APPENDIX A

# Operators and Functions

This appendix covers common operations and functions used in SQLite as a convenient reference. While this book does not cover all the functionalities of SQLite, the functionalities likely to have an immediate need can be found here. The primary goal is to showcase universal SQL concepts that apply to most platforms, not to teach the nuances of the SQLite platform in detail.

A comprehensive coverage of SQLite's features can be found at *https://www.sqlite.org/docs.html*.

## Appendix A1 – Literal Expression Queries

You can test operators and functions easily without querying any tables at all. You simply `SELECT` an expression of literals as in the following query, which will calculate a single value of 12:

```
SELECT 5 + 7
```

Any functions and literals, including text strings, can be tested in this manner as well. This query will check if the word `'TONY'` is in the string `'TONY STARK'`, and it should return `1`:

```
SELECT INSTR('TONY STARK', 'TONY')
```

This is a great way to test operators and functions without using any tables. This appendix will show many examples with this approach, and you can use it for your own experimentation.

# Appendix A2 – Mathematical Operators

SQLite has a small set of basic math operators. More advanced tasks are usually done with functions, but here are the five core mathematical operators.

*Assume x = 7 and y = 3*

| Operator | Description | Example | Result |
|---|---|---|---|
| + | Adds two numbers | `x + y` | 10 |
| - | Subtracts two numbers | `x - y` | 4 |
| * | Multiplies two numbers | `x * y` | 21 |
| / | Divides two numbers | `x / y` | 2 |
| % | Divides two numbers, but returns the remainder | `x % y` | 1 |

# Appendix A3 – Comparison Operators

Comparison operators yield a true (1) or false (0) value based on a comparative evaluation.

*Assume x = 5 and y = 10*

| Operator | Description | Example | Result |
|---|---|---|---|
| = and == | Checks if two values are equal | `x = y` | 0 (false) |
| != and <> | Checks if two values are not equal | `x != y` | 1 (true) |
| > | Checks if value on left is greater than value on right | `x > y` | 0 (false) |
| < | Checks if value on left is less than value on right | `x < y` | 1 (true) |
| >= | Checks if value on left is greater than or equal to value on right | `x >= y` | 0 (false) |
| <= | Checks if value on left is less than or equal to value on right | `x <= y` | 1 (true) |

# APPENDIX A4 – Logical Operators

Logical operators allow you combine Boolean expressions as well as perform more conditional operations.

*Assume x = true (1) and y = false (0)*

*Assume a = 4 and b = 10*

| Operator | Description | Example | Result |
|---|---|---|---|
| AND | Checks for all Boolean expressions to be true | `x AND y` | 0 (false) |
| OR | Checks for any Boolean expression to be true | `x OR y` | 1 (true) |
| BETWEEN | Checks if a value inclusively falls inside a range | `a BETWEEN 1 and b` | 1 (true) |

| Operator | Description | Example | Result |
|---|---|---|---|
| `IN` | Checks if a value is in a list of values | `a IN (1,5,6,7)` | 0 (false) |
| `NOT` | Negates and flips a Boolean expression's value | `a NOT IN (1,5,6,7)` | 1 (true) |
| `IS NULL` | Checks if a value is null | `a IS NULL` | 0 (false) |
| `IS NOT NULL` | Checks if a value is not null | `a IS NOT NULL` | 1 (true) |

# APPENDIX A5 – Text Operators

Text has a limited set of operators, as most text processing tasks are done with functions. There are a few, though. Keep in mind also that regular expressions are beyond the scope of this book, but they are worth studying if you ever work heavily with text patterns.

*Assume city = 'Dallas' and state = 'TX'*

| Operator | Description | Example | Result |
|---|---|---|---|
| `\|\|` | Concatenates one or more values together into text | `city \|\| ', ' \|\| state` | `Dallas, TX` |
| `LIKE` | Allows wildcards _ and % to look for text patterns | `state LIKE 'D_l%'` | 1 (true) |
| `REGEXP` | Matches a text pattern using a regular expression | `state REGEXP '[A-Z]{2}'` | 1 (true) |

A special note to programmers: `REGEXP` is not implemented out of the box for SQLite, so you may have to compile or implement it when using SQLite for your app or program.

# APPENDIX A6 – Common Core Functions

SQLite has many core functions built in. While this is not a comprehensive list, these are the most commonly used ones. A full list of functions and their documentation can be found at *http://www.sqlite.org/lang_corefunc.html*.

*Assume x = –5, y = 2, and z is NULL*

| Operator | Description | Example | Result |
|---|---|---|---|
| `abs()` | Calculates the absolute value of a number | `abs(x)` | 5 |
| `coalesce()` | Converts a possible null value into a default value if it is null | `coalesce(z, y)` | 2 |
| `instr()` | Checks if a text string contains another text string; if so it returns the index for the found position, and otherwise it returns 0 | `instr('HTX','TX')` | 2 |
| `length()` | Provides the number of characters in a string | `length('Dallas')` | 6 |
| `trim()` | Removes extraneous spaces on both sides of a string | `trim(' TX ')` | TX |

| Operator | Description | Example | Result |
|---|---|---|---|
| `ltrim()` | Removes extraneous spaces on the left side of a string | `ltrim(' TX')` | TX |
| `rtrim()` | Removes extraneous spaces on the right side of a string | `rtrim('LA ')` | LA |
| `random()` | Returns a pseudorandom number from the range –9223372036854775808 to +9223372036854775807 | `random()` | 7328249 |
| `round()` | Rounds a decimal to a specified number of decimal places | `round(182.245, 2)` | 182.24 |
| `replace()` | Replaces a substring of text in a string with another string | `replace('Tom Nield','Tom', 'Thomas')` | Thomas Nield |
| `substr()` | Extracts a range of characters from a string with their numeric positions | `substr('DOG',2,3)` | OG |
| `lower()` | Turns all letters in a string to lowercase | `lower('DoG')` | dog |
| `upper()` | Turns all letters in a string to uppercase | `upper('DoG')` | DOG |

# APPENDIX A7 – Aggregate Functions

SQLite has a set of aggregate functions you can use with a `GROUP BY` statement to get a scoped aggregation in some form.

*X = a column you specify the aggregation on*

| Function | Description |
|---|---|
| `avg(X)` | Calculates the average of all values in that column (omits null values). |
| `count(X)` | Counts the number of non-null values in that column. |
| `count(*)` | Counts the number of records. |
| `max(X)` | Calculates the maximum value in that column (omits null values). |
| `min(X)` | Calculates the minimum value in that column (omits null values). |
| `sum(X)` | Calculates the sum of the values in that column (omits null values). |
| `group_concat(X)` | Concatenates all non-null values in that column. You can also provide a second argument specifying a separator, like commas. |

# APPENDIX A8 – Date and Time Functions

Functionality for dates and times in SQL varies greatly between database platforms. Therefore, this book does not cover this topic outside this appendix. You will need to learn the date/time syntax for your specific database platform. Some platforms, such as MySQL, make working with date/time values very intuitive, while others can be less intuitive.

For SQLite, date and time functions cannot be comprehensively covered here as that would be beyond the scope of this book. But the most common date and time tasks will be covered in this section. Full documentation of SQLite date and time handling can be found at the SQLite website (*http://www.sqlite.org/lang_datefunc.html*).

## Date Functions

When working with dates, it is simplest to store them in the string format YYYY-MM-DD as most database platforms inherently understand this format (technically called the ISO8601 format). A four-digit year comes first, following by a two-digit month, and a two-digit day, each separated by a dash (e.g., 2015-06-17). If you format your date strings like this, you will never have to do any explicit conversions.

When running a query, any string in the `'YYYY-MM-DD'` date format will be interpreted as a date. This means you can do chronological tasks like comparing one date to another date:

```
SELECT '2015-05-14' > '2015-01-12'
```

If you do not use this ISO8601 format, SQLite will compare them as text strings and check if the first text comes before the second text alphabetically. This obviously is not desirable as you want dates to be evaluated, compared, and treated as dates.

Conveniently, you can get today's date by passing a `'now'` string to the `DATE()` function:

```
SELECT DATE('now')
```

SQLite also allows you to pass any number of modifier arguments to the `DATE()` function. For instance, you can get yesterday's date by passing `'-1 day'` as an argument:

```
SELECT DATE('now','-1 day')
```

You can pass also pass a date string to the `DATE()` function, and add any number of modifiers to transform the date. This example will add three months and subtract one day from 2015-12-07:

```
SELECT DATE('2015-12-07','+3 month','-1 day')
```

There are several advanced date transformations you can perform. Refer to the SQLite date functions link at the beginning of this section to get a comprehensive overview of these functionalities.

## Time Functions

Time also has a typical format, which is HH:MM:SS (this also is ISO8601 standard). HH is a two-digit military format of hours, MM is a two-digit format of minutes, and SS is a two-digit format of seconds . The separator is a colon. If you format times like

this, the strings will always be interpreted as time values. This string represents a time value of 4:31 PM and 15 seconds:

```
SELECT '16:31:15'
```

You can omit the seconds value if you are not concerned with it. SQLite will infer the seconds value to be 00:

```
SELECT '16:31'
```

Just like with dates, you can do all kinds of operations with times, like comparing one time value to another:

```
SELECT '16:31' < '08:31'
```

The `'now'` string also works with the `TIME()` function to get the current time:

```
SELECT TIME('now')
```

Also like with dates, you can use the `TIME()` function to perform transformations such as adding or subtracting hours, minutes, and seconds:

```
SELECT TIME('16:31','+1 minute')
```

## Date/Time Functions

You can have a date that also has a time value. Reasonably, the standard string format is the date and time formats concatenated together, separated by a space: 'YYYY-MM-DD HH:MM:SS'. SQLite will recognize a string in this format to be a date/time value:

```
SELECT '2015-12-13 16:04:11'
```

All the rules from the `DATE()` and `TIME()` functions can apply to the `DATETIME()` function. You can use `'now'`, transformations, and any other chronological operations we have learned. For instance, you can subtract a day from a date/time value and add three hours:

```
SELECT DATETIME('2015-12-13 16:04:11','-1 day','+3 hour')
```

There are several other functions and features for working with dates and time in SQLite, including converting dates into alternative formats and compensating for times zones. There is also support for Unix time and the Julian day number system. As said earlier, go to *http://www.sqlite.org/lang_datefunc.html* to get a comprehensive list of these functionalities.