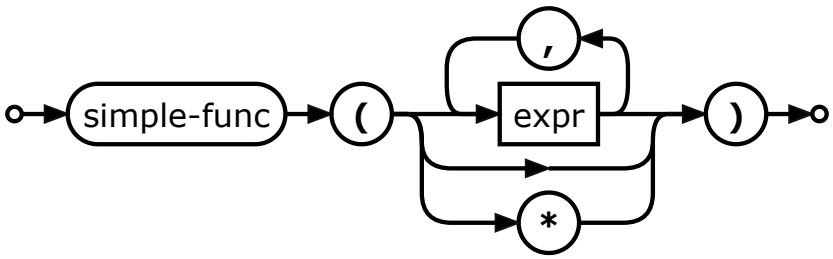
Built-In Scalar SQL Functions

1. Overview

The core functions shown below are available by default. Date & Time functions, aggregate functions, window functions, math functions, and JSON functions are documented separately. An application may define additional functions written in C and added to the database engine using the sqlite3_create_function() API.

simple-function-invocation:



See the functions within expressions documentation for more information about how SQL function invocations fit into the context of an SQL expression.

2. List Of Core Functions

abs(X)
changes()
char(X1,X2,...,XN)
coalesce(X,Y,...)
concat(X,...)
concat_ws(SEP,X,...)
format(FORMAT,...)
glob(X,Y)
hex(X)
if(B1,V1,...)
ifnull(X,Y)
iif(B1,V1,...)
instr(X,Y)
last_insert_rowid()
length(X)
like(X,Y)
like(X,Y,Z)
likelihood(X,Y)
likely(X)
load_extension(X)
load_extension(X,Y)
lower(X)
ltrim(X)
ltrim(X,Y)
max(X,Y,...)
min(X,Y,...)
nullif(X,Y)
octet_length(X)
printf(FORMAT,...)
quote(X)

3. Descriptions of built-in scalar SQL functions

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_changes64() 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.

concat(*X*,...)

The concat(...) function returns a string which is the concatenation of the string representation of all of its non-NULL arguments. If all arguments are NULL, then concat() returns an empty string.

concat_ws(*SEP*,*X*,...)

The concat_ws(SEP,...) function returns a string that is the concatenation of all non-null arguments beyond the first argument, using the text value of the first argument as a separator. If the first argument is NULL, then concat_ws() returns NULL. If all arguments other than the first are NULL, then concat_ws() returns an empty string.

format(*FORMAT*,...)

The format(FORMAT,...) SQL function works like the sqlite3_mprintf() 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. See the built-in printf() documentation for additional information.

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 operator. Y is the string and X is the pattern. So, for example, the following expressions are equivalent:

```
name GLOB '*helium*'
glob('*helium*',name)
```

If the sqlite3_create_function() interface is used to override the glob(X,Y) function with an alternative implementation then the 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.

If the argument *X* in "hex(*X*)" is an integer or floating point number, then "interprets its argument as a BLOB" means that the binary number is first converted into a UTF8 text representation, then that text is interpreted as a BLOB. Hence, "hex(12345678)" renders as "3132333435363738" not the binary representation of the integer value "0000000000BC614E".

See also: unhex()

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() with two arguments.

iif(*B1*,*V1*,...)
if(*B1*,*V1*,...)

The iif(B1,V1,...,BN,VN) function takes arguments in pairs. The first argument of each pair is a Boolean and the second argument is a value to return if the Boolean is true. The iif() function returns the value associated with the first true Boolean. If the number of arguments to iif() is odd, then the last argument is a value that returned if all prior Boolean arguments are false. If the number of arguments is even and all Boolean arguments are false, then NULL is returned. The iif() function requires at least two arguments. The iif() function is really a short-hand notation for a CASE expression. For example, the iif(X,Y,Z) function is logically equivalent to and generates the same bytecode as the CASE expression "CASE WHEN X THEN Y ELSE Z END". The if() function is just an alternative spelling for iif().

The iif() function uses short-circuit evaluation. Arguments are only evaluated if necessary to compute the final result. So, for example, if one of the value arguments involves an expensive computation (such as an elaborate subquery) but the corresponding Boolean is false, the expensive computation never occurs. Similarly, Boolean arguments past the first one that is true are never evaluated.

The iif() function originally required exactly three arguments. The two-argument version of iif() and the ability to spell the function as "if()" where features added in SQLite version 3.48.0 (2025-01-14) The ability to accept more than 3 arguments was added in SQLite version 3.49.0 (2025-02-06).

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 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() C/C++ interface function.

length(*X*)

For a string value X, the length(X) function returns the number of Unicode code points (not bytes) in input string X prior to the first U+0000 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.

Note that for strings, the length(X) function returns the *character* or *code-point* length of the string, not the byte length. The character length is the number of

characters in the string. The character length is always different from the byte length for UTF-16 strings, and can be different from the byte length for UTF-8 strings if the string contains multi-byte characters. Use the octet_length() function to find the byte length of a string.

For BLOB values, length(X) always returns the byte-length of the BLOB.

For string values, length(X) must read the entire string into memory in order to compute the character length. But for BLOB values, reading the whole string into memory is not necessary as SQLite already knows how many bytes are in the BLOB. Hence, for multi-megabyte values, the length(X) function is usually much faster for BLOBs than for strings, since it does not need to load the value into memory.

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 operator. X is the pattern and Y is the string to match against that pattern. Hence, the following expressions are equivalent:

```
name LIKE '%neon%'
like('%neon%',name)
```

The sqlite3_create_function() interface can be used to override the like() function and thereby change the operation of the 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 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()). 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) function is short-hand for likelihood(X,0.0625). The likely(X) 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()). 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(X,0.9375). See also: unlikely(X).

load_extension(*X*)
load_extension(*X*,*Y*)

The load_extension(X,Y) function loads SQLite extensions 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() C-language API.

For security reasons, extension loading is disabled by default and must be enabled by a prior call to sqlite3_enable_load_extension().

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 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 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 collating function is used.

octet_length(*X*)

The octet_length(X) function returns the number of bytes in the encoding of text string X. If X is NULL then octet_length(X) returns NULL. If X is a BLOB value, then octet_length(X) is the same as length(X). If X is a numeric value, then octet_length(X) returns the number of bytes in a text rendering of that number.

Because octet_length(X) returns the number of bytes in X, not the number of characters or code-points, the value returned depends on the database encoding. The octet_length() function can return different answers for the same input string if the database encoding is UTF16 instead of UTF8.

If argument X is a table column and the value is of type text or blob, then octet_length(X) avoids reading the content of X from disk, as the byte length can be computed from metadata. Thus, octet_length(X) is efficient even if X is a column containing a multi-megabyte text or blob value.

printf(*FORMAT*,...)

The printf() SQL function is an alias for the format() SQL function. The format() SQL function was originally named printf(). But the name was later changed to format() for compatibility with other database engines. The printf() name is retained as an alias so as not to break legacy code.

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() and/or 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 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 or negative, it is taken 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.

sign(*X*)

The sign(X) function returns -1, 0, or +1 if the argument X is a numeric value that is negative, zero, or positive, respectively. If the argument to sign(X) is NULL or is a string or blob that cannot be losslessly converted into a number, then sign(X) returns NULL.

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 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() 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.

sqlite_compileoption_used(*X*)

The sqlite_compileoption_used() SQL function is a wrapper around the sqlite3_compileoption_used() 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_offset(*X*)

The sqlite_offset(X) function returns the byte offset in the database file for the beginning of the record from which value would be read. If X is not a column in an ordinary table, then sqlite_offset(X) returns NULL. The value returned by sqlite_offset(X) might reference either the original table or an index, depending on the query. If the value X would normally be extracted from an index, the sqlite_offset(X) returns the offset to the corresponding index record. If the value X would be extracted from the original table, then sqlite_offset(X) returns the offset to the table record.

The sqlite_offset(X) SQL function is only available if SQLite is built using the -DSQLITE_ENABLE_OFFSET_SQL_FUNC compile-time option.

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 SHA3-256 hash for that check-in. This function is an SQL wrapper around the sqlite3_sourceid() 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() C-interface.

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

"substring()" is an alias for "substr()" beginning with SQLite version 3.34.

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_changes64() 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 of the expression X: "null", "integer", "real", "text", or "blob".

unhex(*X*)
unhex(*X*,*Y*)

The unhex(X,Y) function returns a BLOB value which is the decoding of the hexadecimal string X. If X contains any characters that are not hexadecimal digits and which are not in Y, then unhex(X,Y) returns NULL. If Y is omitted, it is understood to be an empty string and hence X must be a pure hexadecimal string. All hexadecimal digits in X must occur in pairs, with both digits of each pair beginning immediately adjacent to one another, or else unhex(X,Y) returns NULL. If either parameter X or Y is NULL, then unhex(X,Y) returns NULL. The X input may contain an arbitrary mix of upper and lower case hexadecimal digits. Hexadecimal digits in Y have no affect on the translation of X. Only characters in Y that are not hexadecimal digits are ignored in X.

See also: hex()

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()). 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(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. This SQL function is implemented using the sqlite3_result_zeroblob() routine from the C/C++ interface.

This page last modified on 2025-02-06 11:58:24 UTC