Compact

When optimized for size, the whole SQLite library with everything enabled is less than 500KiB in size (as measured on an ix86 using the "size" utility from the GNU compiler suite.) Unneeded features can be disabled at compile-time to further reduce the size of the library to under 300KiB if desired.

Most other SQL database engines are much larger than this. IBM boasts that its recently released CloudScape database engine is "only" a 2MiB jar file - an order of magnitude larger than SQLite even after it is compressed! Firebird boasts that its client-side library is only 350KiB. That's as big as SQLite and does not even contain the database engine. The Berkeley DB library from Oracle is 450KiB and it omits SQL support, providing the programmer with only simple key/value pairs.

Manifest typing

Most SQL database engines use static typing. A datatype is associated with each column in a table and only values of that particular datatype are allowed to be stored in that column. SQLite relaxes this restriction by using manifest typing. In manifest typing, the datatype is a property of the value itself, not of the column in which the value is stored. SQLite thus allows the user to store any value of any datatype into any column regardless of the declared type of that column. (There are some exceptions to this rule: An INTEGER PRIMARY KEY column may only store integers. And SQLite attempts to coerce values into the declared datatype of the column when it can.)

As far as we can tell, the SQL language specification allows the use of manifest typing. Nevertheless, most other SQL database engines are statically typed and so some people feel that the use of manifest typing is a bug in SQLite. But the authors of SQLite feel very strongly that this is a feature. The use of manifest typing in SQLite is a deliberate design decision which has proven in practice to make SQLite more reliable and easier to use, especially when used in combination with dynamically typed programming languages such as Tcl and Python.

Variable-length records

Most other SQL database engines allocated a fixed amount of disk space for each row in most tables. They play special tricks for handling BLOBs and CLOBs which can be of wildly varying length. But for most tables, if you declare a column to be a VARCHAR(100) then the database engine will allocate 100 bytes of disk space regardless of how much information you actually store in that column.

SQLite, in contrast, use only the amount of disk space actually needed to store the information in a row. If you store a single character in a VARCHAR(100) column, then only a single byte of disk space is consumed. (Actually two bytes - there is some overhead at the beginning of each column to record its datatype and length.)

The use of variable-length records by SQLite has a number of advantages. It results in smaller database files, obviously. It also makes the database run faster, since there is less information to move to and from disk. And, the use of variable-length records makes it possible for SQLite to employ manifest typing instead of static typing.

Readable source code

The source code to SQLite is designed to be readable and accessible to the average programmer. All procedures and data structures and many automatic variables are carefully commented with useful information about what they do. Boilerplate commenting is omitted.

SQL statements compile into virtual machine code

Every SQL database engine compiles each SQL statement into some kind of internal data structure which is then used to carry out the work of the statement. But in most SQL engines that internal data structure is a complex web of interlinked structures and objects. In SQLite, the compiled form of statements is a short program in a machine-language like representation. Users of the database can view this virtual machine language by prepending the EXPLAIN keyword to a query.

The use of a virtual machine in SQLite has been a great benefit to the library's development. The virtual machine provides a crisp, well-defined junction between the front-end of SQLite (the part that parses SQL statements and generates virtual machine code) and the back-end (the part that executes the virtual machine code and computes a result.) The virtual machine allows the developers to see clearly and in an easily readable form what SQLite is trying to do with each statement it compiles, which is a tremendous help in debugging. Depending on how it is compiled, SQLite also has the capability of tracing the execution of the virtual machine - printing each virtual machine instruction and its result as it executes.

Public domain

The source code for SQLite is in the public domain. No claim of copyright is made on any part of the core source code. (The documentation and test code is a different matter - some sections of documentation and test logic are governed by open-source licenses.) All contributors to the SQLite core software have signed affidavits specifically disavowing any copyright interest in the code. This means that anybody is able to legally do anything they want with the SQLite source code.

There are other SQL database engines with liberal licenses that allow the code to be broadly and freely used. But those other engines are still governed by copyright law. SQLite is different in that copyright law simply does not apply.

The source code files for other SQL database engines typically begin with a comment describing your legal rights to view and copy that file. The SQLite source code contains no license since it is not governed by copyright. Instead of a license, the SQLite source code offers a blessing:

May you do good and not evil

May you find forgiveness for yourself and forgive others

May you share freely, never taking more than you give.

SQL language extensions

SQLite provides a number of enhancements to the SQL language not normally found in other database engines. The EXPLAIN keyword and manifest typing have already been mentioned above. SQLite also provides statements such as REPLACE and the ON CONFLICT clause that allow for added control over the resolution of constraint conflicts. SQLite supports ATTACH and DETACH commands that allow multiple independent databases to be used together in the same query. And SQLite defines APIs that allows the user to add new SQL functions and collating sequences.

**Syntax**

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

SELECT

name,

milliseconds,

albumid

FROM

tracks

ORDER BY

3,2;

i.e. order by third column and second one respectively

* 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 returns one million rows. However, if you just need the first 10 rows in the result set, you add the LIMIT clause to the SELECT statement to get exact 10 rows.

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;

1

2

3

4

5

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;

1

2

3

4

5

SELECT

column\_list

FROM

table

LIMIT offset, row\_count;

----------------------------------------------------------

Home / SQLite IN

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 subquery.

Introduction to the SQLite IN operator

The SQLite IN operator is used to determine 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);

1

expression [NOT] IN (value\_list|subquery);

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

A list of values is a fixed value list or a result set of one column returned by a subquery. The returned type of the expression and values in the list must be the same.

The IN operator returns true or false depending 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.

----------------------------------------------------------

ntroduction 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 character.

The following shows the GLOB wildcards:

The asterisk (\*) wildcard matches any number of characters.

The question mark (?) wildcard matches exact one character.

In addition, 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.

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

<http://www.sqlitetutorial.net/sqlite-glob/>

----------------------------------------------------------

SQLite **PRAGMA** command is a special command to be used to control various environmental variables and state flags within the SQLite environment. A PRAGMA value can be read and it can also be set based on the requirements.

<https://www.tutorialspoint.com/sqlite/sqlite_pragma.htm>

In SQLite the primary key may be null in contrary to other databases

**Dropping Constraint**

SQLite supports a limited subset of ALTER TABLE. The ALTER TABLE command in SQLite allows the user to rename a table or add a new column to an existing table. It is not possible to rename a column, remove a column, or add or remove constraints from a table.

**ISNULL() And IFNULL()**

<https://www.w3schools.com/sql/sql_isnull.asp>

http://www.sqlitetutorial.net/sqlite-functions/sqlite-ifnull/

**Triggers**

<https://www.tutorialspoint.com/sqlite/sqlite_triggers.htm>

**Indexes**

<https://www.tutorialspoint.com/sqlite/sqlite_indexes.htm>

**Comparison Between SQL and SQLite**

|  |  |
| --- | --- |
| SQL | SQLite |
| To retrieve a specific number of rows uses Top | Uses LIMIT Clause |
| SQL supports two wildcard operators in conjunction with the LIKE operator which are explained in detail in the following table(%) and(\_) | GLOB operator is used to match only text values against a pattern using wildcards |
| DOESNOT Exist | SQLite PRAGMA command is a special command to be used to control various environmental variables and state flags within the SQLite environment. A PRAGMA value can be read and it can also be set based on the requirements. |
| CANNOT be NULL | Due to a 'longstanding coding oversight', primary keys can be NULL in SQLite. This is not the case with other databases. |
| * [INNER JOIN](https://www.tutorialspoint.com/sql/sql-inner-joins.htm) * [LEFT JOIN](https://www.tutorialspoint.com/sql/sql-left-joins.htm) * [RIGHT JOIN](https://www.tutorialspoint.com/sql/sql-right-joins.htm) * [FULL JOIN](https://www.tutorialspoint.com/sql/sql-full-joins.htm) * [SELF JOIN](https://www.tutorialspoint.com/sql/sql-self-joins.htm) * [CARTESIAN(Cross) JOIN](https://www.tutorialspoint.com/sql/sql-cartesian-joins.htm) | * The CROSS JOIN * The INNER JOIN * The OUTER JOIN |
| The SQL TRUNCATE TABLE command is used to delete complete data from an existing table. | We do not have TRUNCATE TABLE command in SQLite but you can use SQLite **DELETE**command to delete complete data from an existing table, though it is recommended to use DROP TABLE command to drop the complete table and re-create it once again. If you are using DELETE TABLE command to delete all the records, it is recommended to use **VACUUM** command to clear unused space. |
| CREATE VIEW view\_name AS  SELECT column1, column2.....  FROM table\_name  WHERE [condition]; | Views  CREATE [TEMP | TEMPORARY] VIEW view\_name AS  SELECT column1, column2.....  FROM table\_name  WHERE [condition];  WITH CHECK is not supported(not sure) |
| IFNULL(parameter\_1,parameter\_2); | SELECT ProductName, UnitPrice \* (UnitsInStock + ISNULL(UnitsOnOrder, 0)) FROM Products |

**abs(*X*)**

The abs(X) function returns the absolute value of the numeric argument X. Abs(X) returns NULL if X is NULL. Abs(X) returns 0.0 if X is a string or blob that cannot be converted to a numeric value. If X is the integer -9223372036854775808 then abs(X) throws an integer overflow error since there is no equivalent positive 64-bit two complement value.

**changes()**

The changes() function returns the number of database rows that were changed or inserted or deleted by the most recently completed INSERT, DELETE, or UPDATE statement, exclusive of statements in lower-level triggers. The changes() SQL function is a wrapper around the [sqlite3\_changes()](https://sqlite.org/c3ref/changes.html) C/C++ function and hence follows the same rules for counting changes.

**char(*X1*,*X2*,...,*XN*)**

The char(X1,X2,...,XN) function returns a string composed of characters having the unicode code point values of integers X1 through XN, respectively.

**coalesce(*X*,*Y*,...)**

The coalesce() function returns a copy of its first non-NULL argument, or NULL if all arguments are NULL. Coalesce() must have at least 2 arguments.

**glob(*X*,*Y*)**

The glob(X,Y) function is equivalent to the expression "**Y GLOB X**". Note that the X and Y arguments are reversed in the glob() function relative to the infix [GLOB](https://sqlite.org/lang_expr.html#glob) operator. If the [sqlite3\_create\_function()](https://sqlite.org/c3ref/create_function.html) interface is used to override the glob(X,Y) function with an alternative implementation then the [GLOB](https://sqlite.org/lang_expr.html#glob) operator will invoke the alternative implementation.

**hex(*X*)**

The hex() function interprets its argument as a BLOB and returns a string which is the upper-case hexadecimal rendering of the content of that blob.

**ifnull(*X*,*Y*)**

The ifnull() function returns a copy of its first non-NULL argument, or NULL if both arguments are NULL. Ifnull() must have exactly 2 arguments. The ifnull() function is equivalent to [coalesce()](https://sqlite.org/lang_corefunc.html#coalesce) with two arguments.

**instr(*X*,*Y*)**

The instr(X,Y) function finds the first occurrence of string Y within string X and returns the number of prior characters plus 1, or 0 if Y is nowhere found within X. Or, if X and Y are both BLOBs, then instr(X,Y) returns one more than the number bytes prior to the first occurrence of Y, or 0 if Y does not occur anywhere within X. If both arguments X and Y to instr(X,Y) are non-NULL and are not BLOBs then both are interpreted as strings. If either X or Y are NULL in instr(X,Y) then the result is NULL.

**last\_insert\_rowid()**

The last\_insert\_rowid() function returns the [ROWID](https://sqlite.org/lang_createtable.html#rowid) of the last row insert from the database connection which invoked the function. The last\_insert\_rowid() SQL function is a wrapper around the [sqlite3\_last\_insert\_rowid()](https://sqlite.org/c3ref/last_insert_rowid.html) C/C++ interface function.

**length(*X*)**

For a string value X, the length(X) function returns the number of characters (not bytes) in X prior to the first NUL character. Since SQLite strings do not normally contain NUL characters, the length(X) function will usually return the total number of characters in the string X. For a blob value X, length(X) returns the number of bytes in the blob. If X is NULL then length(X) is NULL. If X is numeric then length(X) returns the length of a string representation of X.

**like(*X*,*Y*)  
like(*X*,*Y*,*Z*)**

The like() function is used to implement the "**Y LIKE X [ESCAPE Z]**" expression. If the optional ESCAPE clause is present, then the like() function is invoked with three arguments. Otherwise, it is invoked with two arguments only. Note that the X and Y parameters are reversed in the like() function relative to the infix [LIKE](https://sqlite.org/lang_expr.html#like) operator. The [sqlite3\_create\_function()](https://sqlite.org/c3ref/create_function.html) interface can be used to override the like() function and thereby change the operation of the [LIKE](https://sqlite.org/lang_expr.html#like) operator. When overriding the like() function, it may be important to override both the two and three argument versions of the like() function. Otherwise, different code may be called to implement the [LIKE](https://sqlite.org/lang_expr.html#like) operator depending on whether or not an ESCAPE clause was specified.

**likelihood(*X*,*Y*)**

The likelihood(X,Y) function returns argument X unchanged. The value Y in likelihood(X,Y) must be a floating point constant between 0.0 and 1.0, inclusive. The likelihood(X) function is a no-op that the code generator optimizes away so that it consumes no CPU cycles during run-time (that is, during calls to [sqlite3\_step()](https://sqlite.org/c3ref/step.html)). The purpose of the likelihood(X,Y) function is to provide a hint to the query planner that the argument X is a boolean that is true with a probability of approximately Y. The [unlikely(X)](https://sqlite.org/lang_corefunc.html#unlikely) function is short-hand for likelihood(X,0.0625). The [likely(X)](https://sqlite.org/lang_corefunc.html#likely) function is short-hand for likelihood(X,0.9375).

**likely(*X*)**

The likely(X) function returns the argument X unchanged. The likely(X) function is a no-op that the code generator optimizes away so that it consumes no CPU cycles at run-time (that is, during calls to [sqlite3\_step()](https://sqlite.org/c3ref/step.html)). The purpose of the likely(X) function is to provide a hint to the query planner that the argument X is a boolean value that is usually true. The likely(X) function is equivalent to [likelihood](https://sqlite.org/lang_corefunc.html#likelihood)(X,0.9375). See also: [unlikely(X)](https://sqlite.org/lang_corefunc.html#unlikely).

**load\_extension(*X*)  
load\_extension(*X*,*Y*)**

The load\_extension(X,Y) function loads [SQLite extensions](https://sqlite.org/loadext.html) out of the shared library file named X using the entry point Y. The result of load\_extension() is always a NULL. If Y is omitted then the default entry point name is used. The load\_extension() function raises an exception if the extension fails to load or initialize correctly.

The load\_extension() function will fail if the extension attempts to modify or delete an SQL function or collating sequence. The extension can add new functions or collating sequences, but cannot modify or delete existing functions or collating sequences because those functions and/or collating sequences might be used elsewhere in the currently running SQL statement. To load an extension that changes or deletes functions or collating sequences, use the[sqlite3\_load\_extension()](https://sqlite.org/c3ref/load_extension.html) C-language API.

For security reasons, extension loaded is turned off by default and must be enabled by a prior call to [sqlite3\_enable\_load\_extension()](https://sqlite.org/c3ref/enable_load_extension.html).

**lower(*X*)**

The lower(X) function returns a copy of string X with all ASCII characters converted to lower case. The default built-in lower() function works for ASCII characters only. To do case conversions on non-ASCII characters, load the ICU extension.

**ltrim(*X*)  
ltrim(*X*,*Y*)**

The ltrim(X,Y) function returns a string formed by removing any and all characters that appear in Y from the left side of X. If the Y argument is omitted, ltrim(X) removes spaces from the left side of X.

**max(*X*,*Y*,...)**

The multi-argument max() function returns the argument with the maximum value, or return NULL if any argument is NULL. The multi-argument max() function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If none of the arguments to max() define a collating function, then the BINARY collating function is used. Note that **max()** is a simple function when it has 2 or more arguments but operates as an [aggregate function](https://sqlite.org/lang_aggfunc.html#maxggunc) if given only a single argument.

**min(*X*,*Y*,...)**

The multi-argument min() function returns the argument with the minimum value. The multi-argument min() function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If none of the arguments to min() define a collating function, then the BINARY collating function is used. Note that **min()** is a simple function when it has 2 or more arguments but operates as an [aggregate function](https://sqlite.org/lang_aggfunc.html#minggunc) if given only a single argument.

**nullif(*X*,*Y*)**

The nullif(X,Y) function returns its first argument if the arguments are different and NULL if the arguments are the same. The nullif(X,Y) function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If neither argument to nullif() defines a collating function then the BINARY is used.

**printf(*FORMAT*,...)**

The printf(FORMAT,...) SQL function works like the [sqlite3\_mprintf()](https://sqlite.org/c3ref/mprintf.html) C-language function and the printf() function from the standard C library. The first argument is a format string that specifies how to construct the output string using values taken from subsequent arguments. If the FORMAT argument is missing or NULL then the result is NULL. The %n format is silently ignored and does not consume an argument. The %p format is an alias for %X. The %z format is interchangeable with %s. If there are too few arguments in the argument list, missing arguments are assumed to have a NULL value, which is translated into 0 or 0.0 for numeric formats or an empty string for %s.

**quote(*X*)**

The quote(X) function returns the text of an SQL literal which is the value of its argument suitable for inclusion into an SQL statement. Strings are surrounded by single-quotes with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. Strings with embedded NUL characters cannot be represented as string literals in SQL and hence the returned string literal is truncated prior to the first NUL.

**random()**

The random() function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807.

**randomblob(*N*)**

The randomblob(N) function return an N-byte blob containing pseudo-random bytes. If N is less than 1 then a 1-byte random blob is returned.

Hint: applications can generate globally unique identifiers using this function together with [hex()](https://sqlite.org/lang_corefunc.html#hex) and/or [lower()](https://sqlite.org/lang_corefunc.html#lower) like this:

hex(randomblob(16))  
  
lower(hex(randomblob(16)))

**replace(*X*,*Y*,*Z*)**

The replace(X,Y,Z) function returns a string formed by substituting string Z for every occurrence of string Y in string X. The [BINARY](https://sqlite.org/datatype3.html#collation) collating sequence is used for comparisons. If Y is an empty string then return X unchanged. If Z is not initially a string, it is cast to a UTF-8 string prior to processing.

**round(*X*)  
round(*X*,*Y*)**

The round(X,Y) function returns a floating-point value X rounded to Y digits to the right of the decimal point. If the Y argument is omitted, it is assumed to be 0.

**rtrim(*X*)  
rtrim(*X*,*Y*)**

The rtrim(X,Y) function returns a string formed by removing any and all characters that appear in Y from the right side of X. If the Y argument is omitted, rtrim(X) removes spaces from the right side of X.

**soundex(*X*)**

The soundex(X) function returns a string that is the soundex encoding of the string X. The string "?000" is returned if the argument is NULL or contains no ASCII alphabetic characters. This function is omitted from SQLite by default. It is only available if the [SQLITE\_SOUNDEX](https://sqlite.org/compile.html#soundex) compile-time option is used when SQLite is built.

**sqlite\_compileoption\_get(*N*)**

The sqlite\_compileoption\_get() SQL function is a wrapper around the [sqlite3\_compileoption\_get()](https://sqlite.org/c3ref/compileoption_get.html) C/C++ function. This routine returns the N-th compile-time option used to build SQLite or NULL if N is out of range. See also the [compile\_options pragma](https://sqlite.org/pragma.html#pragma_compile_options).

**sqlite\_compileoption\_used(*X*)**

The sqlite\_compileoption\_used() SQL function is a wrapper around the [sqlite3\_compileoption\_used()](https://sqlite.org/c3ref/compileoption_get.html) C/C++ function. When the argument X to sqlite\_compileoption\_used(X) is a string which is the name of a compile-time option, this routine returns true (1) or false (0) depending on whether or not that option was used during the build.

**sqlite\_source\_id()**

The sqlite\_source\_id() function returns a string that identifies the specific version of the source code that was used to build the SQLite library. The string returned by sqlite\_source\_id() is the date and time that the source code was checked in followed by the SHA1 hash for that check-in. This function is an SQL wrapper around the [sqlite3\_sourceid()](https://sqlite.org/c3ref/libversion.html) C interface.

**sqlite\_version()**

The sqlite\_version() function returns the version string for the SQLite library that is running. This function is an SQL wrapper around the [sqlite3\_libversion()](https://sqlite.org/c3ref/libversion.html) C-interface.

**substr(*X*,*Y*,*Z*)  
substr(*X*,*Y*)**

The substr(X,Y,Z) function returns a substring of input string X that begins with the Y-th character and which is Z characters long. If Z is omitted then substr(X,Y) returns all characters through the end of the string X beginning with the Y-th. The left-most character of X is number 1. If Y is negative then the first character of the substring is found by counting from the right rather than the left. If Z is negative then the abs(Z) characters preceding the Y-th character are returned. If X is a string then characters indices refer to actual UTF-8 characters. If X is a BLOB then the indices refer to bytes.

**total\_changes()**

The total\_changes() function returns the number of row changes caused by INSERT, UPDATE or DELETE statements since the current database connection was opened. This function is a wrapper around the [sqlite3\_total\_changes()](https://sqlite.org/c3ref/total_changes.html) C/C++ interface.

**trim(*X*)  
trim(*X*,*Y*)**

The trim(X,Y) function returns a string formed by removing any and all characters that appear in Y from both ends of X. If the Y argument is omitted, trim(X) removes spaces from both ends of X.

**typeof(*X*)**

The typeof(X) function returns a string that indicates the [datatype](https://sqlite.org/datatype3.html) of the expression X: "null", "integer", "real", "text", or "blob".

**unicode(*X*)**

The unicode(X) function returns the numeric unicode code point corresponding to the first character of the string X. If the argument to unicode(X) is not a string then the result is undefined.

**unlikely(*X*)**

The unlikely(X) function returns the argument X unchanged. The unlikely(X) function is a no-op that the code generator optimizes away so that it consumes no CPU cycles at run-time (that is, during calls to [sqlite3\_step()](https://sqlite.org/c3ref/step.html)). The purpose of the unlikely(X) function is to provide a hint to the query planner that the argument X is a boolean value that is usually not true. The unlikely(X) function is equivalent to [likelihood](https://sqlite.org/lang_corefunc.html#likelihood)(X, 0.0625).

**upper(*X*)**

The upper(X) function returns a copy of input string X in which all lower-case ASCII characters are converted to their upper-case equivalent.

**zeroblob(*N*)**

The zeroblob(N) function returns a BLOB consisting of N bytes of 0x00. SQLite manages these zeroblobs very efficiently. Zeroblobs can be used to reserve space for a BLOB that is later written using [incremental BLOB I/O](https://sqlite.org/c3ref/blob_open.html). This SQL function is implemented using the [sqlite3\_result\_zeroblob()](https://sqlite.org/c3ref/result_blob.html) routine from the C/C++ interface.