ORIE 3120
Detailed SQLite documentation

This reading is taken from the SQLite documentation. It has been modified to simplify and remove some more advanced discussion that will not be used in the course.

Core functions [from https://www.sqlite.org/lang_corefunc.html]

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 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.

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.

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.

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. If either X or Y are NULL in instr(X,Y) then the result is NULL.

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. 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 operator.

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 is used.

random()

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

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, 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.

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.

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". In 3120 we will not deal with BLOBs.

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.

Aggregate Functions [from https://www.sqlite.org/lang_aggfunc.html]

avg(*X*)
The avg() function returns the average value of all non-NULL *X* within a group. String values that do not look like numbers are interpreted as 0. The result of avg() is always a floating point value as long as at there is at least one non-NULL input even if all inputs are integers. The result of avg() is NULL if and only if there are no non-NULL inputs.

count(*X*)
count(*)
The count(X) function returns a count of the number of times that *X* is not NULL in a group. The count(*) function (with no arguments) returns the total number of rows in the group.

group_concat(*X*)
group_concat(*X*,*Y*)
The group_concat() function returns a string which is the concatenation of all non-NULL values of *X*. If parameter *Y* is present then it is used as the separator between instances of *X*. A comma (",") is used as the separator if *Y* is omitted. The order of the concatenated elements is arbitrary.

max(*X*)
The max() aggregate function returns the maximum value of all values in the group. The maximum value is the value that would be returned last in an ORDER BY on the same column. Aggregate max() returns NULL if and only if there are no non-NULL values in the group.

min(*X*)
The min() aggregate function returns the minimum non-NULL value of all values in the group. The minimum value is the first non-NULL value that would appear in an ORDER BY of the column. Aggregate min() returns NULL if and only if there are no non-NULL values in the group.

sum(*X*)
total(*X*)
The sum() and total() aggregate functions return sum of all non-NULL values in the group. If there are no non-NULL input rows then sum() returns NULL but total() returns 0.0. NULL is not normally a helpful result for the sum of no rows but the SQL standard requires it and most other SQL database engines implement sum()

that way so SQLite does it in the same way in order to be compatible. The non-standard total() function is provided as a convenient way to work around this design problem in the SQL language.

The result of total() is always a floating point value. The result of sum() is an integer value if all non-NULL inputs are integers. If any input to sum() is neither an integer or a NULL then sum() returns a floating point value which might be an approximation to the true sum.

Sum() will throw an "integer overflow" exception if all inputs are integers or NULL and an integer overflow occurs at any point during the computation. Total() never throws an integer overflow.

Operators [from https://sqlite.org/lang_expr.html]

Binary operators

In ORIE 3120, we will assume knowledge of these binary operators. They are listed in order from highest to lowest precedence:

```
||
*   /   %
+   -
<<   >>   &   |
<   <=   >   >=
=   ==   !=   <>   IS   IS NOT   IN   LIKE
AND
OR
```

We will assume knowledge of these unary prefix operators:

```
-   ~   NOT
```

Notes:

Note that there are two variations of the equals and not equals operators. Equals can be either = or ==. The non-equals operator can be either != or <>.

The || operator is "concatenate" - it joins together the two strings of its operands. The operator % outputs the integer value of its left operand modulo its right operand.

The result of any binary operator is either a numeric value or NULL, except for the || concatenation operator which always evaluates to either NULL or a text value.

The IS and IS NOT operators work like = and != except when one or both of the operands are NULL. In this case, if both operands are NULL, then the IS operator evaluates to 1 (true) and the IS NOT operator evaluates to 0 (false). If one operand is NULL and the other is not, then the IS operator evaluates to 0 (false) and the IS NOT operator is 1 (true). It is not possible for an IS or IS NOT expression to evaluate to NULL. Operators IS and IS NOT have the same precedence as =.

The LIKE operator

The LIKE operator does a pattern matching comparison. The operand to the right of the LIKE operator contains the pattern and the left hand operand contains the string to match against the pattern. A percent symbol ("%") in the LIKE pattern matches any sequence of zero or more characters in the string. An underscore ("_") in the LIKE pattern matches any single character in the string. Any other character matches itself or its lower/upper case equivalent (i.e. case-insensitive matching).

If the optional ESCAPE clause is present, then the expression following the ESCAPE keyword must evaluate to a string consisting of a single character. This character may be used in the LIKE pattern to include literal percent or underscore characters. The escape character followed by a percent symbol (%), underscore (_), or a second instance of the escape character itself matches a literal percent symbol, underscore, or a single escape character, respectively.
The infix LIKE operator is implemented by calling the application-defined SQL functions like(*Y*,*X*) or like(*Y*,*X*,*Z*).

The BETWEEN operator

The BETWEEN operator is logically equivalent to a pair of comparisons. "*x* **BETWEEN** *y* **AND** *z*" is equivalent to "*x***>=***y* **AND** *x***<=***z*" except that with BETWEEN, the *x*expression is only evaluated once. The precedence of the BETWEEN operator is the same as the precedence as operators **==** and **!=** and **LIKE** and groups left to right.

The CASE expression

A CASE expression serves a role similar to IF-THEN-ELSE in other programming languages.

The optional expression that occurs in between the CASE keyword and the first WHEN keyword is called the "base" expression. There are two basic forms of the CASE expression: those with a base expression and those without.

In a CASE without a base expression, each WHEN expression is evaluated and the result treated as a boolean, starting with the leftmost and continuing to the right. The result of the CASE expression is the evaluation of the THEN expression that corresponds to the first WHEN expression that evaluates to true. Or, if none of the WHEN expressions evaluate to true, the result of evaluating the ELSE expression, if any. If there is no ELSE expression and none of the WHEN expressions are true, then the overall result is NULL.

A NULL result is considered untrue when evaluating WHEN terms.

In a CASE with a base expression, the base expression is evaluated just once and the result is compared against the evaluation of each WHEN expression from left to right. The result of the CASE expression is the evaluation of the THEN expression that corresponds to the first WHEN expression for which the comparison is true. Or, if none of the WHEN expressions evaluate to a value equal to the base expression, the result of evaluating the ELSE expression, if any. If there is no ELSE expression and none of the WHEN expressions produce a result equal to the base expression, the overall result is NULL.

When comparing a base expression against a WHEN expression, the same collating sequence, affinity, and NULL-handling rules apply as if the base expression and WHEN expression are respectively the left- and right-hand operands of an **=** operator.

If the base expression is NULL then the result of the CASE is always the result of evaluating the ELSE expression if it exists, or NULL if it does not.

The IN and NOT IN operators

The IN and NOT IN operators take an expression on the left and a list of values or a subquery on the right. When the right operand of an IN or NOT IN operator is a subquery, the subquery must have the same number of columns as there are columns in the row value of the left operand. The subquery on the right of an IN or NOT IN operator must be a scalar subquery if the left expression is not a row value expression. If the right operand of an IN or NOT IN operator is a list of values, each of those values must be scalars and the left expression must also be a scalar. The right-hand side of an IN or NOT IN operator can be a table *name* or table-valued function *name* in which case the right-hand side is understood to be subquery of the form "(SELECT * FROM *name*)". When the right operand is an empty set, the result of IN is false and the result of NOT IN is true, regardless of the left operand and even if the left operand is NULL.

The result of an IN or NOT IN operator is determined by the following matrix:

Left operand is NULL	Right operand contains NULL	Right operand is an empty set	Left operand found within right operand	Result of IN operator	Result of NOT IN operator
no	no	no	no	false	true
does not matter	no	yes	no	false	true
no	does not matter	no	yes	true	false
no	yes	no	no	NULL	NULL
yes	does not matter	no	does not matter	NULL	NULL

The EXISTS operator

The EXISTS operator always evaluates to one of the integer values 0 and 1. If executing the SELECT statement specified as the right-hand operand of the EXISTS operator would return one or more rows, then the EXISTS operator evaluates to 1. If executing the SELECT would return no rows at all, then the EXISTS operator evaluates to 0.

The number of columns in each row returned by the SELECT statement (if any) and the specific values returned have no effect on the results of the EXISTS operator. In particular, rows containing NULL values are not handled any differently from rows without NULL values.

Date & Time Functions [from www.sqlite.org/lang_datefunc.html]

In ORIE 3120 we will use four of the five date and time functions that are provided by SQLite. The four that we will use are:

1. **date(***timestring, modifier, modifier, ...***)**
2. **time(***timestring, modifier, modifier, ...***)**
3. **datetime(***timestring, modifier, modifier, ...***)**

4. **strftime(***format, timestring, modifier, modifier, ...***)**

We will not use julianday().

All five date and time functions take a time string as an argument. The time string is followed by zero or more modifiers. The strftime() function also takes a format string as its first argument.

The date() function returns the date in this format: YYYY-MM-DD. The time() function returns the time as HH:MM:SS. The datetime() function returns "YYYY-MM-DD HH:MM:SS". The strftime() routine returns the date formatted according to the format string specified as the first argument. The format string supports the most common substitutions found in the strftime() function from the standard C library plus two new substitutions, %f and %J. The following is a complete list of valid strftime() substitutions:

%d	day of month: 00
%f	fractional seconds: SS.SSS
%H	hour: 00-24
%j	day of year: 001-366
%J	Julian day number
%m	month: 01-12
%M	minute: 00-59
%s	seconds since 1970-01-01
%S	seconds: 00-59
%w	day of week 0-6 with Sunday==0
%W	week of year: 00-53
%Y	year: 0000-9999
%%	%

Notice that all other date and time functions can be expressed in terms of strftime():

Function	**Equivalent strftime()**

date(...)	strftime('%Y-%m-%d', ...)
time(...)	strftime('%H:%M:%S', ...)
datetime(...)	strftime('%Y-%m-%d %H:%M:%S', ...)

The only reasons for providing functions other than strftime() is for convenience and for efficiency.

Time Strings

In ORIE 3120, we will assume that time strings are in the following format, although SQLite supports other formats

YYYY-MM-DD HH:MM:SS