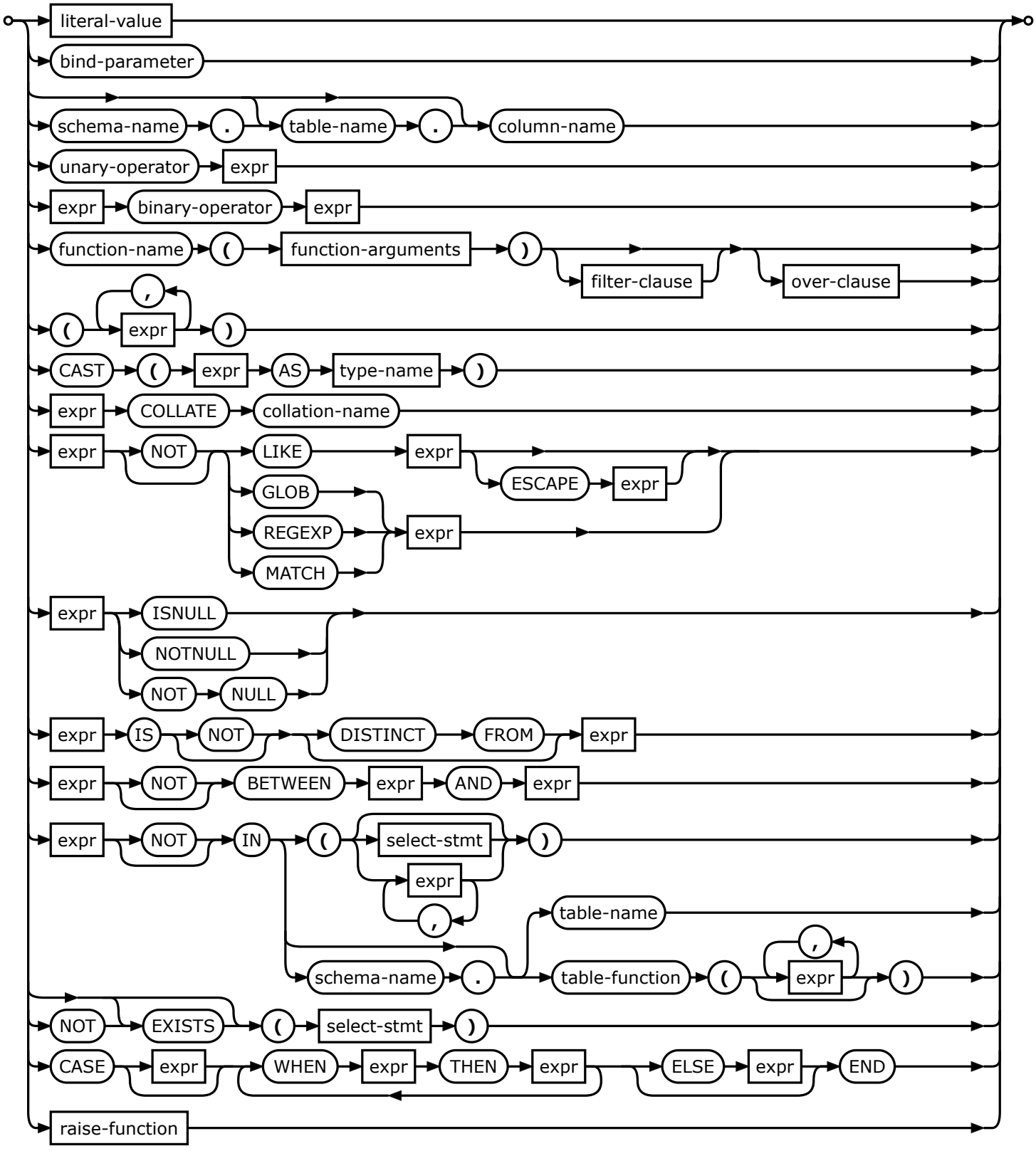
Small. Fast. Reliable.
Choose any three.

SQL Language Expressions

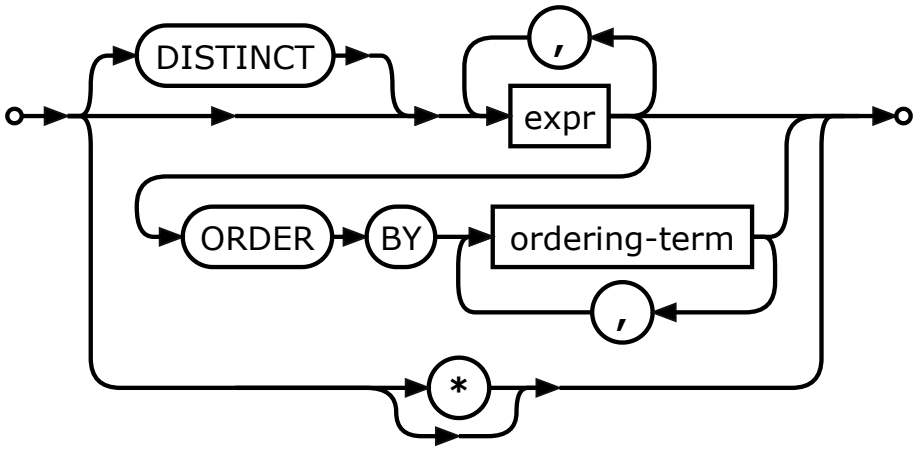
1. Syntax

expr: hide

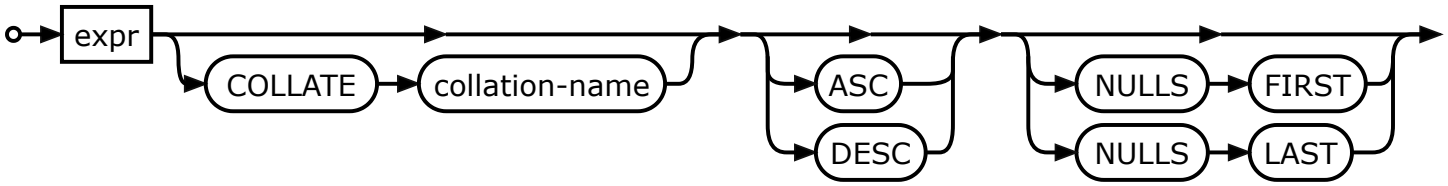


filter-clause: hide

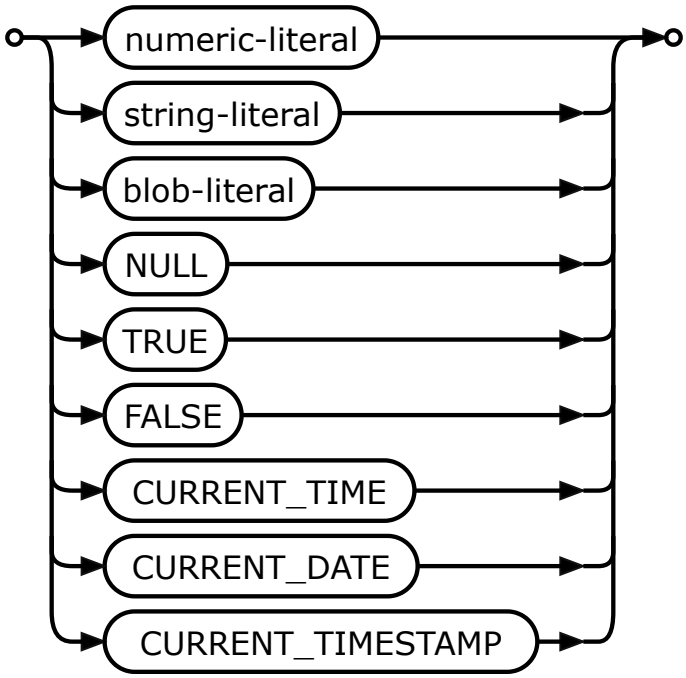
function-arguments: hide



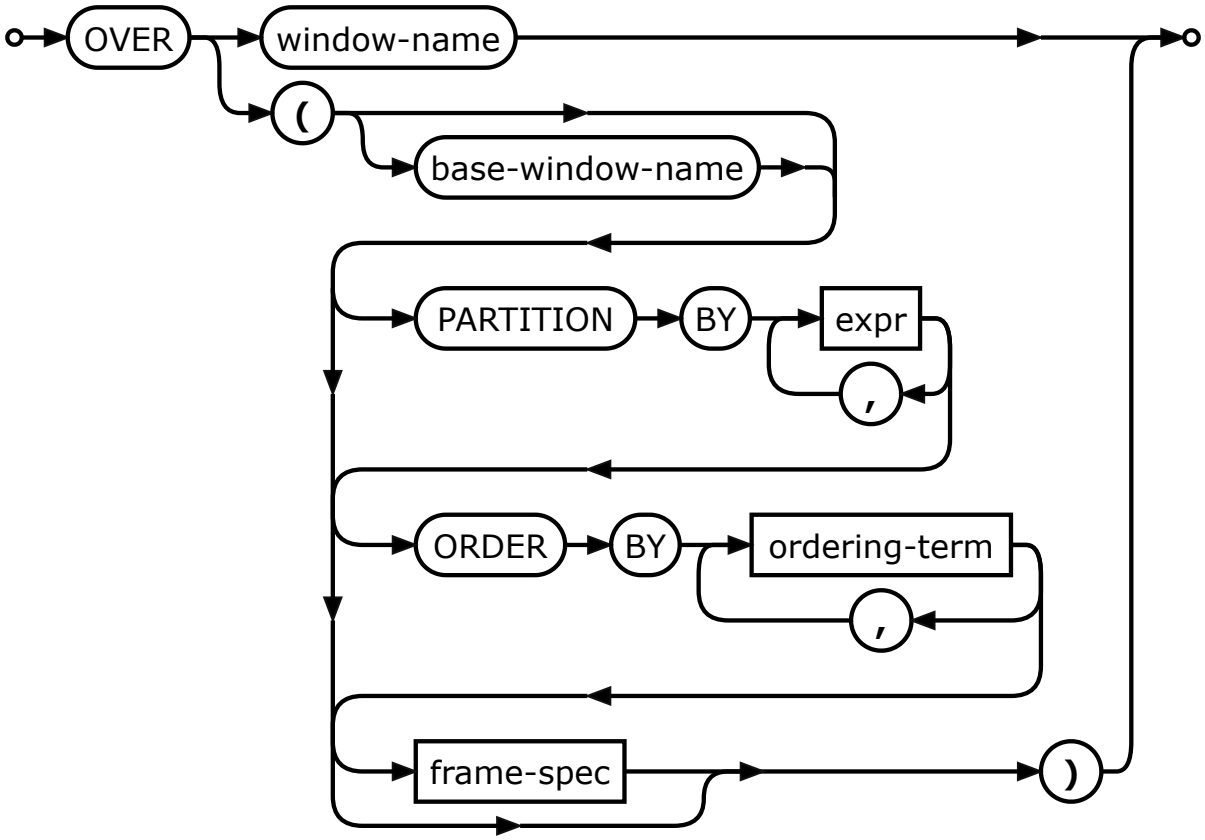
ordering-term: hide



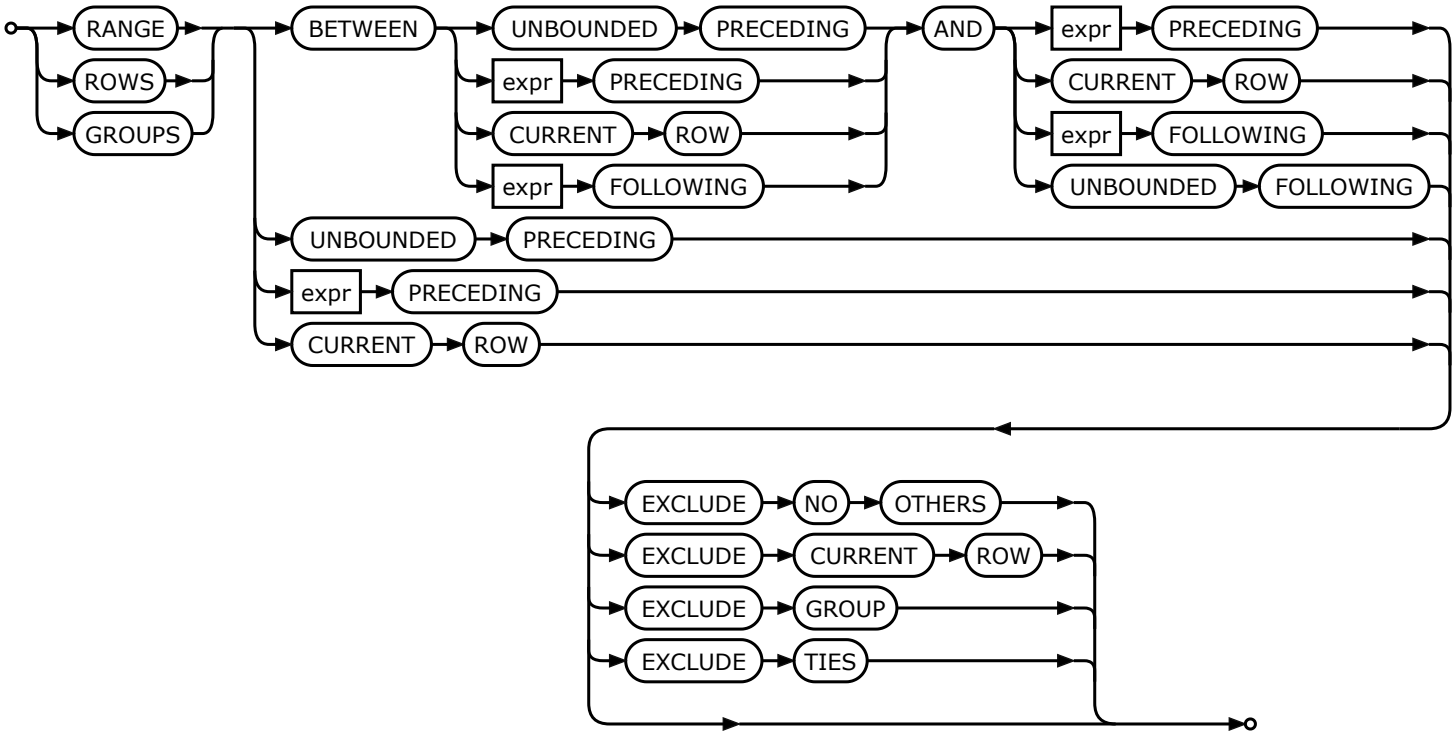
literal-value: hide



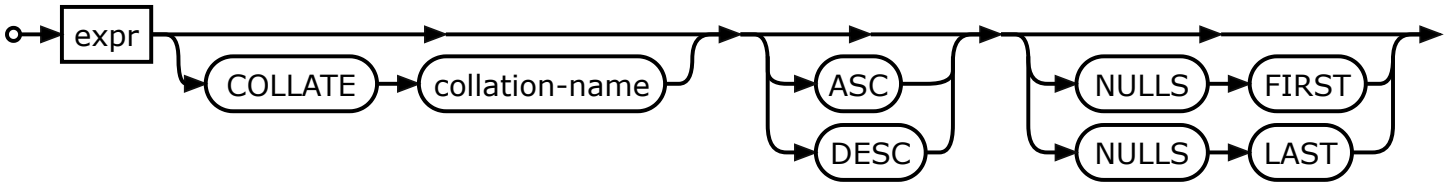
over-clause: hide



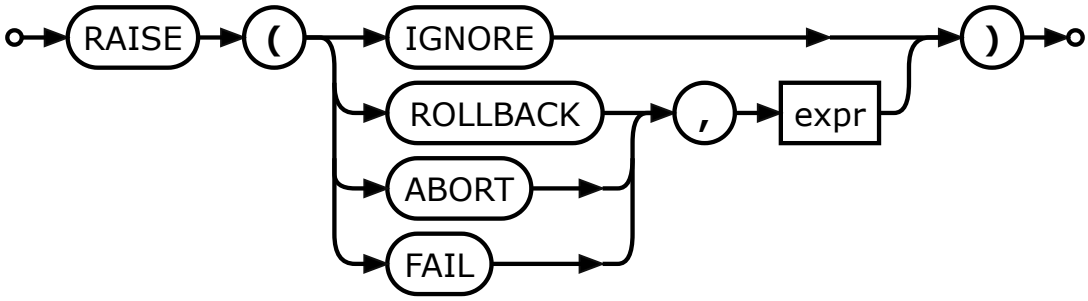
frame-spec: hide



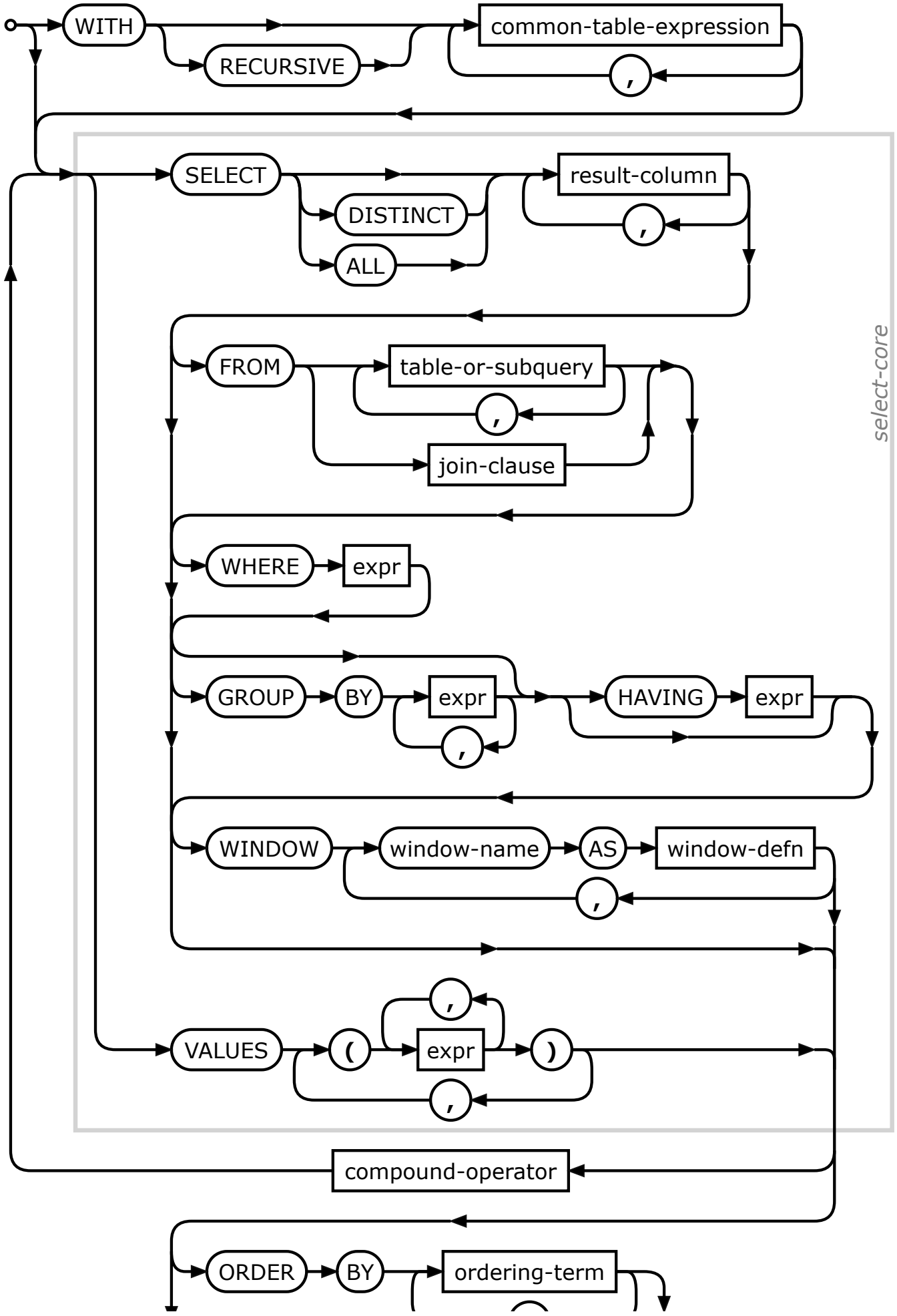
ordering-term: hide

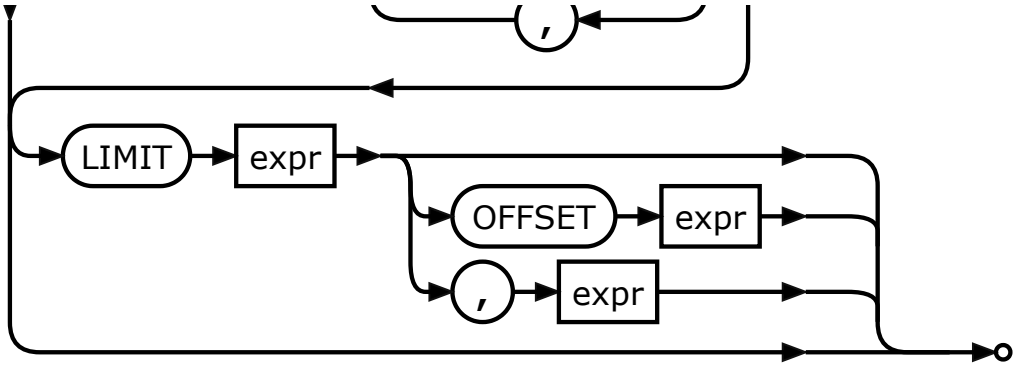


raise-function: hide

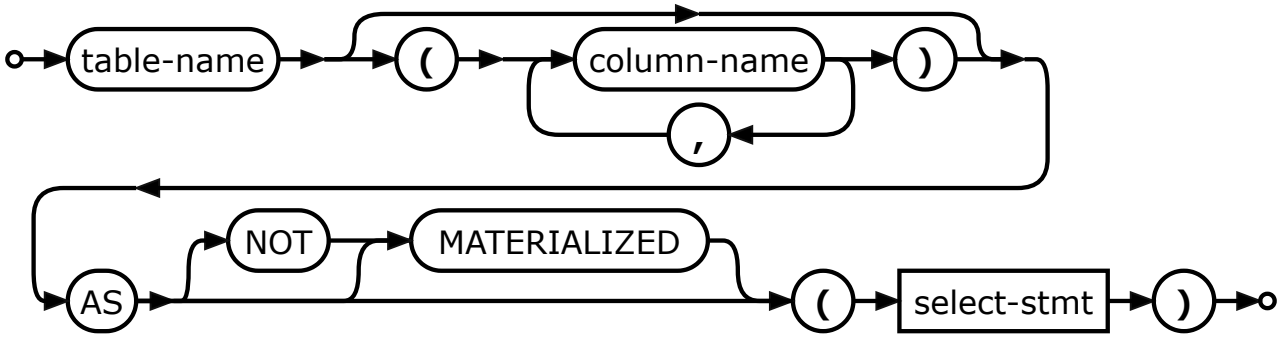


select-stmt: hide

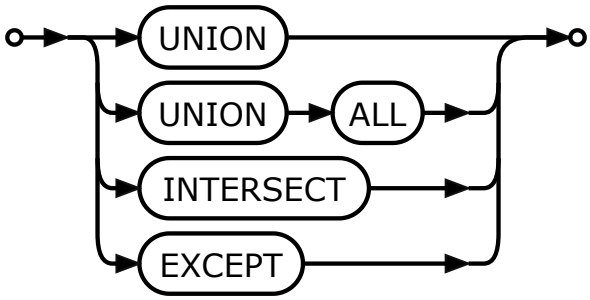

WITH
RECURSIVE
common-table-expression
,
SELECT
DISTINCT
ALL
result-column
,
FROM
table-or-subquery
,
join-clause
WHERE
expr
GROUP
BY
expr
,
HAVING
expr
WINDOW
window-name
AS
window-defn
,
VALUES
(
expr
)
,
,
compound-operator
ORDER
BY
ordering-term
select-core



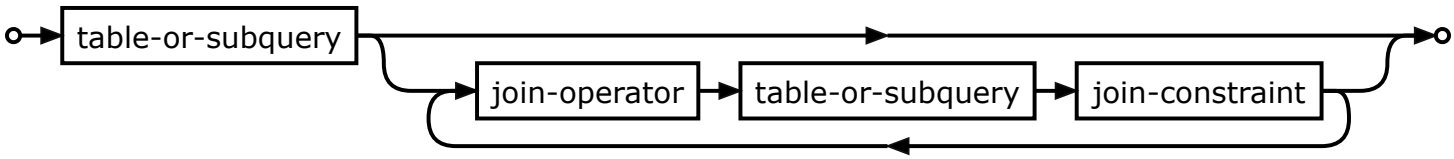
common-table-expression: hide



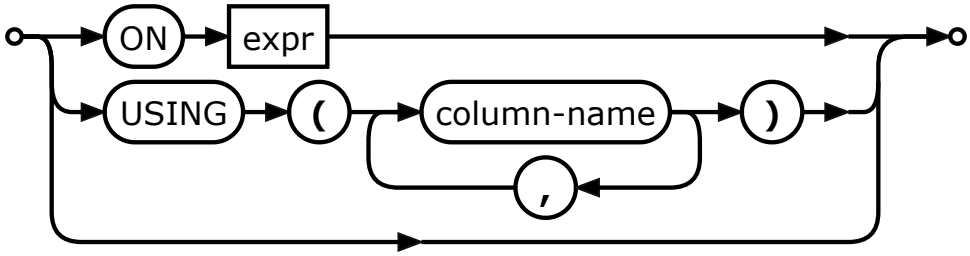
compound-operator: hide



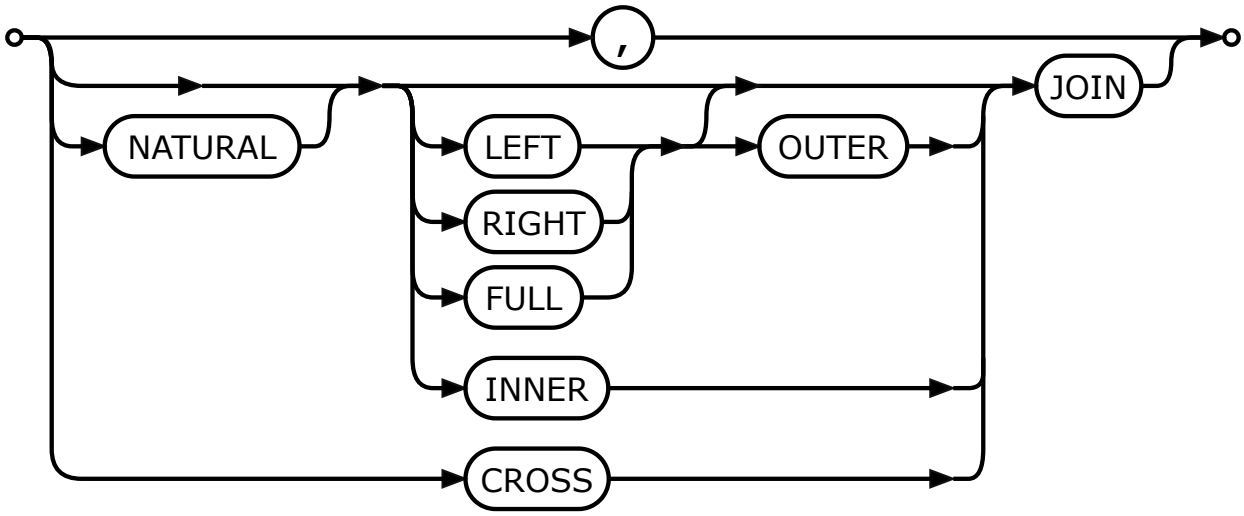
join-clause: hide



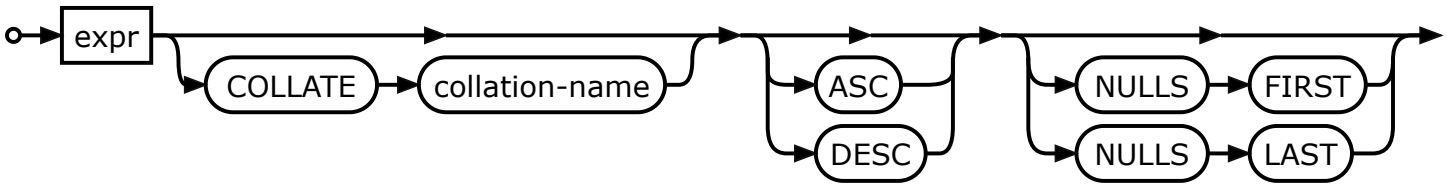
join-constraint: hide



join-operator: hide



ordering-term: hide



result-column: hide

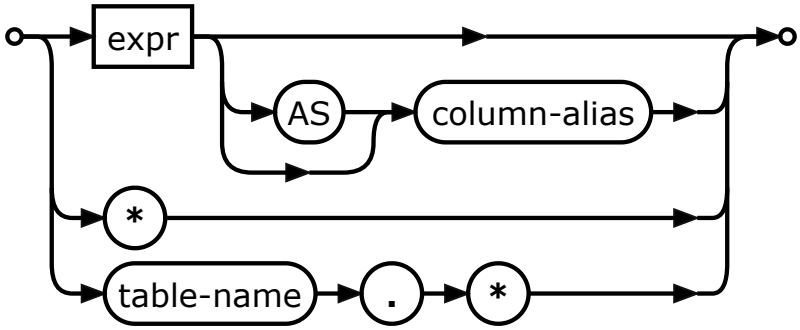
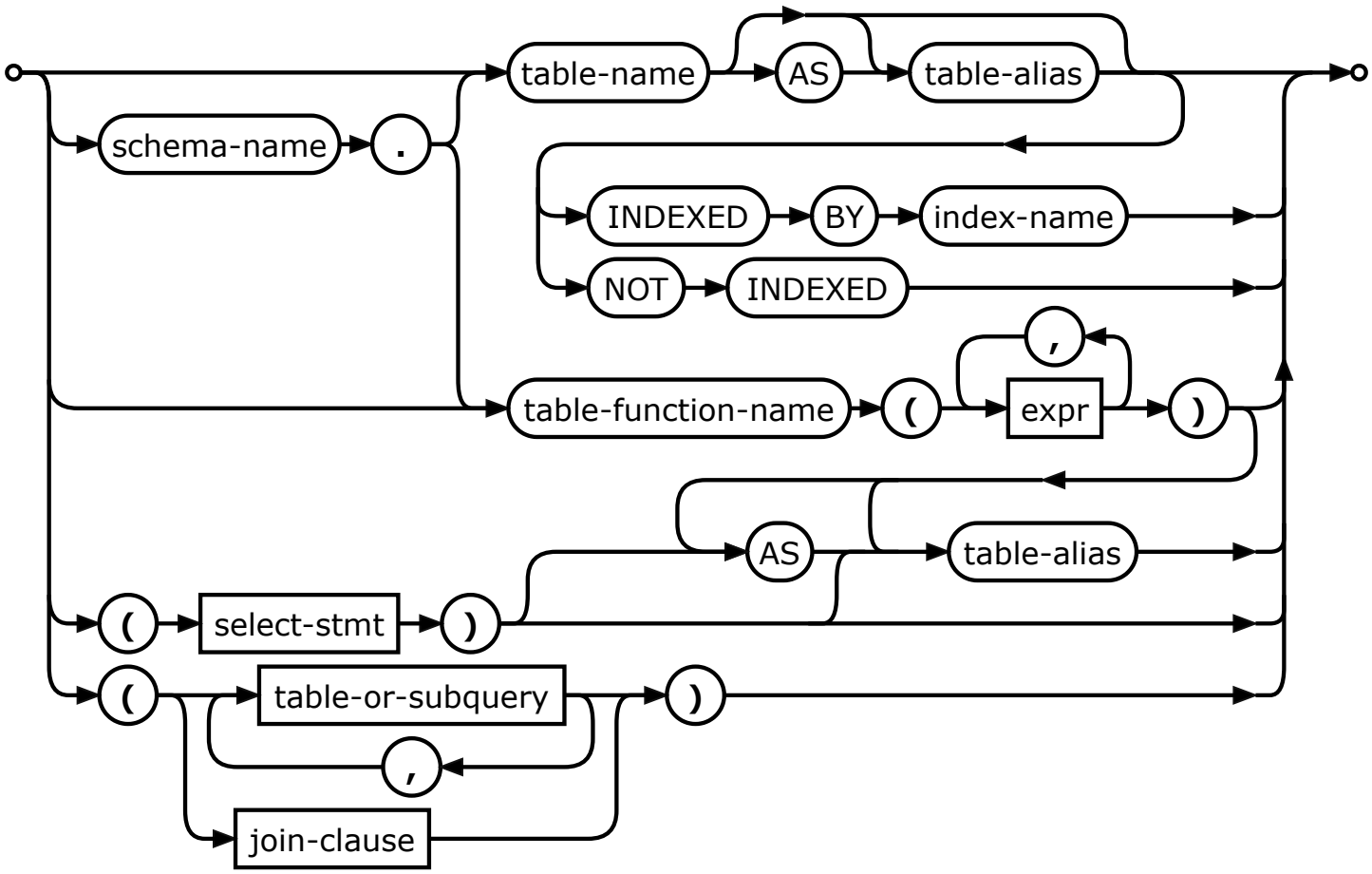
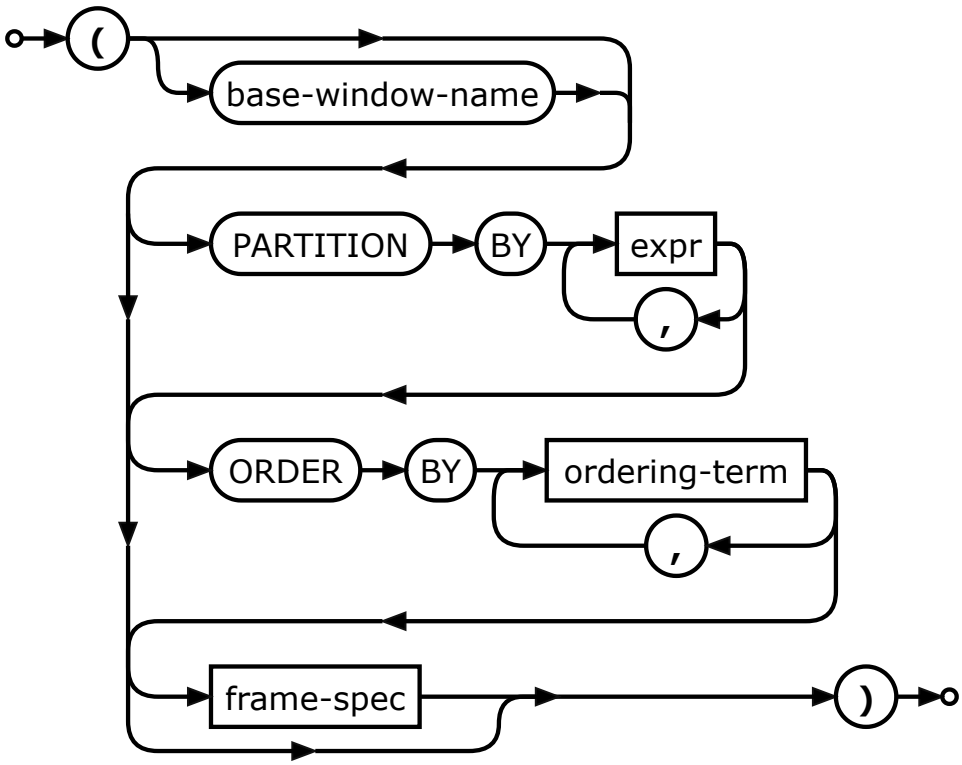


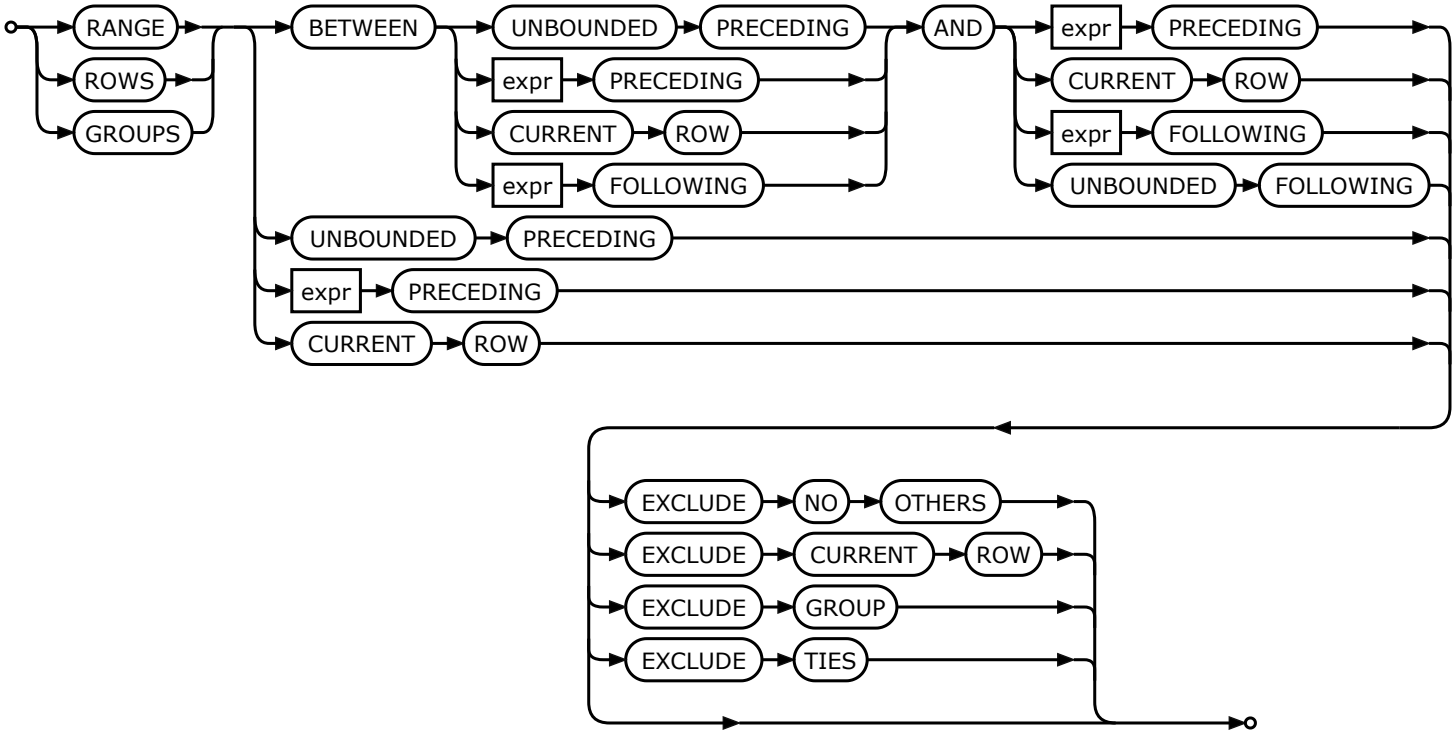
table-or-subquery: hide



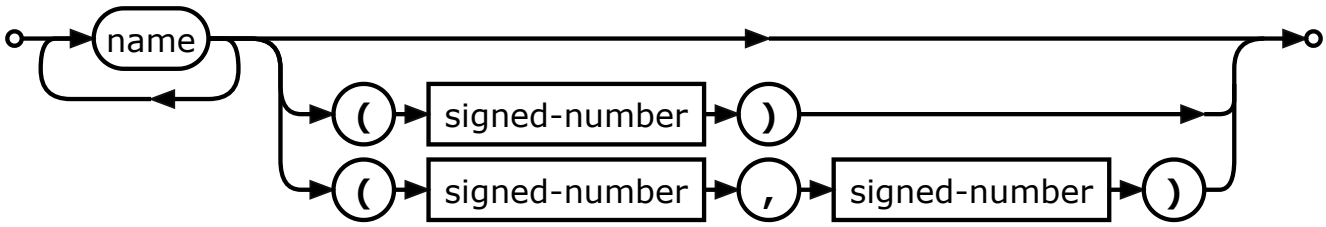
window-defn: hide



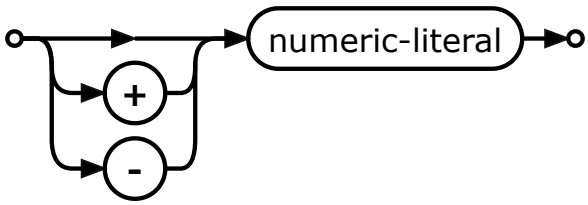
frame-spec: hide



type-name: hide



signed-number: hide



2. Operators, and Parse-Affecting Attributes

SQLite understands these operators, listed in precedence[1] order (top to bottom / highest to lowest):

Operators [2]
~ [expr] + [expr] - [expr]
[expr] COLLATE (collation-name) [3]
|| -> ->>

* / %
+ -
& | << >>
[expr] ESCAPE [escape-character-expr] [4]
< > <= >=
= == <> != IS IS NOT IS DISTINCT FROM IS NOT DISTINCT FROM [expr] BETWEEN[5] [expr] AND [expr] IN[5] MATCH[5] LIKE[5] REGEXP[5] GLOB[5] [expr] ISNULL [expr] NOTNULL [expr] NOT NULL
NOT [expr]
AND
OR

1. Operators shown within the same table cell share precedence.
2. "[expr]" denotes operand locations for non-binary operators.
   Operators with no "[expr]" adjunct are binary and left-associative.
3. The COLLATE clause (with its collation-name) acts as a single postfix operator.
4. The ESCAPE clause (with its escape character) acts as a single postfix operator.
   It can only bind to a preceding [expr] LIKE [expr] expression.
5. Each keyword in (BETWEEN IN GLOB LIKE MATCH REGEXP) may be prefixed
   by NOT, retaining the bare operator's precedence and associativity.

The COLLATE operator is a unary postfix operator that assigns a collating sequence to an expression. The collating sequence set by the COLLATE operator overrides the collating sequence determined by the COLLATE clause in a table column definition. See the detailed discussion on collating sequences in the Datatype In SQLite3 document for additional information.

The unary operator + is a no-op. It can be applied to strings, numbers, blobs or NULL and it always returns a result with the same value as the operand.

Note that there are two variations of the equals and not equals operators. Equals can be either = or ==. The not-equal operator can be either != or <>. The || operator is "concatenate" - it joins together the two strings of its operands. The -> and ->> operators are "extract"; they extract the RHS component from the LHS. For an example, see JSON subcomponent extraction.

The % operator casts both of its operands to type INTEGER and then computes the remainder after dividing the left integer by the right integer. The other arithmetic operators perform integer arithmetic if both operands are integers and no overflow would result, or floating point arithmetic, per IEEE Standard 754, if either operand is a real value or integer arithmetic would produce an overflow. Integer divide yields an integer result, truncated toward zero.

The result of any binary operator is either a numeric value or NULL, except for the || concatenation operator, and the -> and ->> extract operators which can return values of any type.

All operators generally evaluate to NULL when any operand is NULL, with specific exceptions as stated below. This is in accordance with the SQL92 standard.

When paired with NULL:
AND evaluates to 0 (false) when the other operand is false; and
OR evaluates to 1 (true) when the other operand is true.

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.

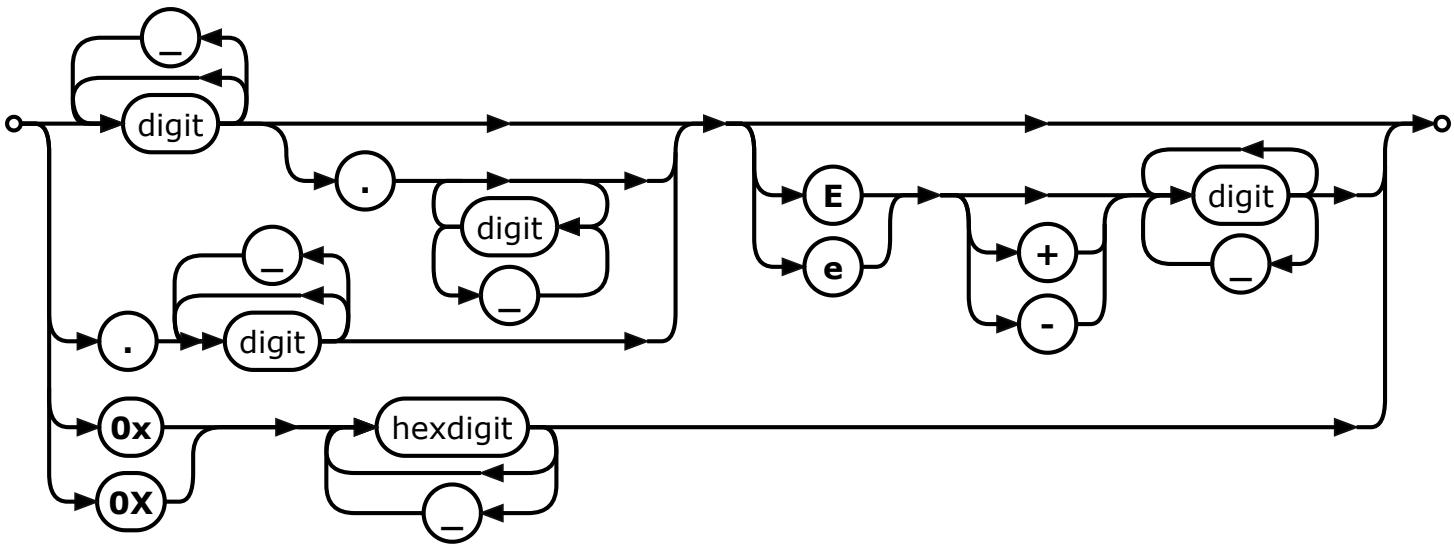
The IS NOT DISTINCT FROM operator is an alternative spelling for the IS operator. Likewise, the IS DISTINCT FROM operator means the same thing as IS NOT. Standard SQL does not support the compact IS and IS NOT notation. Those compact forms are an SQLite extension. You have to use the prolix and much less readable IS NOT DISTINCT FROM and IS DISTINCT FROM operators on other SQL database engines.

3. Literal Values (Constants)

A literal value represents a constant. Literal values may be integers, floating point numbers, strings, BLOBs, or NULLs.

The syntax for integer and floating point literals (collectively "numeric literals") is shown by the following diagram:

numeric-literal:



If a numeric literal has a decimal point or an exponentiation clause or if it is less than -9223372036854775808 or greater than 9223372036854775807, then it is a floating point literal. Otherwise is it is an integer literal. The "E" character that begins the exponentiation clause of a floating point literal can be either upper or lower case. The "." character is always used as the decimal point even if the locale setting specifies "," for this role - the use of "," for the decimal point would result in syntactic ambiguity.

Beginning in SQLite version 3.46.0 (2024-05-23), a single extra underscore ("_") character can be added between any two digits. The underscores are purely for human readability and are ignored by SQLite.

Hexadecimal integer literals follow the C-language notation of "0x" or "0X" followed by hexadecimal digits. For example, 0x1234 means the same as 4660 and 0x8000000000000000 means the same as -9223372036854775808. Hexadecimal integer literals are interpreted as 64-bit two's-complement integers and are thus limited to sixteen significant digits of precision. Support for hexadecimal integers was added to SQLite version 3.8.6 (2014-08-15). For backwards compatibility, the "0x" hexadecimal integer notation is only understood by the SQL language parser, not by the type conversions routines. String variables that contain text formatted like hexadecimal integers are not interpreted as hexadecimal integers when coercing the string value into an integer due to a CAST expression or for a column affinity transformation or prior to performing a numeric operation or for any other run-time conversions. When coercing a string value in the format of a hexadecimal integer into an integer value, the conversion process stops when the 'x' character is seen so the resulting integer value is always zero. SQLite only understands the hexadecimal integer notation when it appears in the SQL statement text, not when it appears as part of the content of the database.

A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row - as in Pascal. C-style escapes using the backslash character are not supported because they are not standard SQL.

BLOB literals are string literals containing hexadecimal data and preceded by a single "x" or "X" character. Example: X'53514C697465'

A literal value can also be the token "NULL".

4. Parameters

A "variable" or "parameter" token specifies a placeholder in the expression for a value that is filled in at runtime using the sqlite3_bind() family of C/C++ interfaces. Parameters can take several forms:

?*NNN*	A question mark followed by a number *NNN* holds a spot for the NNN-th parameter. NNN must be between 1 and SQLITE_MAX_VARIABLE_NUMBER.
?	A question mark that is not followed by a number creates a parameter with a number one greater than the largest parameter number already assigned. If this means the parameter number is greater than

SQLITE_MAX_VARIABLE_NUMBER, it is an error. This parameter format is provided for compatibility with other database engines. But because it is easy to miscount the question marks, the use of this parameter format is discouraged. Programmers are encouraged to use one of the symbolic formats below or the ?NNN format above instead.

:*AAAA* A colon followed by an identifier name holds a spot for a named parameter with the name :AAAA. Named parameters are also numbered. The number assigned is one greater than the largest parameter number already assigned. If this means the parameter would be assigned a number greater than SQLITE_MAX_VARIABLE_NUMBER, it is an error. To avoid confusion, it is best to avoid mixing named and numbered parameters.

@*AAAA* An "at" sign works exactly like a colon, except that the name of the parameter created is @AAAA.

$*AAAA* A dollar-sign followed by an identifier name also holds a spot for a named parameter with the name $AAAA. The identifier name in this case can include one or more occurrences of "::" and a suffix enclosed in "(...)" containing any text at all. This syntax is the form of a variable name in the Tcl programming language. The presence of this syntax results from the fact that SQLite is really a Tcl extension that has escaped into the wild.

Parameters that are not assigned values using sqlite3_bind() are treated as NULL. The sqlite3_bind_parameter_index() interface can be used to translate a symbolic parameter name into its equivalent numeric index.

The maximum parameter number is set at compile-time by the SQLITE_MAX_VARIABLE_NUMBER macro. An individual database connection D can reduce its maximum parameter number below the compile-time maximum using the sqlite3_limit(D, SQLITE_LIMIT_VARIABLE_NUMBER,...) interface.

5. The LIKE, GLOB, REGEXP, MATCH, and extract operators

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). Important Note: SQLite only understands upper/lower case for ASCII characters by default. The LIKE operator is case sensitive by default for unicode characters that are beyond the ASCII range. For example, the expression **'a' LIKE 'A'** is TRUE but **'æ' LIKE 'Æ'** is FALSE. The ICU extension to SQLite includes an enhanced version of the LIKE operator that does case folding across all unicode characters.

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 LIKE operator can be made case sensitive using the case_sensitive_like pragma.

The GLOB operator is similar to LIKE but uses the Unix file globbing syntax for its wildcards. Also, GLOB is case sensitive, unlike LIKE. Both GLOB and LIKE may be preceded by the NOT keyword to invert the sense of the test. The infix GLOB operator is implemented by calling the function glob(*Y*,*X*) and can be modified by overriding that function.

The REGEXP operator is a special syntax for the regexp() user function. No regexp() user function is defined by default and so use of the REGEXP operator will normally result in an error message. If an application-defined SQL function named "regexp" is added at run-time, then the "*X* REGEXP *Y*" operator will be implemented as a call to "regexp(*Y*,*X*)".

The MATCH operator is a special syntax for the match() application-defined function. The default match() function implementation raises an exception and is not really useful for anything. But extensions can override the match() function with more helpful logic.

The extract operators act as a special syntax for functions "->"() and "->>"(). Default implementations for these functions perform JSON subcomponent extraction, but extensions can override them for other purposes.

6. 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.

7. 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 fundamental 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.

Both forms of the CASE expression use lazy, or short-circuit, evaluation.

The only difference between the following two CASE expressions is that the *x* expression is evaluated exactly once in the first example but might be evaluated multiple times in the second:

- `CASE x WHEN w1 THEN r1 WHEN w2 THEN r2 ELSE r3 END`
- `CASE WHEN x=w1 THEN r1 WHEN x=w2 THEN r2 ELSE r3 END`

The built-in iif(x,y,z) SQL function is logically equivalent to "CASE WHEN x THEN y ELSE z END". The iif() function is found in SQL Server and is included in SQLite for compatibility. Some developers prefer the iif() function because it is more concise.

8. 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

Note that SQLite allows the parenthesized list of scalar values on the right-hand side of an IN or NOT IN operator to be an empty list but most other SQL database engines and the SQL92 standard require the list to contain at least one element.

9. Table Column Names

A column name can be any of the names defined in the CREATE TABLE statement or one of the following special identifiers: "**ROWID**", "**OID**", or "**_ROWID_**". The three special identifiers describe the unique integer key (the rowid) associated with every row of every table and so are not available on WITHOUT ROWID tables. The special identifiers only refer to the row key if the CREATE TABLE statement does not define a real column with the same name. The rowid can be used anywhere a regular column can be used.

10. 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.

11. Subquery Expressions

A SELECT statement enclosed in parentheses is a subquery. All types of SELECT statement, including aggregate and compound SELECT queries (queries with keywords

like UNION or EXCEPT) are allowed as scalar subqueries. The value of a subquery expression is the first row of the result from the enclosed SELECT statement. The value of a subquery expression is NULL if the enclosed SELECT statement returns no rows.

A subquery that returns a single column is a scalar subquery and can be used most anywhere. A subquery that returns two or more columns is a row value subquery and can only be used as an operand of a comparison operator or as the value in an UPDATE SET clause whose column name list has the same size.

12. Correlated Subqueries

A SELECT statement used as either a scalar subquery or as the right-hand operand of an IN, NOT IN or EXISTS expression may contain references to columns in the outer query. Such a subquery is known as a correlated subquery. A correlated subquery is reevaluated each time its result is required. An uncorrelated subquery is evaluated only once and the result reused as necessary.

13. CAST expressions

A CAST expression of the form "CAST(*expr* AS *type-name*)" is used to convert the value of *expr* to a different storage class specified by type-name. A CAST conversion is similar to the conversion that takes place when a column affinity is applied to a value except that with the CAST operator the conversion always takes place even if the conversion lossy and irreversible, whereas column affinity only changes the data type of a value if the change is lossless and reversible.

If the value of *expr* is NULL, then the result of the CAST expression is also NULL. Otherwise, the storage class of the result is determined by applying the rules for determining column affinity to the type-name.

Affinity of type-name	**Conversion Processing**
NONE	Casting a value to a type-name with no affinity causes the value to be converted into a BLOB. Casting to a BLOB consists of first casting the value to TEXT in the encoding of the database connection, then interpreting the resulting byte sequence as a BLOB instead of as TEXT.
TEXT	To cast a BLOB value to TEXT, the sequence of bytes that make up the BLOB is interpreted as text encoded using the database encoding. Casting an INTEGER or REAL value into TEXT renders the value as if via sqlite3_snprintf() except that the resulting TEXT uses the encoding of the database connection.
REAL	When casting a BLOB value to a REAL, the value is first converted to TEXT.

	When casting a TEXT value to REAL, the longest possible prefix of the value that can be interpreted as a real number is extracted from the TEXT value and the remainder ignored. Any leading spaces in the TEXT value are ignored when converging from TEXT to REAL. If there is no prefix that can be interpreted as a real number, the result of the conversion is 0.0.
INTEGER	When casting a BLOB value to INTEGER, the value is first converted to TEXT. When casting a TEXT value to INTEGER, the longest possible prefix of the value that can be interpreted as an integer number is extracted from the TEXT value and the remainder ignored. Any leading spaces in the TEXT value when converting from TEXT to INTEGER are ignored. If there is no prefix that can be interpreted as an integer number, the result of the conversion is 0. If the prefix integer is greater than +9223372036854775807 then the result of the cast is exactly +9223372036854775807. Similarly, if the prefix integer is less than -9223372036854775808 then the result of the cast is exactly -9223372036854775808. When casting to INTEGER, if the text looks like a floating point value with an exponent, the exponent will be ignored because it is no part of the integer prefix. For example, "CAST('123e+5' AS INTEGER)" results in 123, not in 12300000. The CAST operator understands decimal integers only — conversion of hexadecimal integers stops at the "x" in the "0x" prefix of the hexadecimal integer string and thus result of the CAST is always zero. A cast of a REAL value into an INTEGER results in the integer between the REAL value and zero that is closest to the REAL value. If a REAL is greater than the greatest possible signed integer (+9223372036854775807) then the result is the greatest possible signed integer and if the REAL is less than the least possible signed integer (-9223372036854775808) then the result is the least possible signed integer. Prior to SQLite version 3.8.2 (2013-12-06), casting a REAL value greater than +9223372036854775807.0 into an integer resulted in the most negative integer, -9223372036854775808. This behavior was meant to emulate the behavior of x86/x64 hardware when doing the equivalent cast.

NUMERIC	Casting a TEXT or BLOB value into NUMERIC yields either an INTEGER or a REAL result. If the input text looks like an integer (there is no decimal point nor exponent) and the value is small enough to fit in a 64-bit signed integer, then the result will be INTEGER. Input text that looks like floating point (there is a decimal point and/or an exponent) and the text describes a value that can be losslessly converted back and forth between IEEE 754 64-bit float and a 51-bit signed integer, then the result is INTEGER. (In the previous sentence, a 51-bit integer is specified since that is one bit less than the length of the mantissa of an IEEE 754 64-bit float and thus provides a 1-bit of margin for the text-to-float conversion operation.) Any text input that describes a value outside the range of a 64-bit signed integer yields a REAL result. Casting a REAL or INTEGER value to NUMERIC is a no-op, even if a real value could be losslessly converted to an integer.

Note that the result from casting any non-BLOB value into a BLOB and the result from casting any BLOB value into a non-BLOB value may be different depending on whether the database encoding is UTF-8, UTF-16be, or UTF-16le.

14. Boolean Expressions

The SQL language features several contexts where an expression is evaluated and the result converted to a boolean (true or false) value. These contexts are:

- the WHERE clause of a SELECT, UPDATE or DELETE statement,
- the ON or USING clause of a join in a SELECT statement,
- the HAVING clause of a SELECT statement,
- the WHEN clause of an SQL trigger, and
- the WHEN clause or clauses of some CASE expressions.

To convert the results of an SQL expression to a boolean value, SQLite first casts the result to a NUMERIC value in the same way as a CAST expression. A numeric zero value (integer value 0 or real value 0.0) is considered to be false. A NULL value is still NULL. All other values are considered true.

For example, the values NULL, 0.0, 0, 'english' and '0' are all considered to be false. Values 1, 1.0, 0.1, -0.1 and '1english' are considered to be true.

Beginning with SQLite 3.23.0 (2018-04-02), SQLite recognizes the identifiers "TRUE" and "FALSE" as boolean literals, if and only if those identifiers are not already used for some other meaning. If there already exists columns or tables or other objects named TRUE or FALSE, then for the sake of backwards compatibility, the TRUE and FALSE identifiers refer to those other objects, not to the boolean values.

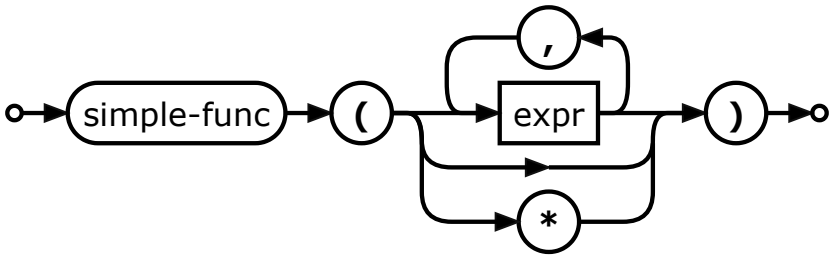
The boolean identifiers TRUE and FALSE are usually just aliases for the integer values 1 and 0, respectively. However, if TRUE or FALSE occur on the right-hand side of an IS operator, then the IS operator evaluates the left-hand operand as a boolean value and returns an appropriate answer.

15. Functions

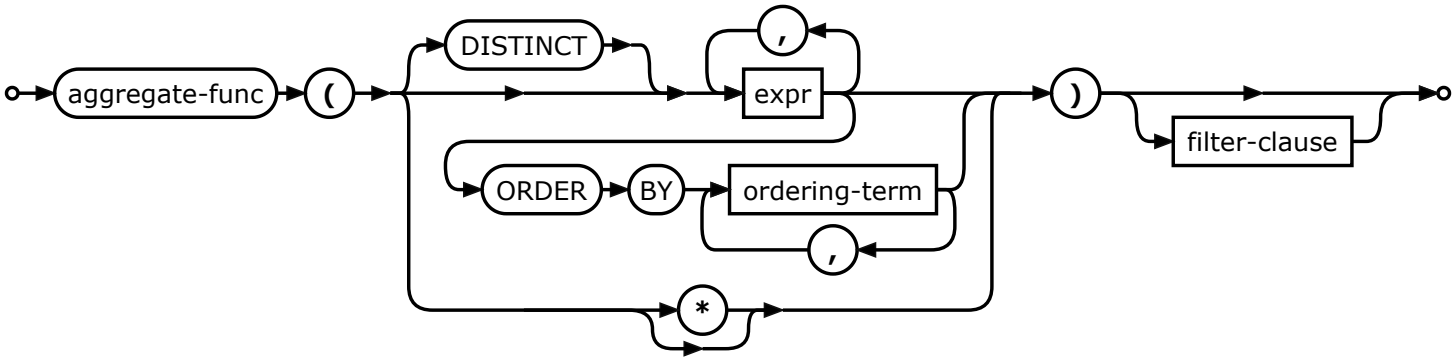
SQLite supports many simple, aggregate, and window SQL functions. For presentation purposes, simple functions are further subdivided into core functions, date-time functions, math functions, and JSON functions. Applications can add new functions, written in C/C++, using the sqlite3_create_function() interface.

The main expression bubble diagram above shows a single syntax for all function invocations. But this is merely to simplify the expression bubble diagram. In reality, each type of function has a slightly different syntax, shown below. The function invocation syntax shown in the main expression bubble diagram is the union of the three syntaxes shown here:

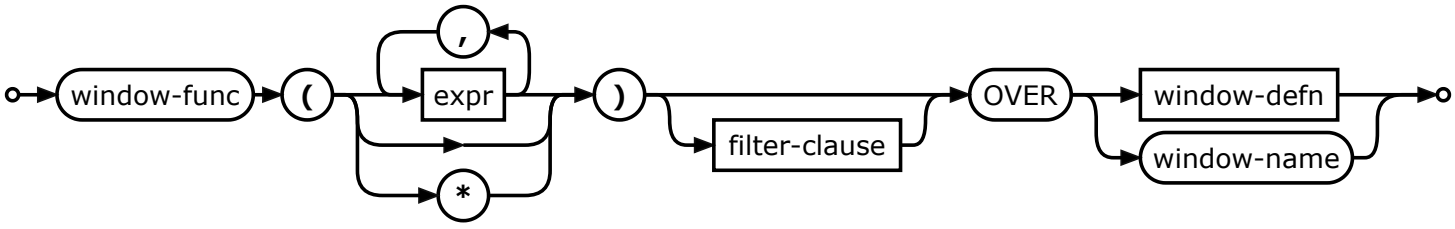
simple-function-invocation:



aggregate-function-invocation:



window-function-invocation:



The OVER clause is required for window functions and is prohibited otherwise. The DISTINCT keyword and the ORDER BY clause is only allowed in aggregate functions. The

FILTER clause may not appear on a simple function.

It is possible to have an aggregate function with the same name as a simple function, as long as the number of arguments for the two forms of the function are different. For example, the max() function with a single argument is an aggregate and the max() function with two or more arguments is a simple function. Aggregate functions can usually also be used as window functions.

This page last modified on 2024-06-02 10:08:16 UTC