Search and Analytics Language

Query Specification

An example of a general search query structure looks like the following:

```
SELECT * | expression[, expression[, ...]]
FROM datatype_name
[WHERE expression]
[WITH TAXONOMY taxonomy_name]
[GROUP BY [PERMUTED] alias | fieldname [, fieldname | alias | fieldname [, ...]]]
[HAVING expression]
[ORDER BY alias | fieldname [ASC | DESC[, alias | fieldname [, ...]]]
[LIMIT limit]
```

A query is a sequence of tokens. Tokens are separated by whitespace, except for non-alphanumeric operators (e.g. +, -), which are self-delimiting.

Below are examples of valid expressions:

```
SELECT time_recv +1 FROM logmsgs;
SELECT time_recv -1 FROM logmsgs;
```

Below are examples of valid expressions that have the self-delimiter + operator:

```
SELECT time_recv++1 FROM logmsgs;
SELECT time_recv+-1 FROM logmsgs;
```

A valid query must have a condition bound to the time of the message because you can only search within a specific time frame. The condition must be entered in the time field (e.g. time_recv) from the primary section of the data type (e.g. logmsgs). Some common data types and their time fields are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Time field</th>
</tr>
</thead>
<tbody>
<tr>
<td>auditmsgs</td>
<td>event_ts</td>
</tr>
<tr>
<td>Type</td>
<td>Time field</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>bdrmsgs</td>
<td>create_ts</td>
</tr>
<tr>
<td>fimdata</td>
<td>ts</td>
</tr>
<tr>
<td>hoststate</td>
<td>create_ts</td>
</tr>
<tr>
<td>logmsgs</td>
<td>time_recv</td>
</tr>
<tr>
<td>observation</td>
<td>ts</td>
</tr>
<tr>
<td>secmsgs</td>
<td>ts</td>
</tr>
<tr>
<td>telemetry</td>
<td>create_ts</td>
</tr>
<tr>
<td>udrmgs</td>
<td>create_ts</td>
</tr>
<tr>
<td>vpcflow</td>
<td>create_ts</td>
</tr>
</tbody>
</table>

A search can only be executed against one data type (e.g. `logmsgs`) and one account ID.

**Identifier**

Identifiers are used as a name for something, such as field, alias, function, taxonomy or datatypes. Identifiers matching the regular expression `^[a-zA-Z0-9_\[\]@.\-\_]*$` are specified as is. Anything else requires quoting with " where " and \ escaped with a backslash (\).

Identifiers that collide with keywords (e.g. function names) also must be quoted to avoid ambiguity.

Below are examples of identifiers:
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>Common case</td>
</tr>
<tr>
<td>&quot;foo&quot;</td>
<td>Not necessary, but valid and identical to above</td>
</tr>
<tr>
<td>&quot;from&quot;, &quot;count&quot;</td>
<td>Collides with keyword, so it must be escaped</td>
</tr>
<tr>
<td>&quot;foo.bar&quot;</td>
<td>This is not json path, but rather identifier with dot in name</td>
</tr>
<tr>
<td>&quot;&quot;&quot;</td>
<td>Exists only for completeness</td>
</tr>
</tbody>
</table>

**Constant identifier**

`false`, `true` and `null` are well-known language constants.

Strings are single quoted (with `"`), where `"` and `\` escaped by a backslash (`\`).

Numbers are `[sign]digits[.digits]` (e.g. the regular expression `/^[+-]?[0-9]+(.[0-9]+)?/`).

**Field names identifier**

Search is built as a data type agnostic service. Field identifiers are validated against a data type schema, which is defined by `ingest` service dynamically.

Using a field as part of a search query is as simple as specifying its name. There is no risk of ambiguity because every search query is executed in the context for a data type and the field name is unique within a section.

Below is an example of using field as an identifier:

```sql
SELECT ts, event_id, proto WHERE ts BETWEEN 1483228800 AND 1484265600
```

**Json-path identifier**

Indirect fields or derived fields, which return complex types (defined as `map` output), can be accessed in query with Jpath syntax, such as the example below:
SELECT ts, event_id, payload WHERE payload[0].ts BETWEEN 1483228800 AND 1484265600

An asterisk can be used in jpath fields for generating collection of child objects, like the example below:

SELECT payload[*].data WHERE ts BETWEEN 1483228800 AND 1484265600

**Other language elements**

Other tokens, used in specific constructions are below:

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>jpath expressions</td>
</tr>
<tr>
<td>,</td>
<td>Separator in SELECT, GROUP BY, ORDER BY statements and function arguments</td>
</tr>
<tr>
<td>[</td>
<td>jpath expressions</td>
</tr>
<tr>
<td>]</td>
<td>jpath expressions</td>
</tr>
<tr>
<td>(</td>
<td>function calls, expression grouping</td>
</tr>
<tr>
<td>)</td>
<td>function calls, expression grouping</td>
</tr>
</tbody>
</table>

**Expressions**

Expressions are assembled from non-keyword tokens and used in statements.

Some expression examples are below:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_name</td>
<td>Identifier is a valid expression</td>
</tr>
<tr>
<td>Expression</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>metadata.dict.dict.some_meta[123]</td>
<td>jpath expression</td>
</tr>
<tr>
<td>COUNT()</td>
<td>Function</td>
</tr>
<tr>
<td>GEOIP(source_ip, 'country')</td>
<td>Another function, with arguments</td>
</tr>
<tr>
<td>COUNT() &gt; 10 AND AVG(foo) &lt; 10</td>
<td>Functions and operators</td>
</tr>
<tr>
<td>GEOIP(parsed.tokens.src_ip).country</td>
<td>Syntactically correct, but not supported now</td>
</tr>
</tbody>
</table>

## Case sensitivity

Field names are case-sensitive, e.g. `parsed.json.foo` and `parsed.json.Foo`, are different valid field names.

All functions are case-insensitive while working with string literals or values stored in fields, e.g. `CAST(xyz, 'iNtEgEr')` and `CAST(xyz, 'integer')` are same valid names. The only exceptions to this as for now is string literals used for regular expression, e.g. `FooBar*` and `foobar*` are different valid regular expression patterns.

Alias names are case-sensitive, e.g. `somealias` and `someAlias` are different valid alias names.

Taxonomy names are case-insensitive, e.g. `SeCuRiTy` and `security` are same valid taxonomy identifiers.

## Parity exceptions

This document describes common SQL used for search and analytics queries. All queries should be compatible between these two entities, however there are some specific cases where full parity is not achieved. Such use cases are defined here.

TBD

1. **Structure**

1. **Structure - AS**
1.0.0

Specifies a name for the result of a calculation or renames a field.

| expression |   | fieldname AS identifier |

Aliases can be declared in the SELECT clause only. If a user requires special sorting or grouping by a transformed field then the aliased identifier of such transformation should be passed into other query parts (e.g. ORDER_BY, GROUP_BY or HAVING).

---

Creating Aliases

In most cases, aliases are used to specify the name of the field for display. Aliases can also be used for the results of functions and complex expressions, such as the example below:

```
SELECT foo AS bar FROM ...
SELECT SUM(foo) AS SumFoo FROM ...
SELECT (foo * 2) AS DoubledFoo FROM ...
```

If an alias contains:

- any characters other than ASCII Latin letters and digits ([a-zA-Z0-9]),
- starts with a character other than a letter,
- the alias is a keyword (SELECT, AS, FROM, WITH, TAXONOMY, WHERE, GROUP, BY, PERMUTED, HAVING, ORDER),
- or function name

then it must be enclosed in double quotes, such as the example below:

```
SELECT foo AS "My [beautiful] field!" FROM ...
SELECT foo AS "Group", COUNT(bar) AS "Count" FROM ...
```

Using names of datatype fields that were already used in the same query as an alias is prohibited. The following are examples of invalid queries:

```
SELECT foo AS foo FROM ...
```
SELECT foo, bar AS foo FROM ...

Declaring a single alias for multiple fields is an error is an invalid query:

SELECT foo AS "My Field", bar AS "My Field" FROM ...

However, aliases are case-sensitive, so the following query is valid:

SELECT foo AS "My Field", bar AS "my field" FROM ...

If you do not specify a field or expression name, a name is chosen automatically. If the expression of the field is a simple datatype field reference, then the chosen name is the same as that name of the field. In more complex cases, the system may fall back on a generated name.

Remember that you should not rely on automatically generated names, as they may be changed in the future. If you want to refer to fields or expressions, you must use aliases.

Using Aliases

Search allows the use of aliases to display in the result fields, and also in expressions of clauses SELECT, WHERE, GROUP BY / GROUP BY PERMUTED, HAVING and ORDER BY, such as the example below:

SELECT foo AS Bar FROM ... WHERE Bar > 0 ...
SELECT foo AS Bar, (Bar + Bar) AS DoubledBar FROM ...
SELECT abc AS FieldName, SUM(foo) AS SumFoo FROM ... GROUP BY FieldName ...
SELECT foo AS Bar FROM ... ORDER BY Bar
SELECT abc AS FieldName, SUM(foo) AS SumFoo FROM ... WHERE FieldName = 'Some Name' GROUP BY FieldName HAVING SumFoo > 100 ORDER BY SumFoo
SELECT foo AS "Very Special(!!!) Alias" FROM ... WHERE "Very Special(!!!) Alias" > 0 ...
Aliases can be used immediately after the declaration. For example, the following query is valid:

```sql
SELECT x AS Y, Y AS Z FROM ... WHERE Z = ...
```

And the following is an invalid query:

```sql
SELECT Y AS Z, x AS Y FROM ... WHERE Z = ...
```

---

**Partial Aliases**

If the type of expression for which the alias is specified is an object (or JSON), the alias can be used as part of another expression in WHERE and ORDER BY clauses, such as the example below:

```sql
SELECT GEOIP(foo) AS Geo FROM ... WHERE Geo.country = 'us' ...

SELECT some_json AS Root FROM ... ORDER BY Root.main_field

some_json.nested_json AS "My JSON" FROM ... WHERE "My JSON".int_value > 0 ... ORDER BY "My JSON".name
```

---

**Named expressions**

Aliases can be declared to display fields, and for use as a reference in the SELECT and WHERE clauses. Such aliases are called named expressions. To create a named expression, enclose the alias declaration in parentheses, such as the example below:

```sql
SELECT (GEOIP(foo) AS Geo).city_name FROM ... WHERE Geo.country = 'us' ...

SELECT ((foo AS MainField) * 10) AS Bar, (MainField + 100) AS Baz FROM ... WHERE MainField > 0 ...
```

Named expressions apply the same rules that apply to all aliases.

**Structure - FROM**
1.0.0
Specifies the data type for query.

FROM datatype_name

The mandatory FROM statement specifies well-known searchable data types.

FROM examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM logmsgs</td>
<td>Use logmsgs data type</td>
</tr>
<tr>
<td>FROM fimdata</td>
<td>Use fimdata data type</td>
</tr>
<tr>
<td>FROM snmsgs</td>
<td>Use snmsgs data type</td>
</tr>
<tr>
<td>FROM observation</td>
<td>Use observation data type</td>
</tr>
</tbody>
</table>

QUERY

1. Structure - GROUP BY [PERMUTED]

1.0.0
Specifies grouping for aggregated results.

[group_by [permuted] alias | fieldname [, fieldname | alias | fieldname [, ...]]]

The optional group_by statement specifies the group criteria for aggregated searches.

If group_by is not set, all final records are aggregated into single output row by default.

permuted option specifies an alternative version of group_by. Group by permuted uses array elements as group keys instead of using array values directly and grouping by full array values.

Therefore, there is a group for each unique value in array and each input row can fall into multiple groups, rather than one.

GROUP BY examples
GROUP BY field1, field2

Aggregate records that have the same field1, field2 combination

GROUP BY PERMUTED array_field1, array_field2

Aggregate records that have the full union of values from field1 and field2 combination

### QUERY

**GROUP BY**

### 1. Structure - HAVING

#### 1.0.0

[HAVING expression]

The optional `HAVING` statement specifies the condition that messages are satisfied to be returned after aggregation. If not set, all records will be filtered (`expression = true` is used).

`expression` should return a boolean value and contain only transforms functions over original field names or aliases over aggregated functions.

**HAVING examples**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT COUNT() AS cnt ... HAVING cnt &gt; 5</td>
<td>Intended usage</td>
</tr>
<tr>
<td>SELECT field ... GROUP BY field HAVING field &gt; 5</td>
<td>Not too useful, but works too</td>
</tr>
<tr>
<td>SELECT field.a GROUP BY field ... HAVING field.a.b</td>
<td>Variation of above example</td>
</tr>
</tbody>
</table>

### QUERY

**HAVING**

### 1. Structure - LIMIT

#### 1.0.0

...
Specifies maximum number of final records in results.

```
[LIMIT limit]
```

If `LIMIT` is not set, maximum value of 500000 is used as default.

### 1. Structure - ORDER BY

1.0.0

Defines the sort specification for the results.

```
[ORDER BY alias | fieldname [ASC | DESC][, alias | fieldname [, ...]]
```

The Optional `ORDER BY` statement specifies the order and sort directions of the results.

- If `ORDER BY` is not specified, natural time-specific primary key(s) are used as defaults for regular searches (for example `time_recv DESC` for log messages)
- If the `ORDER BY` direction is not set, `DESC` is used by default.

The order of messages that have the same sort key is undefined and should not be expected to be stable.

### 1. Structure - SELECT

1.0.0

Specifies fields, functions, or aggregators requested to be extracted.

```
SELECT * | expression[, expression[, ...]]
```

The mandatory `SELECT` statement specifies values you want returned. It is either `*`, or list of expressions with optional aliases.

The presence of aggregation functions (such as `SUM` or `AVG`) in expressions makes the search aggregated. The `SELECT` clause of an aggregated search can contain only aggregation functions or expressions present in `GROUP BY` statement.
See aggregators and transforms sections for exact functions specified in language.

**SELECT** examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT *</td>
<td>Return all fields, according to schema</td>
</tr>
<tr>
<td>SELECT field1, field2, metadata.dict.dict.derivedField</td>
<td>Return explicitly specified fields</td>
</tr>
<tr>
<td>SELECT field1 AS alias1</td>
<td>Rename field</td>
</tr>
<tr>
<td>SELECT &quot;Fancy named field&quot; AS &quot;Fancy alias&quot;</td>
<td>Non-trivial names require quoting</td>
</tr>
<tr>
<td>SELECT SUM(field1) AS sum_of_field1</td>
<td>Aggregation functions are another kind of expression</td>
</tr>
</tbody>
</table>

**QUERY**

**SELECT**

1. Structure - **WHERE**

1.0.0

Specifies condition for query.

```
WHERE expression
```

The optional **WHERE** statement specifies conditions that messages are satisfied to be returned during initial data extraction. If not set, all records will be extracted (i.e. **WHERE true** is used).

**expression** must return a boolean value and contain conditions on record time key for time range to be inferred. This condition cannot contain aggregation functions (as it is applied to individual record).
1. Structure - WITH TAXONOMY

1.0.0
Specifies, which token taxonomy to use.

```
[WITH TAXONOMY taxonomy_name]
```

The optional \texttt{WITH TAXONOMY} statement specifies a well-known token taxonomy name. If not set, the default taxonomy 'Default' is used.

\texttt{WITH TAXONOMY} examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{WITH TAXONOMY security}</td>
<td>Use \texttt{security} taxonomy</td>
</tr>
</tbody>
</table>

2. Transforms

2. Transforms - !=

1.0.0
\textbf{SUMMARY}
Inequality test.

\textbf{SYNOPSIS}

```
a != b
```

\texttt{a} is float|integer|map|list|string|boolean
\texttt{b} is float|integer|map|list|string|boolean

return type is boolean

\textbf{DESCRIPTION}

\texttt{!=} returns true if \texttt{a} and \texttt{b} are both non-NULL and \texttt{a} is not equal to \texttt{b}. It returns false otherwise.

\textbf{EXAMPLES}

Filter records where \texttt{port} is not equal to 22:
2. Transforms - *

1.0.0
SUMMARY
Numerical multiplication.

SYNOPSIS
a * b
a is float|integer
b is float|integer
return type is float|integer

DESCRIPTION
* performs multiplication of its two operands. If the operands differ in type then the lower will be promoted to the type of the higher, where the decreasing order of types is float, unsigned integer and signed integer. The operator returns the type of its operands or NULL if at least one of the operands is NULL or is neither an integer nor a float.

EXAMPLES
Select the product of a and b:
SELECT a * b ...

2. Transforms - +

1.0.0
SUMMARY
Numerical addition.

SYNOPSIS
a + b
a is float|integer
b is float|integer
return type is float|integer

**DESCRIPTION**

+ performs addition of its two operands. If the operands differ in type then the lower will be promoted to the type of the higher, where the decreasing order of types is float, unsigned integer and signed integer. The operator returns the type of its operands or NULL if at least one of the operands is NULL or is neither an integer nor a float.

**EXAMPLES**

Select the sum of a and b:

SELECT a + b ...

**TRANSFORMS**

+  

2. Transforms - -

1.0.0

**SUMMARY**

Numerical subtraction.

**SYNOPSIS**

a - b

a is float|integer
b is float|integer

return type is float|integer

**DESCRIPTION**

- performs subtraction of its two operands. If the operands differ in type then the lower will be promoted to the type of the higher, where the decreasing order of types is float, unsigned integer and signed integer. The operator returns the type of its operands or NULL if at least one of the operands is NULL or is neither an integer nor a float.

**EXAMPLES**
Select the difference between \(a\) and \(b\):

\[
\text{SELECT } a - b \ldots
\]

### 2. Transforms - `<`

**1.0.0**

**SUMMARY**

Less than test.

**SYNOPSIS**

\(a < b\)

\(a\) is float|integer|map|list|string|boolean

\(b\) is float|integer|map|list|string|boolean

Return type is boolean

**DESCRIPTION**

< returns true if \(a\) and \(b\) are both non-NULL and \(a\) is less than \(b\). It returns false otherwise.

**EXAMPLES**

Filter records where port is less than 22:

\[
\ldots \text{ WHERE port < 22}
\]

### 2. Transforms - `<=`

**1.0.0**

**SUMMARY**

Less than or equal to test.

**SYNOPSIS**

\(a \leq b\)
a is float|integer|map|list|string|boolean
b is float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION

\( \leq \) returns true if \( a \) and \( b \) are both non-NULL and \( a \) is less than or equal to \( b \) or if both arguments are NULL. It returns false otherwise.

EXAMPLES

Filter records where \textit{port} is less than or equal to 22:

... WHERE port \( \leq \) 22

TRANSFORMS

\( \leq \)

2. Transforms - =

1.0.0

SUMMARY

Equality test.

SYNOPSIS

\( a = b \)

\( a \) is float|integer|map|list|string|boolean
\( b \) is float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION

\( = \) returns true if \( a \) and \( b \) are both non-NULL and \( a \) is equal to \( b \) or if both arguments are NULL. It returns false otherwise.

EXAMPLES

Filter records where \textit{port} is equal to 22:

... WHERE port = 22

TRANSFORMS
2. Transforms - >

1.0.0
SUMMARY
Greater than test.

SYNOPSIS

\( a > b \)

\( a \) is float|integer|map|list|string|boolean
\( b \) is float|integer|map|list|string|boolean

return type is boolean

DESCRIPTION

\( > \) returns true if \( a \) and \( b \) are both non-NULL and \( a \) is greater than \( b \). It returns false otherwise.

EXAMPLES

Filter records where \( \text{port} \) is greater than 22:

... WHERE port > 22

2. Transforms - >=

1.0.0
SUMMARY
Greater than or equality test.

SYNOPSIS

\( a >= b \)

\( a \) is float|integer|map|list|string|boolean
\( b \) is float|integer|map|list|string|boolean

return type is boolean
DESCRIPTION

>= returns true if a and b are both non-NULL and a is greater than or equal to b or if both arguments are NULL. It returns false otherwise.

EXAMPLES

Filter records where port is greater than or equal to 22:

... WHERE port >= 22

TRANSFORMS

2. Transforms - ALL_IN

1.0.0

SUMMARY

subset test.

SYNOPSIS

set_a all_in set_b

set_a is list of float|integer|map|list|string|boolean

set_b is list of float|integer|map|list|string|boolean

return type is boolean

DESCRIPTION

ALL_IN returns true if all are of the elements in set_a are in set_b. The function returns false if either set_a or set_b is NULL or is not a list.

EXAMPLES

Filter records where all are of the members of addresses are in watchlist:

... WHERE addresses all_in watchlist

TRANSFORMS

ALL_IN

2. Transforms - AND

1.0.0
SUMMARY
Logical conjunction.

SYNOPSIS
a and b
a is boolean
b is boolean
return type is boolean

DESCRIPTION
AND returns true if both a and b are true; otherwise it returns false.

EXAMPLES
Filter records in which both a and b are true:
... WHERE a and b

TRANSFORMS

2. Transforms - ANY_IN

1.0.0
SUMMARY
set intersection test.

SYNOPSIS
set_a any_in set_b
set_a is list of float|integer|map|list|string|boolean
set_b is list of float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION
ANY_IN returns true if any are of the elements in set_a are in set_b. The function returns false if either set_a or set_b is NULL or is not a list.

EXAMPLES
Filter records where any are of the members of addresses are in watchlist:
2. Transforms - ARRAY_ALL

1.0.0
SUMMARY

universal criterion acceptance.

SYNOPSIS

array_all(list, condition)

list is list of float|integer|map|list|string|boolean
condition is boolean
return type is boolean

DESCRIPTION

ARRAY_ALL iterates over list, evaluating condition for each element. Where @ appears in condition it is substituted by the current element. If the result of condition is false, then iteration ceases and the function returns false. The function returns true if condition is always true.

The array iterator functions — ARRAY_ANY, ARRAY_ALL and ARRAY_NONE — may be nested, in which case there will be multiple "current" elements. If nesting occurs then each "current" element appears in condition as @n where n is its nesting depth, starting at 1. A single @ always refers to the current level of nesting.

The function returns false if list is NULL or is not a list.

EXAMPLES

Filter records where every element of ports is 22:

... WHERE array_all(ports, @ = 22)

If entries are a list of objects of the form

```json
{ "ports" : [ 2, 3, 5, 7, ... ], "max" : 11 }
```

then filter records for which there exists at least one object where at least one member of ports is greater than max:
2. Transforms - ARRAY_ANY

1.0.0

SUMMARY

partial criterion acceptance.

SYNOPSIS

array_any(list, condition)

list is list of float|integer|map|list|string|boolean
condition is boolean
return type is boolean

DESCRIPTION

ARRAY_ANY iterates over list, evaluating condition for each element. Where @ appears in condition it is substituted by the current element. If the result of condition is true, then iteration ceases and the function returns true. The function returns false if condition is never true.

The array iterator functions — ARRAY_ANY, ARRAY_ALL and ARRAY_NONE — may be nested, in which case there will be multiple "current" elements. If nesting occurs then each "current" element appears in condition as @n where n is its nesting depth, starting at 1. A single @ always refers to the current level of nesting.

The function returns false if list is NULL or is not a list.

EXAMPLES

Filter records where any element of ports is 22:

... WHERE array_any(ports, @ = 22)

If entries are a list of objects of the form

```json
{ "ports" : [ 2, 3, 5, 7, ... ], "max" : 11 }
```

then filter records for which there exists at least one object where at least one member of ports is greater than max:
... WHERE ARRAY_ANY(entries, ARRAY_ANY(@.ports, @ > @1.max))

**TRANSFORMS**

**ARRAY_ANY**

### 2. Transforms - ARRAY_NONE

#### 1.0.0

**SUMMARY**

partial criterion acceptance.

**SYNOPSIS**

array_none(list, condition)

- list is list of float|integer|map|list|string|boolean
- condition is boolean
- return type is boolean

**DESCRIPTION**

**ARRAY_NONE** iterates over list, evaluating condition for each element. Where @ appears in condition it is substituted by the current element. If the result of condition is true, then iteration ceases and the function returns false. The function returns true if condition is never true.

The array iterator functions — ARRAY_ANY, ARRAY_ALL and ARRAY_NONE — may be nested, in which case there will be multiple "current" elements. If nesting occurs then each "current" element appears in condition as @n where n is its nesting depth, starting at 1. A single @ always refers to the current level of nesting.

The function returns false if list is NULL or is not a list.

**EXAMPLES**

Filter records where no element of ports is 22:

... WHERE array_none(ports, @ = 22)

If entries are a list of objects of the form

```json
{ "ports" : [ 2, 3, 5, 7, ... ], "max" : 11 }
```

then filter records for which there exists at least one object where at least one member of ports is greater than max:
2. Transforms - BETWEEN

1.0.0
SUMMARY
Interval test.

SYNOPSIS
between(a, b, c)
a is float|integer|map|list|string|boolean
b is float|integer|map|list|string|boolean
c is float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION
Erratum: the synopsis above is incorrect. The correct form is a BETWEEN b AND c.

BETWEEN returns true if a is greater than or equal to b and less than c. It returns false otherwise or if any argument is NULL.

EXAMPLES
Filter records in which age is the interval [10,20]:

... WHERE ARRAY_ANY(entries, ARRAY_ANY(@.ports, @ > @.max))
2. Transforms - CAST

1.0.0

SUMMARY

Change type.

SYNOPSIS

`cast(source, destination)`

- **source** is integer|string
- **destination** is enumeration of {integer}
- return type is integer

`cast(source, destination)`

- **source** is string
- **destination** is enumeration of {ip}
- return type is string

`cast(source, destination)`

- **source** is integer|string
- **destination** is enumeration of {string}
- return type is string

DESCRIPTION

`cast` transforms **source** to the **destination** type. If the first argument is NULL or does not match the expected type, then the function returns NULL. The supported values for **destination** are:

<table>
<thead>
<tr>
<th>Destination</th>
<th>Type</th>
<th>Transformation</th>
<th>Notes</th>
</tr>
</thead>
</table>

... WHERE age BETWEEN 10 AND 20

TRANSFORMS

BETWEEN
<table>
<thead>
<tr>
<th>integer</th>
<th>integer</th>
<th>A decimal string to a signed, 64-bit integer.</th>
<th>If the resulting value underflow or overflow, then NULL is returned. If the argument is an integer, then no transformation is applied.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>string</td>
<td>A IPv4 or IPv6 address, optionally base64-encoded, to a canonical form.</td>
<td>If the input is not a valid address, then NULL is returned.</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>A signed, 64-bit integer to a decimal string representation.</td>
<td>If the argument is a string, then no transformation is applied.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

Filter records where `address` is equivalent to 1.2.3.4:

```sql
... WHERE cast(address, 'ip') = '1.2.3.4'
```

**TRANSFORMS**

**CAST**

### 2. Transforms - CHAR_LENGTH

**1.0.0**

**SUMMARY**

Return string length.

**SYNOPSIS**

`char_length(text)`

*text* is string

return type is integer

**DESCRIPTION**

`CHAR_LENGTH` returns the number of characters in `text`. It returns NULL if `text` is not a string or is NULL.

**EXAMPLES**

Filter records where the length of `password` is less than eight:
... WHERE char_length(password) < 8

**TRANSFORMS**

**CHAR_LENGTH**

2. Transforms - CIDR_MATCH

1.0.0  
**SUMMARY**

Subnet membership test.

**SYNOPSIS**

cidr_match(address, CIDR)

*address* is list|string  
*CIDR* is static list of string|string  
return type is boolean

**DESCRIPTION**

CIDR_MATCH returns true if *address* belongs to the network or list of networks specified by *CIDR*. *CIDR*, or each of its elements if a list, is a string in CIDR notation and must be an expression that can be evaluated at query compilation. It cannot, for example, be a field even if the field were to contain a valid CIDR specification.

**EXAMPLES**

Filter records where *address* is in the subnet 131.111.96.0/20:

... WHERE cidr_match(address, '131.111.96.0/20')

Filter records where *address* is in any of the subnets in a tag set value:

... WHERE cidr_match(address, TAGS:LOOKUP('CidrSet', 'watchlist'))

**TRANSFORMS**

CIDR_MATCH

2. Transforms - COALESCE

1.0.0  
**SUMMARY**
Return the first non-NULL entry.

**SYNOPSIS**

`coalesce(list)`

`list` is list of float|integer|map|list|string|boolean

return type is float|integer|map|list|string|boolean

**DESCRIPTION**

`COALESCE` evaluates each member of `list`, stopping at, and returning, the first non-NULL result. It returns NULL if `list` is NULL or if every one of its elements is NULL.

**EXAMPLES**

Filter records in which the first extant field of "user", "username" and "login" is equal to 'admin':

... WHERE coalesce([user, username, login]) = 'admin'

**TRANSFORMS**

`COALESCE`

### 2. Transforms - CONCAT

**1.0.0**

**SUMMARY**

Concatenate string elements.

**SYNOPSIS**

`concat(list)`

`list` is list of float|integer|map|list|string|boolean

return type is string

**DESCRIPTION**

`CONCAT` returns the string formed by concatenating all string elements found in `list`. The function returns NULL if `list` is NULL.

**EXAMPLES**

Select all strings in `users` as a single field:

SELECT concat(users) ...
2. Transforms - CONCAT_WS

1.0.0

SUMMARY

Concatenate string elements.

SYNOPSIS

`concat_ws(list, delimiter)`

- `list` is list of float|integer|map|list|string|boolean
- `delimiter` is static string
- return type is string

DESCRIPTION

`CONCAT_WS` returns the string formed by concatenating all string elements found in `list` using `delimiter`. The function returns NULL if `list` is NULL.

EXAMPLES

Select all strings in `users` as a single field and delimit using ",":

SELECT concat_ws(users, ',') ...

2. Transforms - CONTAINS

1.0.0

SUMMARY

Single substring test.

SYNOPSIS

`haystack contains needle`

- `haystack` is list|string
- `needle` is static string
return type is boolean

**DESCRIPTION**

**CONTAINS** returns true if **haystack** contains **needle**. The function returns NULL if **haystack** is NULL.

**EXAMPLES**

Filter records in which **message** contains the string "denied":

```plaintext
... WHERE message contains 'denied'
```

**TRANSFORMS**

**CONTAINS**

2. Transforms - **CONTAINS_ALL**

**1.0.0**

**SUMMARY**

Universal substring test.

**SYNOPSIS**

**haystack** **contains_all** **needles**

- **haystack** is list|string
- **needles** is static list of string
- return type is boolean

**DESCRIPTION**

**CONTAINS_ALL** returns true if **haystack** contains every element of **needles** and false otherwise. The function returns NULL if **haystack** is NULL.

**EXAMPLES**

Filter records in which **message** contains every of the strings "denied", "refused" and "failed":

```plaintext
... WHERE message contains_all ['denied', 'refused', 'failed']
```

**TRANSFORMS**

**CONTAINS_ALL**

2. Transforms - **CONTAINS_ANY**
1.0.0
SUMMARY
Partial substring test.

SYNOPSIS
haystack contains_any needles
haystack is list|string
needles is static list of string
return type is boolean

DESCRIPTION
CONTAINS_ANY returns true if haystack contains any element of needles and false otherwise. The function returns NULL if haystack is NULL.

EXAMPLES
Filter records in which message contains any of the strings "denied", "refused" and "failed":

... WHERE message contains_any ['denied', 'refused', 'failed']

2. Transforms - DECODE

1.0.0
SUMMARY
Decodes encoded binary data.

SYNOPSIS
decode(source, format)
source is string
format is enumeration of {base64}
return type is string

DESCRIPTION
DECODE transforms source based on the specified format. If the first argument is NULL, then the function returns NULL. The supported values for format are: base64.
EXAMPLES
Select the result of decoding the base64-encoded string `encoded`:

```sql
SELECT decode(encoded, 'base64')
```

2. Transforms - ENDS_WITH

1.0.0

SUMMARY
String suffix test.

SYNOPSIS
```
ends_with(text, suffix)
```

`text` is list|string

`suffix` is list of string|string

return type is boolean

DESCRIPTION
`ENDS_WITH` returns true if `suffix` is a string and `text` ends with `suffix` or if `suffix` is a list and `text` ends with any of its elements. The function otherwise returns NULL.

EXAMPLES
Filter records in which `message` ends with 'foo':

```sql
... WHERE ends_with(message, 'foo')
```

2. Transforms - EXISTS

1.0.0

SUMMARY
Test for presence.

SYNOPSIS
exists`(argument)`

`argument` is float|integer|map|list|string|boolean

return type is boolean

**DESCRIPTION**

exists returns true if `argument` is a field and is present, or if `argument` is an expression and is true. It returns false otherwise.

**EXAMPLES**

Filter records where `field` is present:

```none
... WHERE exists(field)
```

**TRANSFORMS**

**EXISTS**

2. Transforms - FROM_EPOCHTIME

**1.0.0**

**SUMMARY**

Format date and time.

**SYNOPSIS**

`from_epochtime(time, format)`

`time` is integer

`format` is static string

return type is string

**DESCRIPTION**

FROM_EPOCHTIME produces a string representation of `time`, the number of seconds since the UNIX epoch, according to the specification in `format`. Characters in the `format` string are copied to the output one at a time until a `%` is encountered. A `%` indicates that the following letter is a conversion specification and the two characters are substituted by the corresponding translation from the table below. The use of any other character following a `%` is an error.

If `time` is NULL or is not an integer, then the function returns NULL.
<table>
<thead>
<tr>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>month name in full (January to December)</td>
</tr>
<tr>
<td>F</td>
<td>full date (%Y-%m-%d)</td>
</tr>
<tr>
<td>H</td>
<td>hour (00 to 23)</td>
</tr>
<tr>
<td>M</td>
<td>minutes (00 to 59)</td>
</tr>
<tr>
<td>S</td>
<td>seconds (00 to 59)</td>
</tr>
<tr>
<td>T</td>
<td>time in 24 hour format (hh:mm:ss)</td>
</tr>
<tr>
<td>W</td>
<td>weekday name in full (Sunday to Saturday)</td>
</tr>
<tr>
<td>Y</td>
<td>year as a numeric, 4-digit value</td>
</tr>
<tr>
<td>a</td>
<td>abbreviated weekday name (Sun to Sat)</td>
</tr>
<tr>
<td>b</td>
<td>abbreviated month name (Jan to Dec)</td>
</tr>
<tr>
<td>d</td>
<td>month day of as a numeric value (01 to 31)</td>
</tr>
<tr>
<td>h</td>
<td>hour (01 to 12)</td>
</tr>
<tr>
<td>j</td>
<td>day of the year (001 to 366)</td>
</tr>
<tr>
<td>m</td>
<td>month name as a numeric value (01 to 12)</td>
</tr>
<tr>
<td>p</td>
<td>AM or PM</td>
</tr>
<tr>
<td>r</td>
<td>time as 12 hour AM/PM (hh:mm:ss AM/PM)</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>w</td>
<td>weekday where Sunday=0 and Saturday=6</td>
</tr>
<tr>
<td>y</td>
<td>year as a numeric, 2-digit value</td>
</tr>
</tbody>
</table>

**EXAMPLES**

Select the time of day corresponding to `time_recv`:

```sql
SELECT from_epochtime(time_recv, '%H:%M') ...
```

Filter records where `time_recv` corresponds to a Wednesday:

```sql
... WHERE from_epochtime(time_recv, '%W') = 'wednesday'
```

**TRANSFORMS**

**FROM_EPOCHTIME**

**2. Transforms - GEOIP**

**1.0.0
**

**SUMMARY**

geographical network address translation

**SYNOPSIS**

```sql
geoip(address)
```

*address* is string

return type is map

```sql
geoip(address, attribute)
```

*address* is string

*attribute* is enumeration of `{city_name,country_name,country_iso_code,country_name_rr,country_iso_code_rr,aso}`

return type is string

```sql
geoip(address, attribute)
```

*address* is string

*attribute* is enumeration of `{asn}`
DESCRIPTION

In its single-argument form, GEOIP transforms the internet address into a map where each key is a geographical attribute from the table below and the corresponding value is the translation of address. If GEOIP is given attribute as a second argument, then it returns the specific translation as a scalar value. If address is NULL or if no suitable translation exists, then the function returns NULL.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>city_name</td>
<td>string</td>
<td>the city name</td>
</tr>
<tr>
<td>country_name</td>
<td>string</td>
<td>the country name</td>
</tr>
<tr>
<td>country_iso_code</td>
<td>string</td>
<td>the country ISO code</td>
</tr>
<tr>
<td>country_name_rr</td>
<td>string</td>
<td>the name of the represented/registered/geographic country</td>
</tr>
<tr>
<td>country_iso_code_rr</td>
<td>string</td>
<td>the ISO code of the represented/registered/geographic country</td>
</tr>
<tr>
<td>asn</td>
<td>integer</td>
<td>the Autonomous System Number</td>
</tr>
<tr>
<td>aso</td>
<td>string</td>
<td>the Autonomous System Organisation</td>
</tr>
</tbody>
</table>

EXAMPLES

Select all geographical information for address:

SELECT geoip(address)...

Select the city name and country ISO code corresponding to address:

SELECT (geoip(address) AS geoip).city_name, geoip.country_iso_code ...

Filter records where the country corresponding to address is the United States:

... WHERE geoip(address, 'country_iso_code') = 'US'

TRANSFORMS
2. Transforms - IF

1.0.0

SUMMARY

Conditional evaluation.

SYNOPSIS

if(condition, then, else)

condition is boolean

then is float|integer|map|list|string|boolean

else is float|integer|map|list|string|boolean

return type is float|integer|map|list|string|boolean

DESCRIPTION

if evaluates the Boolean expression condition. If the result is true it evaluates and returns the result of then, otherwise it evaluates and returns the result of else.

EXAMPLES

Select the field to project as "destination" according to whether the protocol is TCP:

SELECT if(proto = 6, tcp.port, ip_dest) AS destination ...

2. Transforms - IN

1.0.0

SUMMARY

Set membership test.

SYNOPSIS

element in set

element is integer|map|string|boolean
**DESCRIPTION**

**IN** returns true if the scalar element (which may be NULL) is a member of the list set. It returns false otherwise.

**EXAMPLES**

Filter records where address is in watchlist:

... WHERE address in watchlist

**TRANSFORMS**

**IN**

## 2. Transforms - INTERVAL

**1.0.0**

**SUMMARY**

Round time down to interval.

**SYNOPSIS**

interval\((time, interval)\)

time is integer

interval is integer

return type is integer

**DESCRIPTION**

INTERVAL returns the result of rounding down time to the nearest multiple of interval. The function returns NULL if either argument is NULL or if interval is zero.

**EXAMPLES**

Select the start of the hour corresponding to time_recv:

SELECT interval(time_recv, 3600) ...
2. Transforms - ISNULL

1.0.0
SUMMARY
Test for absence.

SYNOPSIS
isnull(argument)

argument is float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION
isnull returns true if argument is a field and is absent or if argument is an expression and is false. It returns false otherwise.

EXAMPLES
Filter records where field is absent:

... WHERE isnull(field)

2. Transforms - IS_INTERNAL_ADDRESS

1.0.0
SUMMARY
Internal network membership test.

SYNOPSIS
is_internal_address(address)

address is string
return type is boolean

DESCRIPTION
IS_INTERNAL_ADDRESS returns true if address is an IP address on an internet-internal network and false otherwise. The following network ranges are considered internal:
<table>
<thead>
<tr>
<th>Address</th>
<th>Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.0/8</td>
<td></td>
</tr>
<tr>
<td>100.64.0.0/10</td>
<td></td>
</tr>
<tr>
<td>127.0.0.0/8</td>
<td></td>
</tr>
<tr>
<td>169.254.0.0/16</td>
<td></td>
</tr>
<tr>
<td>172.16.0.0/12</td>
<td></td>
</tr>
<tr>
<td>192.168.0.0/16</td>
<td></td>
</tr>
<tr>
<td>::1/128</td>
<td></td>
</tr>
<tr>
<td>fc00::/7</td>
<td></td>
</tr>
<tr>
<td>fe80::/10</td>
<td></td>
</tr>
</tbody>
</table>

**EXAMPLES**

Filter records where `address` is on an internal network:

```sql
... WHERE is_internal_address(address)
```

**TRANSFORMS**

**IS_INTERNAL_ADDRESS**

**2. Transforms - IS_NOT_NULL**

**1.0.0 SUMMARY**

Test for presence.

**SYNOPSIS**

- **argument** is _null_

- **argument** is float|integer|map|list|string|boolean
return type is boolean

DESCRIPTION

`is_not_null` returns true if `argument` is a field and is present or if `argument` is an expression and is true. It returns false otherwise.

EXAMPLES

Filter records where field is present:

```
... WHERE field is_not_null
```

TRANSFORMS

`IS_NOT_NULL`

2. Transforms - `IS_NULL`

1.0.0

SUMMARY

Test for absence.

SYNOPSIS

`argument is_null`

`argument` is float|integer|map|list|string|boolean

return type is boolean

DESCRIPTION

`is_null` returns true if `argument` is a field and is absent or if `argument` is an expression and is false. It returns false otherwise.

EXAMPLES

Filter records where field is absent:

```
... WHERE field is_null
```

TRANSFORMS

`IS_NULL`

2. Transforms - `LENGTH`

1.0.0
**SUMMARY**
List length.

**SYNOPSIS**

length(list)

*list* is list of float|integer|map|list|string|boolean

return type is integer

**DESCRIPTION**

LENGTH returns the number of elements in *list*. It returns NULL if *list* is NULL or is not a list.

**EXAMPLES**

Select the length of the list *addresses*:

SELECT length(addresses) ...

**TRANSFORMS**

LENGTH

### 2. Transforms - LIKE

#### 1.0.0

**SUMMARY**
Simple pattern matching.

**SYNOPSIS**

string like pattern

string is list|string

pattern is static string

return type is boolean

**DESCRIPTION**

LIKE returns true if *string* is non-NULL and matches the case-insensitive *pattern*. It returns false otherwise. Pattern matching characters are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These special characters must otherwise be escaped with \\".

**EXAMPLES**

Filter records where program contains the string "foo":

... WHERE program like '%FOO%'

**TRANSFORMS**

**LIKE**

### 2. Transforms - LOWER

1.0.0

**SUMMARY**

Convert string to lowercase.

**SYNOPSIS**

`lower(text)`

text is string

return type is string

**DESCRIPTION**

`lower` converts upper-case characters in text to their lower-case equivalents and returns the result.

**EXAMPLES**

Select the lower-case equivalent of name:

SELECT lower(name)

**TRANSFORMS**

**NONE_IN**

### 2. Transforms - NONE_IN
1.0.0
SUMMARY
disjoint set test.

SYNOPSIS

set_a none_in set_b

set_a is list of float|integer|map|list|string|boolean
set_b is list of float|integer|map|list|string|boolean

return type is boolean

DESCRIPTION

NONE_IN returns true if none is of the elements in set_a are in set_b. The function returns false if either set_a or set_b is NULL or is not a list.

EXAMPLES

Filter records where none is of the members of addresses are in watchlist:

... WHERE addresses none_in watchlist

2. Transforms - NOT

1.0.0
SUMMARY

Logical negation.

SYNOPSIS

not a

a is boolean

return type is boolean

DESCRIPTION

NOT returns true if a is non-NULL and false, otherwise it returns false.

EXAMPLES

Filter records in which a is non-NULL and false:
2. Transforms - NOW

1.0.0

SUMMARY

The current time.

SYNOPSIS

now()

return type is integer

DESCRIPTION

NOW returns the current time, expressed as the integer number of seconds since the start of the UNIX epoch.

EXAMPLES

Filter records where time_recv is later than one day ago:

... WHERE time_recv > now() - 86400

2. Transforms - OR

1.0.0

SUMMARY

Logical disjunction.

SYNOPSIS

a or b

a is boolean

b is boolean

return type is boolean
**DESCRIPTION**

OR returns true if at least one of a and b is true, otherwise it returns false.

**EXAMPLES**

Filter records in which at least one of a and b is true:

... WHERE a or b

---

**2. Transforms - REGEXP_EXTRACT**

**1.0.0**

**SUMMARY**

Regular expression extraction.

**SYNOPSIS**

`regexp_extract(string, regex)`

*string* is string

*regex* is static string

return type is map

**DESCRIPTION**

`REGEXP_EXTRACT` returns map if *string* is non-NULL and matches *regex*, otherwise it returns NULL.

Returned map contains results only for first matched substring. It has one pair per each named capture group, where key is a name of the group, and value is an extracted group or NULL if it is not present.

*regex* must have at least one named capture group. Names may be up to 32 symbols long and they may contain only ASCII alphanumeric characters and underscores but must start with a non-digit. Names must be unique within *regex*.

See the [documentation for the underlying engine](#) for the specification of *regex*.

**EXAMPLES**

Select extracted fields from *message*:

SELECT regexp_extract(message, 'Login from: (?<ip>([0-9][1,3]\.)[0-9][0-9]{1,3})').ip as ip ...
Select extracted user, ip from message and aggregate by them:

```sql
SELECT (regexp_extract(message, 'User (?<user>[A-Za-z0-9]+) has logged in from (?<ip>([0-9]{1,3}\.(?:[0-9]{1,3}){3}))') AS res).user AS user, res.ip as ip, Count(*) WHERE res IS NOT NULL
GROUP BY user, ip
```

---

### 2. Transforms - REGEXP_EXTRACT

#### 1.0.0

**SUMMARY**

Regular expression matching.

**SYNOPSIS**

```
string regexp_extract string
regexp_match regex
```

`string` is list|string

`regexp` is static string

return type is boolean

**DESCRIPTION**

`REGEXP_EXTRACT` returns true if `string` is non-NULL and matches `regexp`. See the documentation for the underlying engine for the specification of `regexp`.

**EXAMPLES**

Filter records where `program` contains either of the strings "foo" and "bar":

```sql
... WHERE program regexp_match 'foo|bar'
```

---

### 2. Transforms - REGEXP_MATCH

#### 1.0.0

**SUMMARY**

Regular expression matching.

**SYNOPSIS**

```
string regexp_match regex
```

`string` is list|string

`regexp` is static string

return type is boolean

**DESCRIPTION**

`REGEXP_MATCH` returns true if `string` is non-NULL and matches `regexp`. See the documentation for the underlying engine for the specification of `regexp`.

**EXAMPLES**

Filter records where `program` contains either of the strings "foo" and "bar":

```sql
... WHERE program regexp_match 'foo|bar'
```

---

### 2. Transforms - SPLIT

#### 1.0.0

**SUMMARY**

Split a string.
SYNOPSIS

\texttt{split(text, delimiter)}

- \texttt{text} is string
- \texttt{delimiter} is static string
- return type is list of string

DESCRIPTION

\texttt{split} returns the list of strings created by separating \texttt{text} using one of the characters in \texttt{delimiter}. The function returns NULL if \texttt{text} is NULL.

EXAMPLES

Select the list of colon-delimited fields in \texttt{passwd}:

\begin{verbatim}
SELECT split(passwd, ':') ...
\end{verbatim}

TRANSFORMS

2. Transforms - STARTS_WITH

1.0.0

SUMMARY

String prefix test.

SYNOPSIS

\texttt{starts_with(text, prefix)}

- \texttt{text} is list|string
- \texttt{prefix} is list of string|string
- return type is boolean|string

DESCRIPTION

\texttt{starts_with} returns true if \texttt{prefix} is a string and \texttt{text} starts with \texttt{prefix} or if \texttt{prefix} is a list and \texttt{text} starts with any of its elements. The function otherwise returns NULL.

EXAMPLES

Filter records in which \texttt{message} starts with 'foo':

\begin{verbatim}
... WHERE starts_with(message, 'foo')
\end{verbatim}
2. Transforms - TAGS:EXISTS

1.0.0
SUMMARY
Tag set key test.

SYNOPSIS
tagset:exists(tagset, key)
tagset is static string
key is integer|string
return type is boolean

DESCRIPTION
TAGS:EXISTS returns true if tagset contains key, otherwise, it returns false.

EXAMPLES
Filter records in which the tag set 'Users' contains the key 'watchlist':
... WHERE tags:exists('Users', 'watchlist')

2. Transforms - TAGS:LOOKUP

1.0.0
SUMMARY
Return tag set value.

SYNOPSIS
tagset:lookup(tagset, key)
tagset is static string
key is integer|string
return type is float|integer|map|list|string|boolean

tags:lookup(tagset, key, default)
tagset is static string
key is integer|string
default is float|integer|map|list|string|boolean
return type is float|integer|map|list|string|boolean

DESCRIPTION
TAGS:LOOKUP returns the value associated with key in tagset. If key is NULL or not present then the function will return default, if provided, or NULL if not.

EXAMPLES
Filter records in which address is in the list associated with the key 'allowed' in the tag set 'watchlists':
... WHERE address IN tags:lookup('watchlists', 'allowed')

TRANSFORMS

TAGS: LOOKUP

2. Transforms - TO_EPOCHTIME

1.0.0
SUMMARY
Convert date/time string to UNIX time.

SYNOPSIS
to_epochtime(date/time, format)
date/time is string
format is enumeration of {ISO8601}
return type is integer

DESCRIPTION
TO_EPOCHTIME transforms a formatted string containing a date and time into the corresponding number of seconds since the UNIX epoch. The supported values for format are ISO8601.

EXAMPLES
Interpret _date_ as an ISO8601 string and select the equivalent Unix time:

SELECT to_epochtime(_date_, 'ISO8601') ...

2. Transforms - TUNE:THRESHOLD

1.0.0
SUMMARY
Return tag set value.

SYNOPSIS

\texttt{tune:threshold(tagset, key)}

\texttt{tagset} is static string
\texttt{key} is static string

return type is integer|list of integer|string|boolean|string|boolean

DESCRIPTION

\textbf{TUNE:THRESHOLD} returns the value associated with \texttt{key} in \texttt{tagset}. It returns NULL if \texttt{key} is NULL.

EXAMPLES

Filter records in which \texttt{score} is greater than the value associated with 'cvss' in the tag set 'tuneables':

... WHERE score > tune:threshold('tuneables', 'cvss')

2. Transforms - URI_PARSE

1.0.0
SUMMARY
Parse a URI.

SYNOPSIS
**uri_parse** *(uri)*

*uri* is string

return type is map

**DESCRIPTION**

**URI_PARSE** parses *uri* and returns a map whose values are the fields within it; if *uri* is NULL or cannot be parsed successfully then the function returns NULL. The map's keys are the following:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value type</th>
<th>Value notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>string</td>
<td>NULL if <em>uri</em> is relative, i.e. if it starts with a path.</td>
</tr>
<tr>
<td>authority</td>
<td>string</td>
<td>The host and, optionally, the user information and port. NULL if absent.</td>
</tr>
<tr>
<td>path</td>
<td>string</td>
<td>May be an empty (zero-length) string.</td>
</tr>
<tr>
<td>query</td>
<td>string</td>
<td>NULL if absent.</td>
</tr>
<tr>
<td>query_kvs</td>
<td>map</td>
<td>A map whose keys and values represent the unescaped parameters in the query string. NULL if absent.</td>
</tr>
<tr>
<td>fragment</td>
<td>string</td>
<td>NULL if absent.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

Select URI path from *uri*:

SELECT uri_parse(uri).path ...

Filter records where the query string in *uri* contains the key/value pairs "action=login" and "type=user":

... WHERE (uri_parse(uri).query_kvs AS qs).action = 'login' AND qs.type = 'user'

**TRANSFORMS**
2. Transforms - WINDOW

1.0.0
SUMMARY
Sliding window time interval.

SYNOPSIS
window(time, window)

* time is integer
* window is integer
* return type is integer

DESCRIPTION
In search, WINDOW behaves exactly like INTERVAL: it returns the result of rounding down time to the nearest multiple of window. In Log Correlation and Streaming analytics, window is a sliding interval. The function returns NULL if either argument is NULL or if window is zero.

EXAMPLES
Select the start of the hour corresponding to time_recv:

SELECT window(time_recv, 3600) ...

3. Aggregators

3. Aggregators - AVG

1.0.0
SUMMARY
Returns the average of a numeric value for a group.

SYNOPSIS
DESCRIPTION
EXAMPLES
3. Aggregators - COUNT

1.0.0
SUMMARY
Returns number of non-null elements in a group.

SYNOPSIS
DESCRIPTION
EXAMPLES

3. Aggregators - LSET

1.0.0
SUMMARY
Returns set of sorted unique non-null elements for each aggregated group limited up to specified value.

SYNOPSIS
DESCRIPTION
EXAMPLES

3. Aggregators - LU_COUNT

1.0.0
SUMMARY
Returns number of unique non-null elements in a group limited by specified value.
3. Aggregators - MAX

1.0.0
SUMMARY

Returns the maximum of a numeric value for a group.

SYNOPSIS
DESCRIPTION
EXAMPLES

3. Aggregators - MIN

1.0.0
SUMMARY

Returns the minimum of a numeric value for a group.

SYNOPSIS
DESCRIPTION
EXAMPLES

3. Aggregators - SET

1.0.0
SUMMARY
Returns set of unique non-null elements for each aggregated group.

SYNOPSIS
DESCRIPTION
EXAMPLES
AGGREGATORS

SET

3. Aggregators - SUM

1.0.0
SUMMARY
Returns the sum of a numeric value for a group.

SYNOPSIS
DESCRIPTION
EXAMPLES
AGGREGATORS

SUM

3. Aggregators - UCOUNT

1.0.0
SUMMARY
Returns number of non-null elements in a group.

SYNOPSIS
DESCRIPTION
EXAMPLES
AGGREGATORS

UCOUNT