Cortex XDR 3.x Documentation

Stages

- alter
- arrayexpand
- bin
- call
- comp
- config
  - case_sensitive
  - timeframe
- dedup
- fields
- filter
- getrole
- iploc
- join
- limit
- replacenull
- sort
- Tag
- target
- top
- transaction
- union
- view
- windowcomp

Functions

- add
- approx_count
- approx_quantiles
- approx_top
- array_all

array_any
arrayconcat
arraycreate
arraydistinct
arrayfilter
arrayindex
arrayindexof
array_length
arraymap
arraymerge
arrayrange
arraystring
avg
coalesce
concat
convert_from_base_64
count
count_distinct
current_time
date_floor
divide
earliest
extract_time
extract_url_host
extract_url_pub_suffix
extract_url_registered_domain
first
first_value
floor
format_string
format_timestamp
if
incidr
incidr6
incidrlist
int_to_ip
ip_to_int
is_ipv4
is_known_private_ipv4
is_ipv6
is_known_private_ipv6
json_extract
json_extract_array
json_extract_scalar
json_extract_scalar_array
lag
last
last_value
latest

len
list
lowercase
ltrim, rtrim, trim
max
median
min
multiply
object_merge
object_create
parse_epoch
parse_timestamp
pow
rank
regexcapture
regextract
replace
replex
round
row_number
split
stddev_population
stddev_sample
string_count
subtract
sum
time_frame_end
timestamp_diff
timestamp_seconds
to_boolean
to_epoch
to_float
to_integer
to_json_string
to_number
to_string
to_timestamp
uppercase
values
var

Stages

Abstract

Learn more about the Cortex Query Language supported stages.

Stages perform certain operations in evaluating queries. For example, the `dataset` stage specifies a dataset to run the query. Commonly used stages include `dataset`, `fields`, `filters`, `join`, and `sort`. The stages supported in Cortex Query Language are detailed below.

alter

Abstract

Learn more about the Cortex Query Language `alter` stage.

Review the following topic:

- Understanding string manipulation in XQL

Syntax

```
alter <field1> = <function value1> [, <field2> = <function value2>, ...]
```

Description

The `alter` stage is used to change the values of an existing field (column) or to create a new field (column) based on constant values or existing fields (columns). The `alter` stage does this by assigning a value to a field name based on the returned value of the specified function. The field does not have to be known to the dataset or preset schema that you are querying. Further, you can overwrite the current value for a known field using this stage.

After defining a field using the `alter` stage, you can apply other stages, such as filtering, to the new field or field value.

Examples

Given three username fields, use the coalesce function to return a username value in the `default_username` field, making sure to never have a `default_username` that is `root`.

```
dataset = xdr_data
| fields actor_primary_username,
        os_actor_primary_username,
        causality_actor_primary_username
| alter default_username = coalesce(actor_primary_username,
                                    os_actor_primary_username,
                                    causality_actor_primary_username)
| filter default_username != "root"
```

arrayexpand

Abstract

Learn more about the Cortex Query Language `arrayexpand` stage.

Syntax

arrayexpand <array_field> [limit <limit number>]

Description

The `arrayexpand` stage expands the values of a mulit-value array field into separate events and creates one record in the result set for each item in the array, up to a `<limit number>` of records.

Example

Suppose you have a dataset with a single row like this:

uid	username	array_values
123456	ajohnson	[1,2,3,4,5,6,7,8,9,0]

Then if you run an `arrayexpand` stage using the `array_values` field, with a limit of 3, the result set includes the following records:

```
dataset=my_dataset
| arrayexpand array_values limit 3
```

uid	username	array_values
123456	ajohnson	2
123456	ajohnson	1
123456	ajohnson	3

The result records created by `arrayexpand` are in no particular order. However, you can use the sort stage to sort the results:

```
dataset=my_dataset
```

```
| arrayexpand array_values
| sort asc array_values
```

bin

Abstract

Learn more about the Cortex Query Language `bin` stage to group events by quantity or time span.

Syntax

- **Quantity**
  bin <field> bins = <number>
- **Time Span**
  bin <field> span = <time> [timeshift = <epoch time> [timezone = "*<time zone>*"]]

Description

The `bin` stage enables you to group events by quantity or time span. The most common use case is for timecharts.

You can add the `bin` stage to your queries using two different formats depending on whether you are grouping events by quantity or time span. Currently, the `bin` stage is only supported using the equal sign (=) operator in your queries without any boolean operators (`and`, `or`).

When you group events of a particular field by quantity, the `bin` stage is used with `bins` to define how to divide the events.

When you group events of a particular field by time, the `bin` stage is used with `span = <time>`, where `<time>` is a combination of a number and time suffix. Set one time suffix from the list of available options listed in the table below. In addition, you can define a particular start time for grouping the events in your query according to the Unix epoch time by setting `timeshift = <epoch time> timezone = "<time zone>"`, which are both optional. You can configure the `<time zone>` offset using an hours offset, such as `“+08:00”`, or using a time zone name from the List of Supported Time Zones, such as `"America/Chicago"`. The query still runs without defining the epoch time or time zone. If no `timeshift = <epoch time> timezone = "<time zone>"` is set, the query runs according to last time set in the log.

When you group events by quantity, the `<field>` in the `bin` stage must be a number, and when you group by time, the `<field>` must be a date type. Otherwise, your query will fail.

Time Suffixes

Time Suffix	Description
MS	milliseconds
S	seconds
M	minutes
H	hours
D	days
W	weeks
MO	months
Y	years

The time suffix is not case sensitive.

Examples

Quantity Example

Return a maximum of 1,000 `xdr_data` records with the events of the `action_total_upload` field grouped by 50MB. Records with the `action_total_upload` value set to 0 or null are not included in the results.

```
dataset = xdr_data
| filter action_total_upload != 0 and action_total_upload != null
| bin action_total_upload bins = 50
| limit 1000
```

-
- **Time Span Examples**

With a time zone configured using an hours offset:
Return a maximum of 1,000 `xdr_data` records with the events of the `_time` field grouped by 1-hour increments starting from the epoch time `1615353499`, and includes a time zone using an hours offset of `“+08:00”`.

```
dataset = xdr_data
| bin _time span = 1h timeshift = 1615353499 timezone = “+08:00”
| limit 1000
```

-

With a time zone name configured:
Return a maximum of 1,000 `xdr_data` records with the events of the `_time` field grouped by 1-hour increments starting from the epoch time `1615353499`, and includes an `"America/Los_Angeles"` time zone.

```
dataset = xdr_data
| bin _time span = 1h timeshift = 1615353499 timezone = “America/Los_Angeles”
```

- | limit 1000

call

Abstract

Learn more about the Cortex Query Language `call` stage to reference a predefined query from the Query Library.

Syntax

```
call "<name of predefined query>" [<param_name1> = <value1> <param_name2> = <value2>....]
```

Description

The `call` stage is used to reference a predefined query from the Query Library, including your Personal Query Library. In addition, if your query includes parameters you can reference them in the `call` stage using the syntax `<param_name1> = <value1> <param_name2> = <value2>...`. When using parameters in your `call` stage, you need to ensure that a query already exists that uses these parameters.

Example without Parameters

For the predefined query called "CreateRole operation parsed to fields", returns a maximum of 100 records, where the `accessKeyId` equals "1234".

```
call "CreateRole operation parsed to fields"
| filter accessKeyId = "1234"
| limit 100
```

Example with Parameters

Using the same example above, this example shows how to use the same `call` stage with parameters. This example assumes that there is a query that is already saved with a parameter `$key_id = "1234"`.

Saved query:

```
dataset = dataset_name
| filter field_name = $key_id
```

Query to run with using parameters:

```
call "CreateRole operation parsed to fields" key_id = "1234"
| limit 100
```

comp

Abstract

Learn more about the Cortex Query Language `comp` stage that precedes functions calculating statistics.

Syntax

```
comp <aggregate function1> (<field>) [as <alias>][,<aggregate function2>(<field>) [as <alias>],...] [by <field1>[,<field2>...]]
[addrawdata = true|false [as <target field>]]
```

Description

The `comp` stage precedes functions calculating statistics for results to compute values over a group of rows and return a single result for a group of rows.

- Aggregation functions, such as `sum`, `min`, and `max`
- Approximate aggregate functions, such as `approx_count` or `approx_top`

At least one of the comp aggregate functions or comp approximate aggregate functions must be used. Yet, it's also possible to define a comp stage with both types of aggragate functions.

Use approximate aggregate functions to produce approximate results, instead of exact results used with regular aggregate functions, which are more scalable in terms of memory usage and time.

Use the `alias` clause to provide a column label for the `comp` results, and is optional.

The `by` clause identifies the rows in the result set that will be aggregated. This clause is optional. Provide one or more fields to this clause. All fields with matching values are used to perform the aggregation. For example, if you had records such as:

```
number,id,product
100,"se1","A55"
50,"se1","A60"
50,"se1","A60"
25,"se2","A55"
25,"se2","A60"
```

The you can aggregate on the number column, and perform aggregation based on matching values in the id and/or product column. So if you sum the number column by the id column, you would get two results:

- 200 for "se1"
- 50 for "se2"

If you summed by id and product, you would get:

- 100 for "se1" and "A55" (there are no matching pairs).
- 100 for "se1" and "A60" (there is one matching pair).
- 25 for "se2" and "A55" (there are no matching pairs).
- 25 for "se2" and "A60" (there are no matching pairs).

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Wildcard Aggregates

You can use a wildcard to perform an aggregate for every field contained in the result set, except for the field(s) specified in the `by` clause.

Wildcards are only supported with aggregate functions and not approximate aggregate functions.

The syntax for this is:

```
comp <aggregate function>(*) as * [by [asc|desc] <field1>[,<field2>...]]
[addrawdata = true|false as <target field>]
```

For wildcards to work, all of the fields contained in the result set that are not identified in the `by` clause must be aggregatable.

Examples

Sum the action_total_download values for all records with matching pairs of values for the `actor_process_image_path` and `actor_process_command_line` fields. The query

calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final comp results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path,
actor_process_command_line as Process_CMD,
action_total_download as Download
| filter Download > 0
| limit 100
| comp sum(Download) as total by Process_Path, Process_CMD addrawdata = true as raw_data
```

Using the `panw_ngfw_traffic_raw` dataset, sum the `bytes_total`, `bytes_received`, and `bytes_sent` values for every record contained in the result set with a matching value for `source_ip`. The query calculates a maximum of 1000 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final comp results.

The `comp` stage runs on 1000 raw data events, but only a 100 will be displayed in the `raw_data` column.

```
dataset = panw_ngfw_traffic_raw
| fields bytes_total, bytes_received, bytes_sent, source_ip
| limit 1000
| comp sum(*) as * by source_ip addrawdata = true as raw_data
```

comp Aggregate Functions

The aggregate functions you can use with the `comp` stage are:

- avg
- count
- count_distinct
- earliest
- first
- last
- latest
- list
- max
- median
- min
- stddev_population
- stddev_sample
- sum
- values
- var

comp Approximate Aggregate Functions

The approximate aggregate functions you can use with the **`comp`** stage are:

- approx_count
- approx_quantiles
- approx_top

config

Abstract

Learn more about the Cortex Query Language `config` stage that configures the query behavior.

Syntax

```
config <function>
```

Description

The `config` stage configures the query behavior. It must be used with one of the config Functions. This stage must be presented as the first stage in the query.

config Functions

These functions you can use with the `config` stage:

- case_sensitive
- timeframe

case_sensitive

Abstract

Learn more about the Cortex Query Language `case_sensitive` config stage.

Syntax

```
config case_sensitive = true | false
```

Description

The `case_sensitive` configuration identifies whether field values are evaluated as case sensitive or case insensitive. The `config case_sensitive` stage must be added at the beginning of the query. You can also add another `config case_sensitive` stage when adding a join or union stage to a query.

If you do not provide this stage in your query, the default behavior is `false`, and case is not considered when evaluating field values.

Things to keep in mind about before implementing this stage

- The Settings → Configurations → XQL Configuration → Case Sensitivity (case_sensitive) setting can overwrite this `case_sensitive` configuration for all fields in the application, except for BIOCs, which will remain case insensitive no matter what this setting is set to.
- From Cortex XDR version 3.3, the default case sensitivity setting was changed to case insensitive (`config case_sensitive = false`). If you've been using Cortex XDR before this version was released, the default case sensitivity setting is still configured to be case sensitive (`config case_sensitive = true`).

The `config case_sensitive` stage can't be used to compare a field to an inner query. In this situation, ensure to use the `lowercase` or `uppercase` functions on the field and inner query stages and functions syntax.
Example 113.
This query won't provide the correct results of comparing the `agent_hostname` field with the inner query:

```
config case_sensitive = false
| dataset = xdr_data
| fields agent_hostname
| filter agent_hostname in (dataset = <lookup dataset> | fields agent_hostname)
```

This query will provide the correct output:

```
config case_sensitive = false
| dataset = xdr_data
| fields agent_hostname
```

- | filter lowercase(agent_hostname) in (dataset = <lookup dataset> | alter lower_agent_hostname = lowercase(agent_hostname) | fields lower_agent_hostname)

- The `config case_sensitive` stage can't be used to compare a field to an array that contains non-literal strings, for example a field name or function.
  Example 114.
  The results of this example are true, where the left side (`uppercase("a")`) is lowercase as it's not an array, and the right side (`("x", "A")`) is also an array that contains only literal strings.
  | alter field_name = if(uppercase("a") in ("x", "A"), true, false)

  Example 115.
  The results of this example are false, where the left side (`uppercase("a")`) is lowercase as it's not an array, and the right side (`"x", uppercase("a"))`) is an array that contains a function (`uppercase("a")`).
  | alter field_name = if(uppercase("a") in ("x", uppercase("a")),true, false)

Examples

```
config case_sensitive = true
| dataset = xdr_data
| fields actor_process_image_name as apin
| filter apin != NULL and apin contains "python"
| limit 100
```

timeframe

Abstract

Cortex Query Language `timeframe` configuration enables performing searches within a specific time frame from the query execution.

Syntax

- **Exact Time**
  config timeframe between "<Year-Month-Day H:M:S ±Timezone>" and "<Year-Month-Day H:M:S ±Timezone>"
- **Relative Time**
  config timeframe = <number><time unit>
  config timeframe between "<+|-><number><time unit>" and "now"
  config timeframe between "begin" and "<+|-><number><time unit>"
  config timeframe between "<+|-><number><time unit>" and "<+|-><number><time unit>"

Description

The `timeframe` configuration enables you to perform searches within a specific time frame from the query execution. The results for the time frame are based on times listed in the `_Time` column in the results table.

You can add the `timeframe` configuration to your queries using different formats depending on whether the time frame you are setting is an exact time or relative time.

When you set an exact time, include the `config timeframe` details: `between "<Year-Month-Day H:M:S ±Timezone>" and "<Year-Month-Day H:M:S ±Timezone>"`. The `±Timezone` format is: `±xxxx`. When you do not configure a timezone, the default is `UTC`. The exact time is based on a static time frame according to when the query is sent.

When you set a relative time, you have a few options for setting the `config timeframe`, where the syntax `<+|->` indicates whether to go back (`-`) or forward (`+`) in time. The default is back (`-`).

- <number><time unit>
  Enables setting a static time frame according to when the query is sent, where you choose the `<time unit>` from the available time unit options listed in the table below.

- between "<+|-><number><time unit>" and "now"
  Enables setting a time frame between a defined start time, where you choose the `<time unit>` from the available time unit options listed in the table below, and the end time as the time the query is run with the preset keyword "now".
- between "begin" and "<+|-><number><time unit>"
  Enables setting a time frame between a preset start time according to the Unix epoch time 00:00:00 UTC on 1 January 1970 with the "begin" keyword, and a defined ending time, where you choose the `<time unit>` from the available time unit options listed in the table below.
- between "<+|-><number><time unit>" and "<+|-><number><time unit>"
  Enables setting a time frame between a defined starting and ending time, where you choose the `<time unit>` from the available time unit options listed in the table below.

When a query includes any inner queries, the inner queries receives its time frame from the outer query unless the inner query has a separate time frame defined.

Connection to the time period in the Query Builder

When using the Query Builder to define a query, the time period can be set at the top right of the query window using the time picker, and the default is 24 hours. Whenever the time period is changed in the query window, the `config timeframe` is automatically set to the time period defined, but this won't be visible as part of the query. Only if you manually type in the `config timeframe` will this be seen in the query.

Available time units

Time Unit	Description
S	seconds
M	minutes
H	hours
D	days

W	weeks
MO	months
Y	years

The time unit is not case sensitive.

Examples

Relative Time

Example of `<number><time unit>`

For the last 10 hours from when the query is sent, return a maximum of 100 `xdr_data` records.

```
config timeframe = 10h
| dataset = xdr_data
```

- | limit 100

Example of `between "<+|-><number><time unit>" and "now"`

Since the last two days until now when the query is run, return a maximum of 100 `xdr_data` records.

```
config timeframe between "2d" and "now"
| dataset = xdr_data
```

- | limit 100

Example of `between "begin" and "<+|-><number><time unit>"`

Since the Unix epoch time 00:00:00 UTC on 1 January 1970 until the past 2 years when the query is run, return a maximum of 100 `xdr_data` records.

```
config timeframe between "begin" and "2y"
| dataset = xdr_data
```

- | limit 100

Example of `between "<+|-><number><time unit>" and "<+|-><number><time unit>"`

Since the last four days until the next 5 days when the query is run, return a maximum of 100 `xdr_data` records.

```
config timeframe between "-4d" and "+5d"
| dataset = xdr_data
```

- | limit 100

Exact Time

From April 1, 2021 at 9:00 a.m. UTC -02:00 until April 2, 2021 at 10:00 a.m. UTC -02:00, return a maximum of 100 `xdr_data` records.

```
config timeframe between "2021-04-01 09:00:00 -0200" and "2021-04-02 10:00:00 -0200"
| dataset = xdr_data
| limit 100
```

dedup

Abstract

Learn more about the Cortex Query Language `dedup` stage that removes duplicate occurrences of field values.

Syntax

```
dedup <field1>[,<field2>, ...] by asc | desc <field>
```

Description

The `dedup` stage removes all records that contain duplicate values (or duplicate sets of values) from the result set. The record that is returned is identified by the `by` clause, which selects the record by either the first or last occurance of the field specified in this clause.

The `dedup` stage can only be used with fields that contain numbers or strings.

Examples

Return unique values for the `actor_primary_username` field. For any given field value, return the first chronologically occurring record.

```
dataset = xdr_data
| fields actor_primary_username as apu
| filter apu != null
| dedup apu by asc _time
```

Return the last chronologically occurring record for any given `actor_primary_username` value.

```
dataset = xdr_data
| fields actor_primary_username as apu
| filter apu != null
| dedup apu by desc _time
```

Return the first occurrence seen by for any given `actor_primary_username`. field value.

```
dataset = xdr_data
| fields actor_primary_username as apu
| filter apu != null
| dedup apu by asc apu
```

Return unique groups of `actor_primary_username` and `os_actor_primary_username` field values. For each unique grouping, return the pair that first appears on a record with a non-NULL `action_file_size` field.

```
dataset = xdr_data
| fields actor_primary_username as apu,
      os_actor_primary_username as oapu,
      action_file_size as afs
| filter apu != null and afs != null
| dedup apu, oapu by asc afs
```

fields

Abstract

Learn more about the Cortex Query Language `fields` stage that defines the fields returned in the result set.

Syntax

```
fields [-] <field_1> [as <name1>], <field_2> [as <name2>], ...
```

Description

The `fields` stage declares which fields are returned in the result set, including name changes. If this stage is used, then subsequent stages can operate only on the fields identified by this stage.

Use a wildcard (`*`) to include all fields that match the pattern. Use a minus character (`-`) to exclude a field from the result set. The following system fields cannot be excluded and are always displayed:

- _time
- _insert_time
- _raw_log
- _product
- _vendor
- _tag
- _snapshot_id

- _snapshot_log_count
- _snapshot_collection_ts
- _id

Use the `as` clause to set an alias for a field. If you use the `as` clause, then subsequent stages must use that alias to refer to the field.

Examples

Return the `action_country` field from all `xdr_data` records where the `action_country` field is both not null and not "-". Also include all fields with names that match `event_*` except for `event_type`.

```
dataset = xdr_data
| fields action_country as ac
| fields event_*
| fields - event_type
| filter ac != null and ac != "-"
```

filter

Abstract

Learn more about the Cortex Query Language `filter` stage that narrows down the displayed results.

Syntax

filter *<boolean expr>*

Description

The `filter` stage identifies which data records should be returned by the query. Filters are boolean expressions that can use a wide range of functions and operators to express the filter. If a record matches the filter as the filter expression returns `true` when applied to the record, the record is returned in the query's result set.

The functions you can use with a filter are described in Functions. For a list of supported operators, see Supported operators.

Single vs triple double quotes behavior

Cortex XDR enables you to use single double quotes (`"<text>"`) or triple double quotes (`"""<text>"""`) when defining your XQL syntax for string manipulation. This specific syntax is used with different stages, functions, and operators, with or without wildcards. Typically, the `alter` and `filter` stages are used with single or triple double quotes.

Using single double quotes

Single double quotes (`"<text>"`) include the following functionality:

- Treats the string value literally.
- Wildcards using the asterisk (*) are processed as XQL wildcards, and match any sequence of characters.
- Escape sequences, such as `\n` (new line) or `\t` (tab), are not processed and are treated as plain characters.

Example 116.

`"\test\"` means to look for `\test\`

Using triple double quotes

Triple double quotes (`"""<text>"""`) include the following functionality:

- Enables regex-style pattern matching and escape sequence interpretation.
- Escape sequences, such as `\n` (new line) or `\t` (tab), are processed.
- Wildcards using the asterisk (*) are processed as XQL wildcards, and match any sequence of characters.

Example 117.

`"""\\test\\"""` means to look for `\test\`

Understanding the results:

- The double backslashes (`\\`) at the beginning becomes a single backlash (`\`) as it's processed as an escaped backslash.
- `test` is interpreted as literal.
- The double backslashes (`\\`) at the end becomes a single backlash (`\`) as it's processed as an escaped backslash.

Query example using filter

When using the `filter` stage, you can use both single (`"<text>"`) and triple (`"""<text>"""`) double quotes when specifying string values. The difference lies in how special characters and pattern matching are interpreted.

The examples provided are based on the following data table for a dataset called `test_dataset`:

_TIME	TEST
Mar 26th 2022 19:26:07	12\t3
May 7th 2023 15:16:00	12 3
Jun 8th 2024 16:56:27	1233
Mar 26th 2024 19:26:07	123
Apr 5th 2024 11:21:02	12\t34563
Apr 9th 2025 13:22:22	1233345
May 9th 2025 13:22:22	12 35897
May 30th 2025 21:45:02	116

Example 118.

```
config timeframe = 10y
| dataset = test_dataset
| filter test = "12\t3*"
| fields test
```

Output results table:

_TIME	TEST
Mar 26th 2022 19:26:07	12\t3

Apr 5th 2024 11:21:02	12\t34563

Explanation of results:

The asterisk (`*`) in `"12\t3*"` means to process the string field as an XQL wildcard by matching any sequence of characters that begins with `12\t3`. In addition, the `\t` characters are not processed as an escape character, but as plain characters.

Example 119.

```
config timeframe = 10y
| dataset = test_dataset
| filter test = """12\t3*"""
| fields test
```

Output results table:

_TIME	TEST
May 7th 2023 15:16:00	12 3
May 9th 2025 13:22:22	12 35897

Explanation of results:

The `\t` in `"""12\t3*"""` is processed as a tab escape character. The asterisk (`*`) in `"""12\t3*"""` means to process the string field as an XQL wildcard by matching any sequence of characters that begins with `12<tab>3`.

Examples

Return `xdr_data` records where the `event_type` is `NETWORK` and the `event_sub_type` is `NETWORK_HTTP_HEADER`.

```
dataset = xdr_data
| filter event_type = NETWORK and event_sub_type = NETWORK_HTTP_HEADER
```

When entering filters to the XQL Search user interface, possible field values for fields of type `enum` are available using the auto-complete feature. However, the autocomplete can only show enum values that are known to the schema. In some cases, on data import an enum value is included that is not known to the defined schema. In this case, the value will appear in the result set as an unknown value, such as `event_type_unknown_4`. Be aware that even though this value appears in the result set, you cannot create a filter using it. For example, this query will fail, even if you know the value appears in your result set:

```
dataset = xdr_data
| filter event_type = event_type_unknown_4
```

When using fields of type `enum`, the following syntax is supported.

Syntax format A

```
| filter event_type = ENUM.FILE
```

Syntax format B

```
| filter event_type = FILE
```

getrole

Abstract

Learn more about the Cortex Query Language `getrole` stage that enriches events with specific roles associated with usernames or endpoints.

This stage requires an Identity Threat Module license to view the results.

This stage is unsupported in BIOCs and real-time Correlation Rules.

Syntax

```
getrole <field> [as <alias>]
```

Description

The `getrole` stage enriches events with specific roles associated with usernames or endpoints. The `getrole` stage receives as an input a string field that is either a username in the `NETBIOS\SAM` format, such as `mydomain\myuser`, or the agent ID of a host. The agent ID can be found in the `endpoints` dataset as `endpoint_id` or in the `xdr_data` dataset as `agent_id`.

The roles for this field are displayed in a column called asset_roles in the results table. If there is one or more roles associated with the field, the values are represented as a string array, such as `['ADMIN', 'USER']`, and are listed in the asset_roles column. If there are no roles, the resulting column is an empty array.

You can also change the name of the column using `as` in the syntax to define an alias: `getrole <field> as <alias>`.

In addition, it is possible to use the `filter` stage with a new `ROLE` prefix to display the results of a particular role using the syntax:

- To include one specific role:
  - `filter <field> = ROLE.<role name>`
  - `filter array_length(arrayfilter(<field>, "@element" = ROLE.<role name> )) > 0`
- To include more than one specific role:
  - `filter <field> in (ROLE.<role name1>, ROLE.<role name2>, ....)`
- To exclude one specific role:
  - `filter array_length(arrayfilter(<field>, "@element" = ROLE.<role name> )) = 0`
- To exclude more than one specific role:
  - `filter array_length(arrayfilter(<field>, "@element" in (ROLE.<role name1>, ROLE.<role name2>, ....))) = 0`

Examples

Return a maximum of 100 `xdr_data` records with the enriched events including specific roles associated with usernames. If there are one or more roles associated with the value of the `user_id` string field column, the output is displayed in the asset_roles column in the results table. Otherwise, the field is empty.

```
dataset = xdr_data
| limit 100
| getrole user_id
```

Return a maximum of 100 `xdr_data` records of all the powershell executions made by the `SERVICE_ACCOUNTS` user role in the organization. The first `filter` stage indicates how to filter for the parent process, which is powershell.exe. The `fields` stage indicates the field columns to include in the results table and which ones are renamed in the table: `action_process_image_name` to `process_name` and `action_process_image_command_line` to `process_cmd`. The `getrole` stage indicates

the enriched events to include for the specific roles associated with usernames. If the `ROLE.SERVICE_ACCOUNTS` role is associated with any values in the `actor_effective_username` string field column, the row is displayed in the results table. Otherwise, the entire row is excluded from the results table.

```
dataset = xdr_data
| filter event_type = ENUM.PROCESS  and event_sub_type = ENUM.PROCESS_START and
lowercase(actor_process_image_name) = "powershell.exe"
| fields action_process_image_name as process_name, action_process_image_command_line as process_cmd, event_id,
actor_effective_username
| getrole actor_effective_username as user_roles
| filter user_roles = ROLE.SERVICE_ACCOUNTS
| limit 100
```

iploc

Abstract

Learn more about the Cortex Query Language `iploc` stage that associates IPv4 addresses of fields to a list of predefined attributes related to the geolocation.

Syntax

```
iploc <field>
```

Description

The `iploc` stage associates the IPv4 address of any field to a list of predefined attributes related to the geolocation. By default, when using this stage in your queries, the geolocation data is added to the results table in these predefined column names: LOC_ASN_ORG, LOC_ASN, LOC_CITY, LOC_CONTINENT, LOC_COUNTRY, LOC_LATLON, LOC_REGION, and LOC_TIMEZONE.

The `loc_latlon` field contains a string that is a combination of two floating numbers representing the latitude and longitude separated by a comma, for example, “32.0695,34.7621".

The following options are available to you when using this stage in your queries:

- You can specify the geolocation fields that you want added to the results table.
- You can append a suffix to the name of the geolocation field column in the results table.
- You can change the name of the geolocation field column in the results table.
- You can also view the geolocation data on a graph type called map, where the `xaxis` is set to either `loc_country` or `loc_latlon`, and the `yaxis` is a number field.
- The `iploc` stage can only be used with fields that contain numbers or strings.
- To improve your query performance, we recommend that you `filter` the data in your query before the `iploc` stage is run. In addition, limiting the number of fields in the results table further improves the performance.

Examples

Return a maximum of 1000 `xdr_data` records with the specific geolocation data associated with the `action_remote_ip` field, where no record with a null value for `action_remote_ip` is included, and displays the name of the city in a column called `city` and a combination of the latitude and longitude in a column called `loc_latlon` with comma-separated values of latitude and longitude.

```
dataset = xdr_data
| limit 1000
| filter action_remote_ip != null
| iploc action_remote_ip loc_city as city, loc_latlon
```

Return a maximum of 1000 `xdr_data` records with all the available geolocation data with the predefined column names, and add the specified `suffix _remote_id` to each predefined column name, where no record with a null value for `action_remote_ip` is included.

```
dataset = xdr_data
| limit 1000
| filter action_remote_ip != null
| iploc action_remote_ip suffix=_remote_id
```

Return a maximum of 1000 `xdr_data` records with the specific geolocation data associated with the `action_remote_ip` field that includes the name of the country (contained in `loc_country`) in a column called `country`, where no record with a null value for either `country` or `action_remote_ip` is included. The comp stage is used to count the number of IP addresses per country. The results are displayed in a graph type `of kind` map, where the x-axis represents the `country` and the y-axis the `action_remote_ip`.

```
dataset = xdr_data
| limit 1000
| iploc action_remote_ip loc_country as country
| filter country != null and action_remote_ip  != null
| comp count() as ip_count by country
| view graph type = map xaxis = country yaxis = ip_count
```

join

Abstract

Learn more about the Cortex Query Language `join` stage that combines the results of two queries into a single result set.

Syntax

```
join conflict_strategy = both|left|right
    type = inner|left|right
```

```
((<xql query>)
as <execution_name>
<boolean_expr>)
```

Description

The `join()` stage combines the results of two queries into a single result set. This stage is conceptually identical to a SQL join.

Parameter/Clause	Description
`conflict_strategy`	Identifies the join conflict strategy when there is a conflict in the column names between the 2 result sets which one should be chosen, either: • `right`: The column from the inner `join` query is used (default), which implements a right outer join. • `left`: The column from the orignal result set in the dataset is used, which implements a left outer join. • `both`: Both columns are used. The original result set column from the dataset keeps the current name, while the inner `join` query result set column name includes the following suffix added to the current name `_joined_10`, such as `<original column name>_joined_10`, and depending on the number of conflicted fields the suffix increases to `_joined_11`, `_joined_12`....

`type`	Identifies the join type. • `inner` Returns all the records in common between the queries that are being joined. This is the default join type. • `right` Returns all records from the join result set, plus any records from the parent result set that intersect with the join result set. • `left` Returns all records from the parent result set, plus any records from the join result set that intersect with the parent result set.
<xql query>	Provides the XQL query to be joined with the parent query.
as <execution_name>	Provides an alias for the join query's result set. For example, if you specify an execution name of `join1`, and in the join query you return field `agent_id`, then you can subsequently refer to that field as `join1.agent_id`.
<boolean_expr>	Identifies the conditions that must be met in order to place a record in the join result set.

This stage does not preserve sort order. If you are combing this stage with a sort stage, specify the `sort` stage after the `join`.

Examples

Return `microsoft_windows_raw` records, which are combined with the `xdr_data` records to include a new column called `edr`. For the `event_type` set to `EVENT_LOG`, the `actor_process_image_name` and `event_id` fields are returned from all `xdr_data` records, which are then compared to the fields inside the `microsoft_windows_raw` dataset, where `edr.event_id = edr_event_id`, and the results are added to the new `edr` column.

```
dataset  = microsoft_windows_raw
| join (dataset = xdr_data | filter event_type = EVENT_LOG | fields actor_process_image_name, event_id )
as edr edr.event_id = edr_event_id
```

Return a maximum of 100 `xdr_data` records with the events of the `agent_id`, `event_id` , and `_product` fields, where the `_product` field is displayed as `product`. The `agent_id`, `event_id`, and `_product` fields are returned from all `xdr_data` records and are then compared to the fields inside the `panw_ngfw_filedata_raw` dataset, where `_time = panw.time`, and the results are added to the new `panw` column. When there is a conflict in the column names between the 2 result sets both columns are used.

```
dataset = xdr_data
| fields agent_id, event_id, _product as product
| join conflict_strategy = both (dataset = panw_ngfw_filedata_raw | fields _product as product)
as panw _time = panw._time
| limit 100
```

limit

Abstract

Learn more about the Cortex Query Language `limit` stage that sets the maximum number of records that can be returned in the result set.

Syntax

```
limit <number>
```

Description

The `limit` stage sets the maximum number of records that can be returned in the result set. If this stage is not specified in the query, 1,000,000 is used.

Using a small limit can greatly increase the performance of your query by reducing the number of records that Cortex XDR can return in the result set.

Examples

Set the maximum number of records returned by the query to 10.

```
dataset = xdr_data | limit 10
```

replacenull

Abstract

Learn more about the Cortex Query Language `replacenull` stage that replaces null field values with a text string.

Syntax

```
replacenull <field> = <text string>
```

Description

The `replacenull` stage replaces null field values with the specified text string. This guarantees that every field in your result set will contain a value.

If you use the `replacenull` stage, then all subsequent stages that refer to the field's null value must use the replacement text string.

Examples

Return the `action_country` field from every `xdr_data` records where the `action_country` field is null, using the text string `N/A` in the place of an empty field value.

```
dataset = xdr_data
| fields action_country as ac
| replacenull ac = "N/A"
| filter ac = "N/A"
```

sort

Abstract

Learn more about the Cortex Query Language `sort` stage that identifies the sort order for records returned in the result set.

Syntax

```
sort asc|desc <field1>[, asc|desc <field2>...]
```

Description

The `sort` stage identifies the sort order for records returned in the result set. Records can be returned in ascending (`asc`) or descending (`desc`) order. If you include more than one field in the `sort` stage, records are sorted in field specification order.

Keep the following points in mind before running a query with the sort stage:

- To acheive the correct sorting results when a query includes strings representing numbers, it's recommend to sort by integer fields and to convert all string fields to integers; for example, by using the `to_integer` function.

- When sorting by multiple columns, the sort is saved correctly, but the user interface will only display the results according to the first sorted column.

Examples

Return the `action_boot_time` and `event_timestamp` fields from all `xdr_data` records. Sort the result set first by the `action_boot_time` field value in descending order, then by `event_timestamp` field in ascending order.

```
dataset = xdr_data
| fields action_boot_time as abt, event_timestamp as et
| sort desc abt, asc et
| limit 1
```

Tag

Abstract

Learn more about the Cortex Query Language `tag` stage that adds a single tag or list of tags to the _tag system field.

Syntax

- Add a single tag:
  | tag add <tag name>
- Add a list of tags:
  | tag add "<tag name1>", "<tag name2>", "<tag name3>",.....

Description

The `tag` stage is used in combination with the `add` operator to append a single tag or list of tags to the `_tag` system field, which you can easily query in the dataset.

Examples

In the `xdr_data` dataset, add a single tag called `"test"` to the `_tag` system field.

```
dataset = xdr_data
| tag add "test"
```

In the `xdr_data` dataset, add a list of tags, `"test1"`, `"test2"`, and `"test3"`, to the `_tag` system field.

```
dataset = xdr_data
| tag add "test1", "test2", "test3"
```

target

Abstract

Learn more about the Cortex Query Language `target` stage that saves query results to a dataset or lookup dataset.

Syntax

target type=dataset|lookup [append=true|false] <*dataset name*>

Description

The `target()` stage saves query results to a named dataset or lookup. These are persistent and can be used in subsequent queries. This stage must be the last stage specified in the query.

The `type` argument defines the type of dataset to create, when a new one needs to be created. The following types are supported:

- `dataset`: A regular dataset of type `USER`. Use `dataset` if you are saving the query results for use in future queries.
- `lookup`: A small lookup table with a 50 MB limit. When uploading a lookup dataset from the Dataset Management page, the limit is 30 MB. This lookup table can be used with parsing rules and downloaded as a JSON file. Use `lookup` if you want to export the query results to a disk.

Dataset and lookup tables support low frequency changes of up to 1200 modifications per day. Changes are implemented whenever a dataset or lookup dataset are edited.

Optional Append

Use `append` to define whether the data from the current query should be appended to the dataset (`true`) or re-created as a new dataset (`false`). If no `append` is included, the default is `false`. This means that after the query runs the data in an existing dataset is replaced with the new data.

Example 1

Save the results of a simple query to a named dataset.

```
dataset = xdr_data
| fields action_boot_time as abt
| filter abt != null
| target type=dataset abt_dataset
```

Subsequently, you can query the new dataset. Notice that the field names used by the new dataset conform to the aliases that you used when you created the dataset:

```
dataset = abt_dataset
| filter abt = 1603986614040
```

Example 2

The following example creates a dataset with the number of agents per country.

```
dataset = xdr_data
| fields agent_id,  action_country
| comp count_distinct(agent_id) as count by action_country
| target type=dataset append=false agents_per_country
```

This results in the following XQL JSON:

```
{
   "tables": [
      "xdr_data"
   ],
   "original_query": "\n
            dataset=xdr_data\n
            | fields agent_id,  action_country \n
            | comp count_distinct(agent_id) as count by action_country\n
            | target type=dataset append=false agents_per_country\n
            ",    "stages":
[
      {
         "FIELD_SELECT": {
            "fields": [
               {                    "name": "agent_id",                    "as": None
               },
               {                    "name": "action_country",                    "as": None
               }
            ],
            "exclude": []
         }
      },
      {
         "GROUP": {
            "aggregations": [
               {
                  "function": "count_distinct",
                  "parameters": [
                     "$agent_id"
                  ],
                  "name": "count"
               }
            ],
            "key": [
               "action_country"
            ]
         }
      }
   ],
   "output": [
```

```
    {
      "TARGET": {
        "type": "dataset",
        "target": "agents_per_country",
        "append": False
      }
    }
  ]
}
```

top

Abstract

Learn more about the Cortex Query Language `top` stage that returns the approximate count of top elements for a field and percentage of the count results.

This stage is unsupported with Correlation Rules.

Syntax

top <*integer*> <*field*> [by <*field1*> ,<*field2*>...] [top_count as <*column name*>, top_percent as <*column name*>]

Description

The `top` stage returns the approximate count of top elements for a given field and the percentage of the count results relative to the total number of values for the designated field. Use this top stage to produce approximate results, which are more scalable in terms of memory usage and time.

The <*integer*> in the syntax represents the number of top elements to return. If a number is not specified, up to 10 elements are returned by default. The approximate count is listed in the results table in a column called TOP_COUNT and the percentage in a column called TOP_PERCENT. You can update the column names for both tables by defining `top_count as` <*column name*> , top_percent as <*column name*> in the syntax. If you only define one column name to update in the syntax, the results table displays that column without displaying the other column.

Examples

Returns a table with 3 columns called EVENT_ID, TOP_COUNT, and TOP_PERCENT with up to 10 unique values for `event_id` with the corresponding counts and percentages.

```
dataset = xdr_data
| top event_id
```

Returns a table with 3 columns called ACTION_COUNTRY, EVENT_ID, and TOTAL with a single unique value for the `event_id` for each `action_country` with the corresponding count in the TOTAL column.

```
dataset = xdr_data
| top 1 event_id by action_country top_count as total
```

transaction

Abstract

Learn more about the Cortex Query Language transaction stage used to find transactions based on events that meet certain constraints.

Syntax

transaction <*field_1, field_2, ...*> [span = <*time*> [timeshift = <*epoch time*> [timezone = "<*time zone*>"]] | startswith = <*condition*> endswith = <*condition*> allowunclosed= true|false] maxevents = <*number of events per transaction*>

Description

The `transaction` stage is used to find transactions based on events that meet certain constraints. This stage aggregates all fields in a JSON string array by fields defined as transaction fields. For example, using the `transaction` stage to find transactions based on the `user` and `user_ip` fields will make the aggregation of json strings of all fields by the `user` and `user_ip` fields. A maximum of 50 fields can be aggregated in a `transation` stage.

You can also configure whether the transactions falls within a certain time frame, which is optional to define. You can set one of the following:

- `span=<time>`: Use this command to set a time frame per transaction, where `<time>` is a combination of a number and time suffix. Set one time suffix from the list of available options listed in the table below. In addition, you can define a particular start time for grouping the events in your query according to the Unix epoch time by setting `timeshift = <epoch time> timezone = "<time zone>"`, which are both optional. You can configure the `<time zone>` offset using an hours offset, such as `“+08:00”`, or using a time zone name from the List of Supported Time Zones, such as `"America/Chicago"`. The query still runs without defining the epoch time or time zone. If no `timeshift = <epoch time> timezone = "<time zone>"` is set, the query runs according to last time set in the log.
- `startswith` and `endswith`: Use these commands to set a condition for the beginning or end of the transaction, where the condition can be a logical expression or free text search.

Set the `allowunclosed` flag to `true` to include transactions which don't contain an ending event. The last event will be 12 hours after the starting event. By default, this is set to `true` and transactions without an ending event are included.

Use the `maxevents` command to define the maximum number of events to include per transaction. If this command is not set, the default value is 100.

When using the transaction stage, 5 additional fields are added to the results displayed:

- `_start_time`: Indicates the initial timestamp of the transaction.
- `_end_time`: Indicates the last timestamp for the transaction.
- `_duration`: Displays the difference in seconds between the timestamps for the first and last events in the transaction.
- `_num_of_rows`: Indicates the number of events in the transaction.
- `_transaction_id`: Displays the unique transaction ID.

Time Suffix	Description
MS	milliseconds
S	seconds
M	minutes
H	hours
D	days
W	weeks
MO	months
Y	years

Example using Span

Return a maximum of 10 events per transaction from the `xdr_data` records based on the `user` and `agent_id` fields, where the transaction time frame is 1 hour.

```
dataset=xdr_data
|transaction user, agent_id span=1h timeshift = 1615353499
timezone = “+08:00” maxevents=10
```

This query results in the following XQL JSON:

```
{'TRANSACTION': {'fields': ['user', 'agent_id'], 'maxevents': 10, 'span': {'amount': 1, 'units': 'h', 'timeshift': None}}}
```

Example using Startswith and Endswith

Return a maximum of 99 events per transaction from the `xdr_data` records based on the `f1` and `f2` fields. The starting event of each transaction is an event, where one of the fields contains a string `"str_1"`, and the ending event of each transaction is an event, where one of the fields contains a string `"str_2"`.

```
dataset=xdr_data
| transaction f1, f2 startswith="str_1" endswith="str2" maxevents=99
```

This query results in the following XQL JSON:

```
{'TRANSACTION': {'fields': ['f1', 'f2'], 'search': {'startswith': {'filter': {'free_text': 'str_1'}}, 'endswith': {'filter': {'free_text': 'str2'}}},
'maxevents': 99}}
```

union

Abstract

Learn more about the Cortex Query Language `union` stage that combines two result sets into a single result set.

Syntax

```
union <datasetname>
union (<inner xql query>)
```

Description

The `union()` stage combines two result sets into one result. It can be used in two different ways.

If a dataset name is provided with no other arguments, the two datasets are combined for the duration of the query, and the fields in both datasets are available to subsequent stages.

If a Cortex Query Language (XQL) query is provided to this stage, the result set from that XQL union query is combined with the result set from the rest of the query. This is effectively an inner join statement.

Examples

First, create a dataset using the target stage. This results in a persistent stage that we can use later with a `union` stage.

```
dataset = xdr_data
| filter event_type = FILE and event_sub_type = FILE_WRITE
| fields agent_id, action_file_sha256 as file_hash, agent_hostname
| target type=dataset file_event
```

Then run a second query, using `union` so that the query can access the contents of the `file_event` dataset. Notice that this second query uses the `file_hash` alias that was defined for the `file_event` dataset.

```
dataset = xdr_data
| filter event_type = PROCESS and event_sub_type = PROCESS_START
| union file_event
| fields agent_id, agent_hostname, file_hash,
    actor_process_image_path as executed_by,
    actor_process_signature_vendor as executor_signer
| filter file_hash != null and executed_by != null
```

view

Abstract

Learn more about the Cortex Query Language `view` stage that configures the display of the result set.

Syntax

```
view highlight fields = <field1>[,<field2>,...] values = <value1>[,<value2>,...]
```

```
view graph type = area | bubble |column | funnel | gauge | line | map | pie | scatter | single | wordcloud
   xaxis = <field1>
   yaxis = <field2> [<optional parameters>]
```

Optional `series` parameter:

```
| view graph type = area | bubble | column | line | map | scatter
xaxis = <field1>
yaxis = <field2> [<optional parameters>]
```

- [series = <field3> [<optional parameters>]]

```
view column order = default | populated
```

Description

The `view()` stage configures the display of the result set in the following ways:

- `highlight`: Highlights specified strings that Cortex XDR finds on specified fields. The highlight values that you provide are performed as a substring search, so only partial value can be highlighted in the final results table.
- `graph type`: Creates an `area`, `bubble`, `column`, `funnel`, `gauge`, `line`, `map`, `pie`, `scatter`, `single`, or `wordcloud` chart based on the values found for the fields specified in the `xaxis` and `yaxis` parameters. In this mode, `view` also offers a large number of parameters that allow you to control colors, decorations, and other behavior used for the final chart, where the options can differ depending on the type of graph selected. You can also define a graph `subtype`, when setting the `graph type` to either `column` or `pie`.
    - (Optional) `series`: When creating an `area`, `bubble`, `column`, `line`, `map`, or `scatter` chart, you can define a `series` parameter by specifying a field (column) to group chart results based on y-axis values. The series parameter is only supported when defining a single y-axis value.
- You can also generate graphs and outputs of your query data directly in the Query Builder after running a Cortex Query Language (XQL) query in the Query Results tab without having to add the syntax in the query. For more information, see Graph query results.
  If you use `graph type`, the fields specified for `xaxis` and `yaxis` must be collatable or the query will fail.
- `column order`: Enables you to list the query results by popularity, where the most non-null returned fields are displayed first using the syntax `view column order = populated`. By default, if `column order` is not defined (or `view column order=default`), the original column order is used.
  This option does not apply to Cortex Query Language (XQL) queries in widgets, Correlation Rules, public APIs, reports, and dashboards. If you include the `view column order` syntax in these types of queries, Cortex XDR disregards the stage from the query and completes the rest of the query.

Examples

Use the dedup stage collect unique combinations of `event_type` and `event_sub_type` values. Highlight the word "STREAM" when it appears in the result set.

```
dataset = xdr_data
| fields event_type, event_sub_type
| dedup event_type, event_sub_type by asc _time
| view highlight fields = event_sub_type values = "STREAM"
```

Count the number of unique files accessed by each user, and show a column graph of the results, where the number of unique files are grouped by username. This query uses comp count_distinct to calculate the number of unique files per username.

```
dataset = xdr_data
| fields actor_effective_username as username, action_file_path as file_path
| filter file_path != null and username != null
| comp count_distinct(file_path) as file_count by username
| view graph type = column xaxis = username yaxis = file_count series = username
```

Count the number of unique files accessed by each user, and display the results by popularity according to the most non-null values returned fields. This query uses comp count_distinct to calculate the number of unique files per username.

```
dataset = xdr_data
| fields actor_effective_username as username, action_file_path as file_path
| filter file_path != null and username != null
| comp count_distinct(file_path) as file_count by username
| view column order = populated
```

windowcomp

Abstract

Learn more about the Cortex Query Language `windowcomp` stage that precedes functions calculating statistics.

Syntax

```
windowcomp <analytic function> (<field>)[by <fieldA> [,<fieldB>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]
```

Defining a field with an analytic function is optional when using a count function. For rank and row_number functions, it's not allowed.

Description

The `windowcomp` stage precedes functions calculating statistics. The results compute values over a group of rows and return a single result for each row, for all records that contain matching values for the fields identified using a combination of the by clause, sort, and range. Only one function can be defined per field, while the other parameters are optional. Yet, it's possible to define multiple fields.

Example 120.

```
| windowcomp sum(field_1) by field_2 sort field_3 as field_4, min(field_5) by field_6 sort field_7 as field_8
```

Supported functions

This stage includes the following functions:

Function Type	Function
Numbering functions	• rank • row_number
Navigation functions	• first_value • lag • last_value
Statistical aggregate functions	• stddev_sample • stddev_population
Aggregate functions	• avg • count • max • median • min • sum

Optional parameters

The optional parameters available to define in the `windowcomp` function are explained in the following table:

Optional parameters	Syntax	Description

By clause	`[by <fieldA> [,<fieldB>,...]`	The `by` clause is used to break up the input field rows into separate partitions, over which the `windowcomp` function is independently evaluated. • Multiple partition fields are allowed when using a partition `by` clause. • When this optional clause is omitted, all rows in the input table comprise a single partition.
Sort	`[sort [asc|desc] <field1> [,[asc|desc] <field2>,...]]`	Defines how field rows are ordered within a partition as either ascending (`asc`) or descending (`desc`). This clause is optional in most situations, but is required in some cases for navigation functions and `rank` function.

Between window frame clause	`[between 0|null|<number>|-<number> [and 0|null|<number>|-<number>]`	Sets the window frame around the current row within a partition, over which the window function is evaluated. Numbering functions and the `lag` function can't be used in the window frame clause. Creates a window frame with a lower and upper boundary. The first boundary represents the lower boundary. The second boundary represents the upper boundary. Every boundary can include the following options:
		
		• `null`: Starts at the beginning or at the end of the partition, depending on the placement of the `null`.
		• `0`: Is set to the current row, where the window frame starts or ends at the current row.
		• positive/negative `<number>`: The end of the window frame or the start of the window frame relative to the current row.
		  ○ If only a start `<number>` is defined, only a negative number is allowed: `-<number>`.
		  ○ If a start `<number>` and end `<number>` are defined, the end `<number>` must be greater than the start `<number>`.
		
		If the `sort` is included, but the window frame clause isn't, the following window frame clause is used by default:
		
		between null and 0

frame_type	`[frame_type=rows | range]`	Defines the option of the frame as either: • `rows` (default): Computes the window frame based on physical offsets from the current row. For example, you could include two rows before and after the current row. To apply the default `frame_type=rows`, nothing needs to be added to the `windowcomp` stage syntax as it's automatically built into the query. • `range`: Computes the window frame based on a logical range of rows around the current row, based on the current row's sort key value. The provided range value is added or subtracted to the current row's key value to define a starting or ending range boundary for the window frame. Setting the `range` with start or end numeric, nonzero boundaries requires using exactly one numeric type of sort field. When setting `frame_type=range`, the `sort` must be included in the `windowcomp` stage syntax; otherwise, only `between null and null` is supported. Example 121. This is unsupported: | windowcomp sum(field_a) between -2 and 0 frame_type = range Yet, the following is supported: | windowcomp sum(field_a) sort desc field_b between -1 and 1 frame_type = range Or | windowcomp sum(field_a) between null and

		null frame_type = range
Alias clause	`[as <alias>]`	Use the `alias` clause to provide a column label (field name) for the `windowcomp` results. When the new field name already exists in the schema, it's replaced with the new name. Example 122. If the `xdr_data` dataset already has a field in the schema called `existing_field`, the new `existing_field` replaces the old one. dataset = xdr_data | windowcomp sum(field_a) as existing_field

Examples

Data table for ips dataset

The examples provided are based on the following data table for a dataset called `ips`:

ip	category	logins
192.168.10.1	pc	23
192.168.10.2	server	2
192.168.20.1	pc	9

192.168.20.4	server	8
192.168.20.5	pc	2
192.168.30.1	pc	10

Query 1: Compute the total logins for all IPs

```
dataset = ips
| windowcomp sum(logins) as total_logins
```

Output results table

ip	logins	category	total_logins
192.168.10.2	2	server	54
192.168.20.5	2	pc	54
192.168.20.4	8	server	54
192.168.20.1	9	pc	54
192.168.30.1	10	pc	54
192.168.10.1	23	pc	54

Query 2: Compute a subtotal for each category

```
dataset = ips
| windowcomp sum(logins) by category sort asc logins between null and null as total_logins
```

Output results table

ip	logins	category	total_logins

192.168.10.2	2	server	10
192.168.20.4	8	server	10
192.168.20.5	2	pc	44
192.168.20.1	9	pc	44
192.168.30.1	10	pc	44
192.168.10.1	23	pc	44

Query 3: Compute a cumulative sum for each category

The sum is computed with respect to the order defined using the `sort` clause. These two queries produce the same results:

```
dataset = ips
| windowcomp sum(logins) by category sort asc logins between null and 0 as total_logins
```

OR

```
dataset = ips
| windowcomp sum(logins) by category sort asc logins between null as total_logins
```

Output results table

ip	logins	category	total_logins
192.168.10.2	2	server	2
192.168.20.4	8	server	10
192.168.20.5	2	pc	2

192.168.20.1	9	pc	11
192.168.30.1	10	pc	21
192.168.10.1	23	pc	44

Query 4: Compute a cumulative sum, where only preceding rows are analyzed.

The analysis starts two rows before the current row in the partition.

```
dataset = ips
| windowcomp sum(logins) sort asc logins between null and -2 as total_logins
```

Output results table

ip	logins	category	total_logins
192.168.10.2	2	server	NULL
192.168.20.5	2	pc	NULL
192.168.20.4	8	server	2
192.168.20.1	9	pc	4
192.168.30.1	10	pc	12
192.168.10.1	23	pc	21

Query 5: Compute a changing average

The lower boundary is 1 row before the current row. The upper boundary is 1 row after the current row.

```
dataset = ips
| windowcomp avg(logins) sort asc logins between -1 and 1 as avg_logins
```

Output results table

ip	logins	category	avg_logins
192.168.10.2	2	server	2
192.168.20.5	2	pc	4
192.168.20.4	8	server	6.33333
192.168.20.1	9	pc	9
192.168.30.1	10	pc	14
192.168.10.1	23	pc	16.5

Query 6: Retrieve the most popular IP in each category

Defines how rows in a window are partitioned and ordered in each partition.

```
dataset = ips
| windowcomp last_value(ip) by category sort asc logins between null and null as most_popular
```

Output results table

ip	logins	category	most_popular
192.168.10.2	2	server	192.168.20.4
192.168.20.4	8	server	192.168.20.4
192.168.20.5	2	pc	192.168.10.1

192.168.20.1	9	pc	192.168.10.1
192.168.30.1	10	pc	192.168.10.1
192.168.10.1	23	pc	192.168.10.1

Query 7: Calculate the rank of each IP within the category based on the login

```
dataset = ips
| windowcomp rank() by category sort asc logins as rank
```

Output results table

ip	logins	category	rank
192.168.10.2	2	server	1
192.168.20.4	8	server	2
192.168.20.5	2	pc	1
192.168.20.1	9	pc	2
192.168.30.1	10	pc	3
192.168.10.1	23	pc	4

Query 8: Retrieve the most popular IP in a specific window frame by range and not category

The window frame analyzes up to three rows at a time.

```
dataset = ips
| windowcomp last_value(ip) by category sort asc logins between -1 and 1 as most_popular
```

Output results table

ip	logins	category	most_popular
192.168.10.2	2	server	192.168.20.4
192.168.20.4	8	server	192.168.20.4
192.168.20.5	2	pc	192.168.20.1
192.168.20.1	9	pc	192.168.30.1
192.168.30.1	10	pc	192.168.10.1
192.168.10.1	23	pc	192.168.10.1

Query 9: Retrieve the number of IPs that have similar logins

Count in range of -1 and 1 from their login value.

```
dataset = ips | fields ip, category , logins
| windowcomp count() sort asc logins between -1 and 1 frame_type = range as similar_logins
```

Output results table

ip	logins	category	similar_logins
192.168.10.5	2	pc	2
192.168.10.2	2	server	2
192.168.20.4	8	server	2
192.168.20.1	9	pc	3

192.168.30.1	10	pc	2
192.168.10.1	23	pc	1

Functions

Abstract

Learn more the functions that can be used with Cortex Query Language (XQL) stages in Cortex XDR.

Some Cortex Query Language (XQL) stages can call XQL functions to convert the data to a desired format. For example, the `current_time()` function returns the current timestamp, while the `extract_time()` function can obtain the hour information in the timestamp. Functions may or may not need input parameters. The `filter` and `alter` stages are the two stages that can use functions for data transformations.

add

Abstract

Learn more about the Cortex Query Language `add()` function that adds two integers.

Syntax

```
add (<string> | <integer>, <string> | <integer>)
```

Description

The `add()` function adds two positive integers. Parameters can be either integer literals, or integers as a string type, such as might be contained in a data field.

Example

```
dataset = xdr_data
| alter mynum = add(action_file_size, 3)
| fields action_file_size, mynum
| filter action_file_size > 0
| limit 1
```

approx_count

Abstract

Learn more about the Cortex Query Language `approx_count` approximate aggregate comp function.

Syntax

comp approx_count(<*field*>) [as <*alias*>] [by <*field1*>[,<*field2*>...]] [addrawdata = true|false [as <*target field*>]]

Description

The `approx_count` approximate aggregate is a comp function that counts the number of distinct values in the given field over a group of rows. For the group of rows, the function returns an approximate result as a single interger value, for all records that contain matching values for the fields identified in the `by` clause. Use this approximate aggregate function to produce approximate results, instead of exact results used with regular aggregate functions, which are more scalable in terms of memory usage and time. This approximate aggregate function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting addrawdata to either `true` or `false` (default), which are used to configure the final comp results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Example

Returns a single integer value after approximately counting the number of distinct values in the `event_id` field over a group of rows.

```
dataset = xdr_data
| fields event_id
| comp approx_count(event_id)
```

approx_quantiles

Abstract

Learn more about the Cortex Query Language `approx_quantiles` approximate aggregate comp function.

Syntax

comp approx_quantiles(<*field*>, <*number*>, <*true*|*false*>) [as <*alias*>] [by <*field1*>[,<*field2*>...]][addrawdata = true|false [as <*target field*>]]

Description

The `approx_quantiles` approximate aggregate is a comp function returns the approximate boundaries as a single value for a group of distinct or non-distinct values (default `false`) for the specified field over a group of rows, for all records that contain matching values for the fields

identified in the `by` clause. This function returns an array of `<number>` + 1 elements, where the first element is the approximate minimum and the last element is the approximate maximum. Use this approximate aggregate function to produce approximate results, instead of exact results used with regular aggregate functions, which are more scalable in terms of memory usage and time. This approximate aggregate function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Distinct Values Example

Returns the approximate boundaries for a group of distinct values in the `event_id` field.

```
dataset = xdr_data
| fields event_id
| comp approx_quantiles(event_id, 100, true)
```

Non-Distinct Values Example

Returns the approximate boundaries for a group of non-distinct values in the `event_id` field.

```
dataset = xdr_data
| fields event_id
| comp approx_quantiles(event_id, 100)
```

approx_top

Abstract

Learn more about the Cortex Query Language `approx_top` approximate aggregate comp function.

Syntax

comp approx_top as count

comp approx_top(<string field>, <number>) [as <alias>] [by <field1>[,<field2>...]][addrawdata = true|false [as <target field>]]

comp approx_top as sum

comp approx_top(<string field>, <number>, <weight string field>) [as <alias>] [by <field1>[,<field2>...]][addrawdata = true|false [as <target field>]]

Description

The `approx_top` approximate aggregate is a comp function that, depending on the number of parameters, returns either an approximate count or sum of top elements. This approximate

aggregate function returns a single value for the given field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause. This function is used in combination with a `comp` stage. When a third parameter is specified, it references a field that contains a numeric value (weight) that is used to calculate a sum. The return value is an array with up to `<number>` of JSON strings. Each string represents an object (struct) containing 2 keys and corresponding values. The keys depend on whether a third parameter has been supplied or not.

When defining `approx_top` to count and the third parameter is omitted, each struct will have these keys: "value" and "count", where the "value" specifies a unique field value and "count" specifies the number of occurrences. When the third parameter is specified in `approx_top`, it has to be a name of a field that contains a numeric value that is used to calculate the final sum for each unique value in the first specified field. Each struct in this case will have these keys: "value" and "sum".

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Use this approximate aggregate function to produce approximate results, instead of exact results used with regular aggregate functions, which are more scalable in terms of memory usage and time.

Examples

comp approx_top as count

Returns an approximate count of the top 10 agent IDs in the `agent_id` field that appear the most frequently. The return value is an array containing 10 JSON strings with a "value" and "count".

```
dataset = xdr_data
| fields agent_id
| comp approx_top(agent_id, 10)
```

comp approx_top as sum

Returns an approximate sum of the top 10 agent IDs in the `agent_id` field by their `action_session_duration`. The return value is an array containing 10 JSON strings with a "value" and "sum" for each `agent_id`.

```
dataset = xdr_data
| fields agent_id, action_session_duration
| comp approx_top(agent_id, 10, action_session_duration)
```

array_all

Abstract

Learn more about the Cortex Query Language `array_all()` function.

Syntax

```
array_all(<array>, "@element"<operator>"<array element>")
```

The `<operator>` can be any of the ones supported, such as `=` and `!=`.

Description

The `array_all()` function returns `true` when all the elements in a particular array match the condition in the specified array element. Otherwise, the function returns `false`.

Example

When the `dfe_labels` array is not empty, use the alter stage to create a new column called `x` that returns true when all the elements in the `dfe_labels` array is equal to `network`; otherwise, the function returns `false`.

```
dataset = xdr_data
| filter dfe_labels != null
| alter x = array_all(dfe_labels , "@element" = "network")
| fields x, dfe_labels
| limit 100
```

array_any

Abstract

Learn more about the Cortex Query Language `array_any()` function.

Syntax

```
array_any(<array>, "@element"<operator>"<array element>")
```

The `<operator>` can be any of the ones supported, such as `=` and `!=`.

Description

The `array_any()` function returns `true` when at least 1 element in a particular array matches the condition in the specified array element. Otherwise, the function returns `false`.

Example

When the `dfe_labels` array is not empty, use the alter stage to create a new column called `x` that returns true when at least 1 element in the `dfe_labels` array is equal to `network`; otherwise, the function returns `false`.

```
dataset = xdr_data
| filter dfe_labels != null
| alter x = array_any(dfe_labels , "@element" = "network")
| fields x, dfe_labels
| limit 100
```

arrayconcat

Abstract

Learn more about the Cortex Query Language `arrayconcat()` function that returns an array containing unique values found in the original array.

Syntax

```
arrayconcat (<array1>,<array2>[,<array3>...])
```

Description

The `arrayconcat()` function accepts two or more arrays, and it joins them into a single array.

Example

Given three arrays:

```
first_array : [1,2,3]
second_array : [44,55]
third_array : [4,5,6]
```

Using this query:

```
alter all_arrays = arrayconcat(first_array, second_array, third_array)
```

Results in an `all_arrays` field containing:

```
[1,2,3,44,55,4,5,6]
```

arraycreate

Abstract

Learn more about the Cortex Query Language `arraycreate()` function that returns an array based on the given parameters defined for the array elements.

Syntax

arraycreate ("<array element1>", "<array element2>",...)

Description

The `arraycreate()` function returns an array based on the given parameters defined for the array elements.

Example

Returns a final array to a field called x that is comprised of the elements [1,2].

```
dataset = xdr_data
| alter x = arraycreate("1", "2")
| fields x
```

arraydistinct

Abstract

Learn more about the Cortex Query Language `arraydistinct()` function that returns an array containing unique values found in the original array.

Syntax

arraydistinct (<array>)

Description

The `arraydistinct()` function accepts an array, and it returns a new array containing only unique elements found in the original array. That is, given the array:

[0,1,1,1,4,5,5]

This function returns:

[0,1,4,5]

arrayfilter

Abstract

Learn more about the Cortex Query Language `arrayfilter()` function.

Syntax

arrayfilter(<array>, <condition>)

arrayfilter(<array>, "@element"<operator>"<array element>")

The <operator> can be any of the ones supported, such as = and !=.

Description

The `arrayfilter()` function returns a new array with the elements which meet the given condition. The function does this by filtering the results of an array in one of the following ways:

- Returns the results when a certain condition is applied to the array.
- Returns the results when a particular array is set to a specified array element.

Though it's possible to define the `arrayfilter()` function with any condition, the examples below focus on conditions using the `@element` that are based on the current element being tested.

Basic Example

When the `dfe_labels` array is not empty, use the alter stage to assign a value to a field called `x` that returns the value of the `arrayfilter` function. The `arrayfilter` function filters the `dfe_labels` array for the array element set to `network`.

```
dataset = xdr_data
| filter dfe_labels != null
| alter x = arrayfilter(dfe_labels , "@element" = "network")
| fields x, dfe_labels
| limit 100
```

Advanced Example

This queries below illustrate how to check whether any IPs are included or not included in the blocked list called CIDRS. The Query Results tables are also included to help explain what happens as the `arrayfilter()` function is slightly modified.

Return Non-Matching CIDRS

This query returns results for each IP that don't match anything in the CIDRS array blocked list:

```
dataset = xdr_data
| limit 1
| alter cidrs = arraycreate("10.0.0.0/8","172.16.0.0/16"), ip = arraycreate("192.168.1.1", "172.16.20.18")
| fields cidrs, ip
| arrayexpand ip
| alter non_matching_cidrs = arrayfilter(cidrs, ip not incidr "@element")
```

Results:

The following table details for each IP the logic that is first performed before the final results for the query are displayed:

IP	Statement	TRUE/FALSE
192.168.1.1	not in 10.0.0.0/8	TRUE
192.168.1.1	not in 172.16.0.0/16	TRUE
172.16.20.18	not in 10.0.0.0/8	TRUE
172.16.20.18	not in 172.16.0.0/16	FALSE

For each IP, an array of CIDRS is returned in the NON_MATCHING_CIDRS column, which doesn't match the CIDRS array. In addition, from the above table, `arrayfilter()` only returns anything that resolves as TRUE. This explains the query results displayed in the following table:

IP	CIDRS	NON_MATCHING_CIDRS
192.168.1.1	10.0.0.0/8,172.16.0.0/16	10.0.0.0/8,172.16.0.0/16
172.16.20.18	10.0.0.0/8,172.16.0.0/16	10.0.0.0/8

Return Matching CIDRS

Now, let's update the query to return results for each IP that match anything in the CIDRS array:

```
dataset = xdr_data
| limit 1
| alter cidrs = arraycreate("10.0.0.0/8","172.16.0.0/16"), ip = arraycreate("192.168.1.1", "172.16.20.18")
| fields cidrs, ip
| arrayexpand ip
| alter matching_cidrs = arrayfilter(cidrs, ip incidr "@element")
```

Results:

The following table details for each IP the logic that is first performed before the final results for the query are displayed:

IP	Statement	TRUE/FALSE
192.168.1.1	in 10.0.0.0/8	FALSE
192.168.1.1	in 172.16.0.0/16	FALSE
172.16.20.18	in 10.0.0.0/8	FALSE
172.16.20.18	in 172.16.0.0/16	TRUE

For each IP, an array of CIDRS is returned in the MATCHING_CIDRS column, which matches the CIDRS array. In addition, from the above table, `arrayfilter()` only returns anything that resolves as TRUE. This explains the query results displayed in the following table:

IP	CIDRS	MATCHING_CIDRS
192.168.1.1	10.0.0.0/8,172.16.0.0/16	empty array
172.16.20.18	10.0.0.0/8,172.16.0.0/16	172.16.0.0/16

arrayindex

Abstract

Learn more about the Cortex Query Language `arrayindex()` function that returns the array element contained at the specified index.

Syntax

arrayindex(<array>, <index>)

Description

The `arrayindex()` function returns the value contained in the specified array position. Arrays are 0-based, and negative indexing is supported.

Examples

Use the split function to split IP addresses into an array of octets. Return the 3rd octet contained in the IP address.

```
dataset = xdr_data
| fields action_local_ip  as alii
| alter ip_third_octet = arrayindex(split(alii, "."), 2)
| filter alii != null and alii != "0.0.0.0"
| limit 10
```

arrayindexof

Abstract

Learn more about the Cortex Query Language `arrayindexof()` function that returns the index value of an array.

Syntax

```
arrayindexof(<array>, <condition>)
arrayindexof(<array>, "@element"<operator>"<array element>")
```

The `<operator>` can be any of the ones supported, such as `=` and `!=`.

Description

The `arrayindexof()` function enables you to return a value related to an array in one of the following ways.

- Returns 0 if a particular array is not empty and the specified condition is true. If the condition is not met, a NULL value is returned.
- Returns the 0-based index of a particular array element if a particular array is not empty and the specified condition using an `@element` is true. If the condition is not met, a NULL value is returned.

Examples

Condition

Use the alter stage to assign a value returned by the `arrayindexof` function to a field called `x`. The `arrayindexof` function reviews the `dfe_labels` array and returns 0 if the array is not empty and the `backtrace_identities` array contains more than 1 element. Otherwise, a NULL value is assigned to the `x` field.

```
dataset in (xdr_data)
| alter x = arrayindexof(dfe_labels , array_length(backtrace_identities) > 1)
| fields x, dfe_labels
```

```
| limit 100
```

@Element

When the `dfe_labels` array is not empty, use the alter stage to assign the 0-based index value returned by the `arrayindexof` function to a field called `x`. The `arrayindexof` function reviews the `dfe_labels` array and looks for the array element set to `network`. Otherwise, a NULL value is assigned to the `x` field.

```
dataset = xdr_data
| filter dfe_labels != null
| alter x = arrayindexof(dfe_labels , "@element" = "network")
| fields x, dfe_labels
| limit 100
```

array_length

Abstract

Learn more about the Cortex Query Language `array_length()` function that returns the length of an array.

Syntax

```
array_length (<array>)
```

Description

The `array_length()` function returns the number of elements in an array.

Example

```
dataset = xdr_data
| fields action_local_ip as alii
| alter ip_len = array_length(split(alii, "."))
| filter alii != null and alii != "0.0.0.0"
| limit 1
```

arraymap

Abstract

Learn more about the Cortex Query Language `arraymap()` function that applies a callable function to every element of an array.

Syntax

```
arraymap (<array>, <function()>)
```

Description

The `arraymap()` function applies a specified function to every element of an array. For functions that require a fieldname, use `"@element"`.

Examples

Extract the MAC address from the `agent_interface_map` field. This example uses the json_extract_scalar, to_json_string, json_extract_array, and arraystring functions to extract the desired information.

```
dataset = xdr_data
| alter mac =
    arraystring (
        arraymap (
            json_extract_array (to_json_string(agent_interface_map),"$."),
            json_extract_scalar ("@element", "$.mac")
        ), ",")
```

arraymerge

Abstract

Learn more about the Cortex Query Language `arraymerge()` function that returns an array created from a merge of the inner json-string arrays.

Syntax

```
arraymerge(<field>)
```

Description

The `arraymerge()` function returns an array, which is created from a merge of the inner json-string arrays, including merging a number of arraymap() function arrays. This function accepts a single array of json-strings, which is the `<field>` in the syntax.

Example 1

Returns a final array called `result` that is created from a merge of the inner json-string arrays from array `x` and array `y` with the values ["a", "b", "c", "d"].

```
dataset = xdr_data
| alter x= to_json_string(arraycreate("a","b")), y = to_json_string(arraycreate("c","d"))
| alter xy = arraycreate(x,y)
| alter xy=arraymerge(xy)
```

Example 2

Returns a final array that is created from a merge of the arraymap by extracting the IP address from the agent_interface_map field and the first IPV4 address found in the first element of the

`agent_interface_map` array. This example uses the to_json_string and json_extract_array functions to extract the desired information.

```
dataset = xdr_data
| alter a =
arraymerge (arraymap (agent_interface_map, to_json_string (json_extract_array (to_json_string("@element"), "$.ipv4") ) ) )
```

arrayrange

Abstract

Learn more about the Cortex Query Language `arrayrange()` function that returns a portion of an array based on specified array indices.

Syntax

```
arrayrange (<array>, <start>, <end>)
```

Description

The `arrayrange()` function returns a portion, or a slice, of an array given a start and end range. Indices are 0-based, and the start range is inclusive, but the end range is exclusive.

Example

So if you have an array:

```
[0,1,2,3,4,5,6]
```

and you specify:

```
arrayrange(<array>, 2, 4)
```

the function will return:

```
[2,3]
```

If you specify an end index that is higher than the last element in the array, the resulting array contains the starting element to the end of the array.

```
arrayrange(<array>, 2, 8)
```

the function will return:

```
[2,3,4,5,6]
```

arraystring

Abstract

Learn more about the Cortex Query Language `arraystring()` function that returns a string from an array, where each array element is joined by a defined delimiter.

Syntax

```
arraystring (<string>, <delimiter>)
```

Description

The `arraystring()` function returns a string from an array, where each array element is joined by a defined delimiter.

Examples

Retrieve all `action_app_id_transitions` that are not null, combine each array into a string where array elements are delimited by " : ", and then use dedup the resulting string.

```
dataset = xdr_data
| fields action_app_id_transitions  as aait
| alter transitions_string = arraystring(aait, " : ")
| dedup transitions_string by asc _time
| filter aait != null
```

avg

Abstract

Learn more about the Cortex Query Language `avg` used with both `comp` and `windowcomp` stages.

Syntax

comp stage

```
comp avg(<field>) [as <alias>] by <field_1>,<field_2> [addrawdata = true|false [as <target field>]]
```

windowcomp stage

```
windowcomp avg(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between
0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]
```

Description

The `avg()` function is used to return the average value of an integer field over a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `avg` aggregation function is used with a comp stage, the function returns a single average value of an integer field for a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `avg` aggregate function is used with a windowcomp stage, the function returns a single average value of an integer field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.

Examples

comp example

Return a single average value of the `action_total_download` field for a group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp avg(Download) as average_download by Process_Path, Process_CMD
addrawdata = true as raw_data
```

windowcomp example

Return the events that are above average per `Process_Path` and `Process_CMD`. The query returns a maximum of 100 `xdr_data` records in a column called `avg_download`.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| windowcomp avg(Download) by Process_Path, Process_CMD as avg_download
| filter Download > avg_download
```

coalesce

Abstract

Learn more about the Cortex Query Language `coalesce()` function that returns the first value that is not null from a defined list of fields.

Syntax

```
coalesce (<field_1>, <field_2>,...<field_n>)
```

Description

The `coalesce()` function takes an arbitrary number of arguments and returns the first value that is not NULL.

Example

Given a list of fields that contain usernames, select the first one that is not `null` and display it in the `username` column.

```
dataset = xdr_data
| fields actor_primary_username,
      os_actor_primary_username,
      causality_actor_primary_username
| alter username = coalesce(actor_primary_username,
                        os_actor_primary_username,
                        causality_actor_primary_username)
```

concat

Abstract

Learn more about the Cortex Query Language `concat()` function joins multiple strings into a single string.

Syntax

```
concat (<string1>, <string2>, ...)
```

Description

The `concat()` function joins multiple strings into a single string. When using the `concat()` function with multiple fields and any of the fields have a null/empty value, the function returns empty.

Example

Display the first non-NULL `action_boot_time` field value. In a second column called `abt_string`, use the `concat()` function to prepend "str: " to the value, and then display it.

```
dataset = xdr_data
| fields action_boot_time as abt
| filter abt != null
| alter abt_string = concat("str: ", to_string(abt))
| limit 1
```

convert_from_base_64

Abstract

Learn more about the Cortex Query Language `convert_from_base_64` function.

Syntax

```
convert_from_base_64("<base64-encoded input>")
```

Description

The `convert_from_base_64()` function converts the base64-encoded input to the decoded string format.

Example

Returns the decoded string format `Hello world` from the base64-encoded input `"SGVsbG8gd29ybGQ="`.

```
convert_from_base_64("SGVsbG8gd29ybGQ=")
```

count

Abstract

Learn more about the Cortex Query Language `count` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

```
comp count([<field>]) [as <alias>] by <field_1>,<field_2> [addrawdata = true|false [as <target field>]]
```

windowcomp stage

```
windowcomp count([<field>]) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]
```

Description

The `count()` function is used to return a single count for the number of rows either for a field over a group of rows, where only the number of non-null values found are returned, or without a field to count the number of rows, including null values. The function syntax and application is based on the preceding stage:

comp stage

When the `count` aggregation function is used with a comp stage, the function returns one of the following:

- With a field: Returns a single count for the number of non-null rows, for all records that contain matching values for the fields identified in the `by` clause.
- Without a field: Counts the number of rows and includes null values.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Use count_distinct to retrieve the number of unique values in the result set.

windowcomp stage

When the `count` aggregate function is used with a windowcomp stage, the function returns one of the following:

- With a field: Returns a single count for the number of non-null rows for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.
- Without a field: Counts the number of rows and includes null values.

Examples

comp example

Return a single count of all values found for the `actor_process_image_path` field in the group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp count(Process_Path) as num_process_path by process_path, process_cmd addrawdata = true as raw_data
| sort desc process_path
```

windowcomp example

Return a single count for the number of values found in the `dns_query_name` field for each row in the group of rows, for all records that contain matching values in the `agent_ip_addresses` field. The query returns a maximum of 100 `xdr_data` records. The results are provided in the `count_dns_query_name` column.

```
dataset = xdr_data
| limit 100
| windowcomp count(dns_query_name) by agent_ip_addresses as count_dns_query_name
```

count_distinct

Abstract

Learn more about the Cortex Query Language `count_distinct` aggregate comp function that counts the number of unique values found for a field in the result set.

Syntax

```
comp count_distinct(<field>) [as <alias>] by <field_1>,<field_2> [addrawdata= true|false [as <target field>]]
```

Description

The `count_distinct` aggregation is a comp function that returns a single value for the number of unique values found for a field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause. This aggregate function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Use count to retrieve the total number of values in the result set.

Examples

Return a single count of the number of unique values found for the `actor_process_image_path` field over a group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line values`. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp count_distinct(Process_Path) as num_process_path by process_path, process_cmd addrawdata = true as raw_data
| sort desc process_path
```

current_time

Abstract

Learn more about the Cortex Query Language `current_time()` function that returns the current time as a timestamp.

Syntax

```
current_time()
```

Description

The `current_time()` function returns a timestamp value representing the current time in the format `MMM dd YYYY HH:mm:ss`, such as `Jul 12th 2023 20:51:34`.

Example

From the `xdr_data` dataset, returns the events of the last 24 hours whose actor process started running more than 30 days ago.

```
dataset = xdr_data
| filter timestamp_diff(current_time(),to_timestamp(actor_process_execution_time, "MILLIS"), "DAY") > 30
```

date_floor

Abstract

Learn more about the Cortex Query Language `date_floor()` function.

Syntax

```
date_floor (<timestamp field>, "<time unit>" [, "<time zone>")
```

Description

The `date_floor()` function converts a timestamp value for a particular field or function result that contains a number, and returns a timestamp rounded down to the nearest whole value of a specified `<time unit>`, including a year (y), month (mo), week (w), day (d), or hour (h). The `<time zone>` offset is optional to configure using an hours offset, such as “+08:00”, or using a time zone name from the List of Supported Time Zones, such as "America/Chicago". When you do not configure a time zone, the default is UTC.

Example

Returns a maximum of 100 `xdr_data` records with the events of the `_time` field that are less than equal to a timestamp value. The timestamp value undergoes a number of different function manipulations. The current time is first rounded to the nearest whole value for the week according to the America/Los_Angeles time zone. This timestamp value is then converted to the Unix epoch timestamp format in seconds and is added to the -2073600 Unix epoch time. This Unix epoch time value in seconds is then converted to the final timestamp value that is used to filter the `_time` fields and return the resulting records.

```
dataset = xdr_data
| filter _time < to_timestamp(add(to_epoch(date_floor(current_time(),"w", "America/Los_Angeles")),-2073600))
| limit 100
```

divide

Abstract

Learn more about the Cortex Query Language `divide()` function that divides two integers.

Syntax

```
divide (<string> | <integer>, <string> | <integer>)
```

Description

The `divide()` function divides two positive integers. Parameters can be either integer literals, or integers as a string type, such as might be contained in a data field.

Example

```
dataset = xdr_data
| alter mynum = divide(action_file_size, 3)
| fields action_file_size, mynum
| filter action_file_size > 3
| limit 1
```

earliest

Abstract

Learn more about the Cortex Query Language `earliest` aggregate comp function that returns the earliest field value found with the matching criteria.

Syntax

```
comp earliest(<field>) [as <alias>] by <field_1>,<field_2> [addrawdata = true|false [as <target field>]]
```

Description

The `earliest` aggregation is a comp function that returns the chronologically earliest value found for a field over a group of rows that has matching values for the fields identified in the `by` clause. This function is dependent on a time-related field, so for your query to be considered valid, ensure that the dataset running this query contains a time-related field. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp`

results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Return the chronologically earliest timestamp found for any given `action_total_download` value for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` fields. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields _time, actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp earliest(_time) as download_time by Process_Path, Process_CMD addrawdata = true as raw_data
```

extract_time

Abstract

Learn more about the Cortex Query Language `extract_time()` function that returns a specified portion of a timestamp.

Syntax

```
extract_time (<timestamp>, <part>)
```

Description

The `extract_time` values are based on the GMT time, even if you've adjusted the Timezone or Timestamp Format server settings as these configurations only affect how to display in Cortex XDR. For more information on the server settings, see Configure server settings.

The `extract_time()` function returns a specified part of a timestamp. The `part` parameter must be one of the following keywords:

- `DAY`
- `DAYOFWEEK`
- `DAYOFYEAR`
- `HOUR`
- `MICROSECOND`
- `MILLISECOND`
- `MINUTE`
- `MONTH`

- QUARTER
- SECOND
- YEAR

Example

```
dataset = xdr_data
| alter timepart = extract_time(current_time(), "HOUR")
| fields timepart
| limit 1
```

extract_url_host

Abstract

Learn more about the Cortex Query Language `extract_url_host()` function.

Syntax

```
extract_url_host ("<URL>")
```

Description

The `extract_url_host()` function returns the host of the URL. The function always returns a value in lowercase characters even if the URL provided contains uppercase characters.

Example

Output examples when using the function

Returns `paloaltonetworks.com` from the complete URL: `https://www.paloaltonetworks.com`.

```
extract_url_host ("https://www.paloaltonetworks.com")
```

Returns `a.b` for the URL: `//user:password@a.b:80/path?query`

```
extract_url_host ("//user:password@a.b:80/path?query")
```

Returns `www.example.co.uk` in lowercase for the complete URL: `www.Example.Co.UK`, which includes uppercase characters.

```
extract_url_host ("www.Example.Co.UK")
```

Returns `www.test.paloaltonetworks.com` for the following URL containing suffixes: `https://www.test.paloaltonetworks.com/suffix/another_suffix`

```
extract_url_host ("https://www.test.paloaltonetworks.com/suffix/another_suffix")
```

Complete XQL Query Example

Returns one `xdr_data` record in the results table where the host of the URL `https://www.test.paloaltonetworks.com` is listed in the `URL_HOST` column as `www.test.paloaltonetworks.com`.

```
dataset = xdr_data
| alter url_host = extract_url_host("https://www.test.paloaltonetworks.com")
| fields url_host
| limit 1
```

extract_url_pub_suffix

Abstract

Learn more about the Cortex Query Language `extract_url_pub_suffix()` function.

Syntax

```
extract_url_pub_suffix ("<URL>")
```

Description

The `extract_url_pub_suffix()` function returns the public suffix of the URL, such as com, org, or net. The function always returns a value in lowercase characters even if the URL provided contains uppercase characters.

Example

Output examples when using the function

Returns `com` for the following URL: `https://paloaltonetworks.com`

```
extract_url_pub_suffix ("https://paloaltonetworks.com")
```

Returns `com` for the following URL containing suffixes: `https://www.test.paloaltonetworks.com/suffix/another_suffix`

```
extract_url_pub_suffix ("https://www.test.paloaltonetworks.com/suffix/another_suffix")
```

Complete XQL Query Example

Returns one `xdr_data` record in the results table where the public suffix of the URL `https://www.paloaltonetworks.com` is listed in the `URL_PUB_SUFFIX` column as `com`.

```
dataset = xdr_data
| alter url_pub_suffix = extract_url_pub_suffix("https://paloaltonetworks.com")
| fields url_pub_suffix
| limit 1
```

extract_url_registered_domain

Abstract

Learn more about the Cortex Query Language `extract_url_registered_domain()` function.

Syntax

```
extract_url_registered_domain ("<URL>")
```

Description

The `extract_url_registered_domain()` function returns the registered domain or registerable domain, the public suffix plus one preceding label, of a URL. The function always returns a value in lowercase characters even if the URL provided contains uppercase characters.

Examples

Output examples when using the function

Returns `paloaltonetworks.com` from the complete URL: `https://www.paloaltonetworks.com`.

```
extract_url_registered_domain ("https://www.paloaltonetworks.com")
```

Returns NULL for the URL: `//user:password@a.b:80/path?query`

```
extract_url_registered_domain ("//user:password@a.b:80/path?query")
```

Returns `example.co.uk` in lowercase for the complete URL: `www.Example.Co.UK`, which includes uppercase characters.

```
extract_url_registered_domain ("www.Example.Co.UK")
```

Returns `paloaltonetworks.com` for the following URL containing suffixes: `https://www.test.paloaltonetworks.com/suffix/another_suffix`

```
extract_url_registered_domain ("https://www.test.paloaltonetworks.com/suffix/another_suffix")
```

Complete XQL query example

Returns one `xdr_data` record in the results table where the registered domain of the URL `https://www.test.paloaltonetworks.com` is listed in the `REGISTERED_DOMAIN` column as `paloaltonetworks.com`.

```
dataset = xdr_data
```

```
| alter registered_domain = extract_url_registered_domain("https://www.test.paloaltonetworks.com")
| fields registered_domain
| limit 1
```

first

Abstract

Learn more about the Cortex Query Language `first` aggregate comp function that returns the first field value found in the dataset with the matching criteria.

Syntax

```
comp first(<field>) [as <alias>] by <field_1>,<field_2> [addrawdata = true|false [as <target field>]]
```

Description

The `first` aggregation is a comp function that returns a single first value found for a field in the dataset over a group of rows that has matching values for the fields identified in the `by` clause. This function is dependent on a time-related field, so for your query to be considered valid, ensure that the dataset running this query contains a time-related field. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Return the first timestamp found in the dataset for any given `action_total_download` value for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` fields. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields _time,actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download
as Download
| filter Download > 0
| limit 100
| comp first(_time) as download_time by Process_Path, Process_CMD addrawdata = true as raw_data
```

first_value

Abstract

Learn more about the Cortex Query Language `first_value()` navigation function that is used with a `windowcomp` stage.

Syntax

```
windowcomp first_value(<field>) [by <field> [,<field>,...]] sort [asc|desc] <field1> [, [asc|desc] <field2>,...] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]
```

Description

The `first_value()` function is a navigation function that is used in combination with a windowcomp stage. This function is used to return a single value of a field for the first row of each row in the group of rows in the current window frame, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort` (mandatory), and `between` window frame clause.

Example

Return the first IP address a user authenticated from successfully.

```
preset = authentication_story
| filter auth_identity not in (null, """""""") and auth_outcome = """SUCCESS""" and action_country != UNKNOWN
| alter et = to_epoch(_time), t = _time
| bin t span = 1d
| limit 100
| windowcomp first_value(action_local_ip) by auth_identity, t sort asc et between null and null as first_action_local_ip
| fields auth_identity , *action_local_ip
```

floor

Abstract

Learn more about the Cortex Query Language `floor()` function that rounds a field that contains a number down to the nearest whole integer.

Syntax

```
floor (<number>)
```

Description

The `floor()` function converts a field that contains a number, and returns an integer rounded down to the nearest whole number.

Example

format_string

Abstract

Learn more about the Cortex Query Language `format_string()` function.

Syntax

```
format_string("<format string>", <field_1>, <field_2>,...<field_n> )
```

Description

The `format_string()` function returns a string from a format string that contains zero or more format specifiers, along with a variable length list of additional arguments that matches the format specifiers. A format specifier is initiated by the % symbol, and must map to one or more of the remaining arguments. Usually, this is a one-to-one mapping, except when the * specifier is used.

Examples

STRING

```
dataset = xdr_data
| alter stylished_action_category_appID = format_string("-%s-", action_category_of_app_id )
| fields stylished_action_category_appID
| limit 100
```

-

Simple integer

```
dataset = xdr_data
| filter action_remote_ip_int != null
| alter simple_int = format_string("%d", action_remote_ip_int)
| fields simple_int
| limit 100
```

-

Integer with left blank padding

```
dataset = xdr_data
| filter action_remote_ip_int != null
| alter int_with_left_blank = format_string("|%100d|", action_remote_ip_int)
| fields int_with_left_blank
| limit 100
```

-

Integer with left zero padding

```
dataset = xdr_data
| filter action_remote_ip_int != null
| alter int_with_left_zero_padding = format_string("+%0100d+", action_remote_ip_int)
| fields int_with_left_zero_padding
| limit 100
```

-

format_timestamp

Abstract

Learn more about the Cortex Query Language `format_timestamp()` function that returns a string after formatting a timestamp according to a specified string format.

Syntax

format_timestamp("<format string>", <timestamp field>)

format_timestamp("<format string>", <timestamp field>, "<time zone>")

Description

The `format_timestamp()` function returns a string after formatting a timestamp according to a specified string format. The `<time zone>` is optional to configure using an hours offset, such as “+08:00”, or using a time zone name from the List of Supported Time Zones, such as "America/Chicago". The `format_timestamp()` function should include an alter stage. For more information, see the examples below.

Examples

Without a time zone configured

Returns a maximum of 100 `xdr_data` records, which includes a string field called `new_time` in the format `YYYY/MM/dd HH:mm:ss`, such as 2021/11/12 12:10:30. This format is detailed in the `format_timestamp` function, which defines retrieving the new_time (`%Y/%m/%d %H:%M:%S`) from the `_time` field.

```
dataset = xdr_data
| alter new_time = format_timestamp("%Y/%m/%d %H:%M:%S", _time)
| fields new_time
| limit 100
```

-

With a time zone configured using an hours offset

Returns a maximum of 100 `xdr_data` records, which includes a string field called new_time in the format YYYY/MM/dd HH:mm:ss, such as 2021/11/12 01:53:35. This format is detailed in the `format_timestamp` function, which defines the retrieving the new_time (`%Y/%m/%d %H:%M:%S`) from the `_time` field and adding +03:00 hours as the time zone format.

```
dataset = xdr_data
| alter new_time = format_timestamp("%Y/%m/%d %H:%M:%S", _time, "+03:00")
| fields new_time
| limit 100
```

-

With a time zone name configured

Returns a maximum of 100 `xdr_data` records, which includes a string field called `new_time` in the format `YYYY/MM/dd HH:mm:ss`, such as `2021/11/12 01:53:35`. This format is detailed

in the `format_timestamp` function, which defines the retrieving the `new_time` (`%Y/%m/%d %H:%M:%S`) from the `_time` field, and includes an "America/Chicago" time zone.

```
dataset = xdr_data
| fields _time
| alter new_time = format_timestamp("%Y/%m/%d %H:%M:%S", _time, "America/Chicago")
| fields new_time
```

- | limit 100

if

Abstract

Learn more about the Cortex Query Language `if()` function that returns a result after evaluating a condition.

Syntax

Regular if statement

```
if (<boolean expression>, <true return expression>[, <false return expression>])
```

Nested if/else statement

- if(<boolean expression1>, <true return expression1>, <boolean expression2>, <true return expression2>[, <boolean expression3>, <true return expression3>,...][, <false return expression>])
- if(<boolean expression1>, if(<boolean expression2>, <true return expression2> [,<false return expression2>])...[,<false return expression1>])

  In the above syntax, `if(<boolean expression2>, <true return expression2> [,<false return expression2>])` represents the `<true return expression1>`.

Description

The `if()` function evaluates a single expression or group of expressions depending on the syntax used to define the function. The syntax can be set up in the following ways:

- Regular if statement: A single boolean expression is evaluated. If the expression evaluates as `true`, the function returns the results defined in the second function argument. If the expression evaluates as `false` and a false return expression is defined, the function returns the results of the third function argument; otherwise, if no false return expression is set, returns null.
- Nested if/else statment: At least two boolean expressions and two true return expressions are required when using this option. The first boolean expression is evaluated. If the first expression evaluates as `true`, the function returns the results defined in the second function argument. The second boolean expression is evaluated. If the second expression evaluates as `true`, the function returns the results defined in the fourth function argument. If there are any other boolean expressions defined, they are

evaluated following the same pattern when evaluated as `true`. If any of the expressions evaluates as `false` and a false return expression is defined, the function returns the results defined in the last function argument for the false return expression; otherwise, if no false return expression is set, returns null.

Examples

Regular if statement

If '.exe' is present on the `action_process_image_name` field value, replace that substring with an empty string. This example uses the replace and lowercase functions, as well as the contains operator to perform the conditional check. When the '.exe' is not present, the value is returned as is.

```
dataset = xdr_data
| fields action_process_image_name as apin
| filter apin != null
| alter remove_exe_process =
    if(lowercase(apin) contains ".exe",  // boolean expression
       replace(lowercase(apin),".exe",""), // return if true
       lowercase(apin))  // return if false
| limit 10
```

Nested if/else statement

Return a maximum of 1 `xdr_data` record from the past 7 days. The table results include a new column called `check_ip`, which evaluates and returns the following:

- If the `action_local_ip` contains an IP address that begins with 10, return `Local 10`.
- If the `action_local_ip` contains an IP address that begins with 172, return `Local 172 ?`.
- If the `action_local_ip` contains an IP address that begins with 192.168, return `Local 192`.
- If all the above expressions evaluate as `false`, return null.

```
config timeframe = 7d | dataset = xdr_data
| limit 1
| alter
    check_ip = if(action_local_ip ~= "^10",//boolean expression1
              "Local 10", // true return expression1
              action_local_ip ~= "^172", //boolean expression2
              "Local 172 ?", //true return expression2
              action_local_ip ~= "^192\.168", //boolean expression3
              "Local 192") //true return expression3
```

incidr

Abstract

Learn more about the Cortex Query Language `incidr()` function.

Syntax

incidr(<IPv4_address>, <CIDR1_range1> | <CIDR1_range1, CIDR2_range2, ...>)

Description

The `incidr()` function accepts an IPv4 address, and an IPv4 range or comma separated IPv4 ranges using CIDR notation, and returns `true` if the address is in range. Both the IPv4 address and CIDR ranges can be either an explicit string using quotes (`""`), such as `"192.168.0.1"`, or a string field.

The first parameter must contain an IPv4 address contained in an IPv4 field. For production purposes, this IPv4 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv4 address to a field, and then use that field with this function.

Multiple CIDRs are defined with comma separated syntax when building an XQL query with the Query Builder or in Correlation Rules. When defining multiple CIDRs, the logical `OR` is used between the CIDRS listed, so as long as one address is in range the entire statement returns `true`. Here are a few examples of how this logic works to determine whether the `incidr()` function returns `true` and displays results or `false`, where no results are displayed:

Function returns `true` and results are displayed:

```
dataset = test
| alter ip_address = "192.168.0.1"
```

- | filter incidr(ip_address, "192.168.0.0/24, 1.168.0.0/24") = true

Function returns `false` and no results are displayed:

```
dataset = test
| alter ip_address = "192.168.0.1"
```

- | filter incidr(ip_address, "2.168.0.0/24, 1.168.0.0/24") = true

Function returns `false` and no results are displayed:

```
dataset = test
| alter ip_address = "192.168.0.1"
```

- | filter incidr(ip_address, "192.168.0.0/24, 1.168.0.0/24") = false

Function returns `true` and results are displayed:

```
dataset = test
| alter ip_address = "192.168.0.1"
```

- | filter incidr(ip_address, "2.168.0.0/24, 1.168.0.0/24") = false

The same logic is used when using the `incidr` and `not incidr` operators. For more information, see Supported operators.

Examples

Return a maximum of 10 `xdr_data` records, if the IPV4 address (`192.168.10.14`) is in range by verifying against a single CIDR (`192.168.10.0/24`):

```
alter my_ip = "192.168.10.14"
| alter inrange = incidr(my_ip, "192.168.10.0/24")
| fields inrange
| limit 10
```

Return a maximum of 10 `xdr_data` records, if the IPV4 address (`192.168.0.1`) is in range by verifying against multiple CIDRs (`192.168.0.0/24` or `1.168.0.0/24`):

```
dataset = xdr_data
| alter ip_address = "192.168.0.1"
| filter incidr(ip_address, "192.168.0.0/24, 1.168.0.0/24") = true
| limit 10
```

incidr6

Abstract

Learn more about the Cortex Query Language `incidr6()` function.

Syntax

```
incidr6(<IPv6_address>, <CIDR1_range1> | <CIDR1_range1, CIDR2_range2, ...>)
```

Description

The `incidr6()` function accepts an IPv6 address, and an IPv6 range or comma separated IPv6 ranges using CIDR notation, and returns `true` if the address is in range. Both the IPv6 address and CIDR ranges can be either an explicit string using quotes (`""`), such as `"3031:3233:3435:3637:3839:4041:4243:4445"`, or a string field.

The first parameter must contain an IPv6 address contained in an IPv6 field. For production purposes, this IPv6 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv6 address to a field, and then use that field with this function.

Multiple CIDRs are defined with comma separated syntax when building an XQL query with the Query Builder or in Correlation Rules. When defining multiple CIDRs, the logical `OR` is used between the CIDRS listed, so as long as one address is in range the entire statement returns

`true`. Here are a few examples of how this logic works to determine whether the `incidr6()` function returns `true` and displays results or `false`, where no results are displayed:

Function returns `true` and results are displayed:

```
dataset = test
| alter ip_address = "3031:3233:3435:3637:3839:4041:4243:4445"
```

- | filter incidr(ip_address, "3031:3233:3435:3637:0000:0000:0000:0000/64, 6081:6233:6435:6637:0000:0000:0000:0000/64") = true

Function returns `false` and no results are displayed:

```
dataset = test
| alter ip_address = "3031:3233:3435:3637:3839:4041:4243:4445"
```

- | filter incidr(ip_address, "6081:6233:6435:6637:0000:0000:0000:0000/64, 7081:7234:7435:7737:0000:0000:0000:0000/64, fe80::/10") = true

Function returns `false` and no results are displayed:

```
dataset = test
| alter ip_address = "3031:3233:3435:3637:3839:4041:4243:4445"
```

- | filter incidr(ip_address, "3031:3233:3435:3637:0000:0000:0000:0000/64, 7081:7234:7435:7737:0000:0000:0000:0000/64, fe80::/10") = false

Function returns `true` and results are displayed:

```
dataset = test
| alter ip_address = "3031:3233:3435:3637:3839:4041:4243:4445"
```

- | filter incidr(ip_address, "6081:6233:6435:6637:0000:0000:0000:0000/64, 7081:7234:7435:7737:0000:0000:0000:0000/64, fe80::/10") = false

The same logic is used when using the `incidr6` and `not incidr6` operators. For more information, see Supported operators.

Example

Return a maximum of 10 `xdr_data` records, if the IPV6 address (`3031:3233:3435:3637:3839:4041:4243:4445`) is in range by verifying against a single CIDR (`3031:3233:3435:3637:0000:0000:0000:0000/64`):

```
alter my_ip = "3031:3233:3435:3637:3839:4041:4243:4445"
| alter inrange = incidr6(my_ip, "3031:3233:3435:3637:0000:0000:0000:0000/64")
| fields inrange
| limit 10
```

Return a maximum of 10 `xdr_data` records, if the IPV6 address (`3031:3233:3435:3637:3839:4041:4243:4445`) is in range by verifying against multiple CIDRs (`2001:0db8:85a3:0000:0000:8a2e:0000:0000/64` or `fe80::/10`):

```
dataset = xdr_data
| alter ip_address = "fe80::1"
| filter incidr6(ip_address, "2001:0db8:85a3:0000:0000:8a2e:0000:0000/64, fe80::/10") = true
| limit 10
```

incidrlist

Abstract

Learn more about the Cortex Query Language `incidrlist()` function.

Syntax

```
incidrlist(<IP_address list>, <CIDR_range>)
```

Description

The `incidrlist()` function accepts a string containing a comma-separated list of IP addresses, and an IP range using CIDR notation, and returns `true` if all the addresses are in range.

Examples

Return `true` if the list of IP addresses fall within the specified IP range. Note that the input type is a comma-separated list of IP addresses, and not an array of IP addresses.

```
alter inrange = incidrlist("192.168.10.16,192.168.10.3",
                           "192.168.10.0/24")
| fields inrange
| limit 1
```

If you want to evaluate a true array of IP addresses, convert the array to a comma-separated list using arraystring(). For example, using the `pan_ngfw_traffic_raw` dataset:

```
dataset = panw_ngfw_traffic_raw
| filter dest_ip != null
| comp values(dest_ip) as dips by source_ip,action
| alter dips = arraystring(dips, ", ")
| alter inrange = incidrlist(dips, "192.168.10.0/24")
| fields source_ip, action, dips, inrange
| limit 100
```

int_to_ip

Abstract

Learn more about the Cortex Query Language `int_to_ip()` function that safely converts a signed integer representation of an IPv4 address to a string equivalent.

Syntax

```
int_to_ip(<IPv4_integer>)
```

Description

The `int_to_ip()` function tries to safely convert a signed integer representation of an IPv4 address into its string equivalent.

Examples

Returns the IPv4 address "4.130.58.140" from the integer representation of the IPv4 address provided as 75643532.

```
int_to_ip(75643532)
```

Returns the IPv4 address "251.125.197.116" from the integer representation of the IPv4 address provided as -75643532.

```
int_to_ip(-75643532)
```

ip_to_int

Abstract

Learn more about the Cortex Query Language `ip_to_int()` function that safely converts a string representation of an IPv4 address to an integer equivalent.

Syntax

```
ip_to_int(<IPv4_address>)
```

This function was previously called `safe_ip_to_int()` and was renamed to `ip_to_int()`.

Description

The `ip_to_int()` function tries to safely convert a string representation of an IPv4 address into its integer equivalent.

Example

Returns the integer 808530483 from the string representation of the IPv4 address provided as "48.49.50.51".

```
ip_to_int("48.49.50.51")
```

is_ipv4

Abstract

Learn more about the Cortex Query Language `is_ipv4()` function.

Syntax

is_ipv4(<IPv4_address>)

Description

The `is_ipv4()` function accepts a string, and returns `true` if the string is a valid IPv4 address. The IPv4 address can be either an explicit string using quotes (`""`), such as `"192.168.0.1"`, or a string field.

The `<IPv4_address>` must contain an IPv4 address in an IPv4 field. For production purposes, this IPv4 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv4 address to a field, and then use that field with this function.

Example

Data table for ips_test_raw dataset

The example provided is based on the following data table for a dataset called `ips_test_raw`:

_TIME	IP	_VENDOR	_PRODUCT
Mar 26th 2025 19:26:07	1.1.1.1	ips	test
Mar 26th 2025 19:26:07	192.168.1.100	ips	test
Mar 26th 2025 19:26:07	FF0E::1	ips	test
Mar 26th 2025 19:26:07	127.0.0.1	ips	test
Mar 26th 2025 19:26:07	172.32.0.1	ips	test
Mar 26th 2025 19:26:07	2606:4700:4700::1111	ips	test

Query: Filter the IPv4 addresses

```
dataset = ips_test_raw
| alter IsIpv4 = is_ipv4(ip)
```

| filter IsIpv4

Output results table

Returns all the IPv4 addresses from the `ip` field in the `ips_test_raw` dataset. When the `is_ipv4` function returns `true`, the results are displayed with a new IsIpv4 column (field) indicating a true value. If the function returns `false`, no results are returned.

_TIME	IP	_VENDOR	_PRODUCT	ISIPV4
Mar 26th 2025 19:26:07	1.1.1.1	ips	test	true
Mar 26th 2025 19:26:07	192.168.1.100	ips	test	true
Mar 26th 2025 19:26:07	127.0.0.1	ips	test	true
Mar 26th 2025 19:26:07	172.32.0.1	ips	test	true

is_known_private_ipv4

Abstract

Learn more about the Cortex Query Language `is_known_private_ipv4()` function.

Syntax

is_known_private_ipv4(<IPv4_address>)

Description

The `is_known_private_ipv4()` function accepts an IPv4 address, and returns `true` if the IPv4 string address belongs to any of the following known set of private network IPs:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

The IPv4 address can be either an explicit string using quotes (`""`), such as `"192.168.0.1"`, or a string field.

The `<IPv4_address>` must contain an IPv4 address in an IPv4 field. For production purposes, this IPv4 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv4 address to a field, and then use that field with this function.

Example

Data table for ips_test_raw dataset

The example provided is based on the following data table for a dataset called `ips_test_raw`:

_TIME	IP	_VENDOR	_PRODUCT
Mar 26th 2025 19:26:07	1.1.1.1	ips	test
Mar 26th 2025 19:26:07	192.168.1.100	ips	test
Mar 26th 2025 19:26:07	FF0E::1	ips	test
Mar 26th 2025 19:26:07	127.0.0.1	ips	test
Mar 26th 2025 19:26:07	172.32.0.1	ips	test
Mar 26th 2025 19:26:07	2606:4700:4700::1111	ips	test

Query: Filter the IPv4 addresses belonging to a set of known private network IPs

```
dataset = ips_test_raw
| alter IsKnownPrivateIpv4 = is_known_private_ipv4(ip)
| filter IsKnownPrivateIpv4
```

Output results table

Returns all the IPv4 addresses that belong to a set of known private network IPs from the `ip` field in the `ips_test_raw` dataset. When the `is_known_private_ipv4` function returns `true`, the results are displayed with a new IsKnownPrivateIpv4 column (field) indicating a true value. If the function returns `false`, no results are returned.

_TIME	IP	_VENDOR	_PRODUCT	ISKNOWNPRIVATEIPV4
Mar 26th 2025 19:26:07	192.168.1.100	ips	test	true

is_ipv6

Abstract

Learn more about the Cortex Query Language `is_ipv6()` function.

Syntax

is_ipv6(<IPv6_address>)

Description

The `is_ipv6()` function accepts a string, and returns `true` if the string is a valid IPv6 address. The IPv6 address can be either an explicit string using quotes (`""`), such as `"3031:3233:3435:3637:3839:4041:4243:4445"`, or a string field.

The `<IPv6_address>` must contain an IPv6 address in an IPv6 field. For production purposes, this IPv6 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv6 address to a field, and then use that field with this function.

Example

Data table for ips_test_raw dataset

The example provided is based on the following data table for a dataset called `ips_test_raw`:

_TIME	IP	_VENDOR	_PRODUCT
Mar 26th 2025 19:26:07	1.1.1.1	ips	test
Mar 26th 2025 19:26:07	192.168.1.100	ips	test

Mar 26th 2025 19:26:07	FF0E::1	ips	test
Mar 26th 2025 19:26:07	127.0.0.1	ips	test
Mar 26th 2025 19:26:07	172.32.0.1	ips	test
Mar 26th 2025 19:26:07	2606:4700:4700::1111	ips	test

Query: Filter the IPv6 addresses

```
dataset = ips_test_raw
| alter IsIpv6 = is_ipv6(ip)
| filter IsIpv6
```

Output results table

Returns all the IPv6 addresses from the `ip` field in the `ips_test_raw` dataset. When the `is_ipv6` function returns `true`, the results are displayed with a new IsIpv6 column (field) indicating a true value. If the function returns `false`, no results are returned.

_TIME	IP	_VENDOR	_PRODUCT	ISIPV6
Mar 26th 2025 19:26:07	FF0E::1	ips	test	true
Mar 26th 2025 19:26:07	2606:4700:4700::1111	ips	test	true

is_known_private_ipv6

Abstract

Learn more about the Cortex Query Language `is_known_private_ipv6()` function.

Syntax

```
is_known_private_ipv6(<IPv6_address>)
```

Description

The `is_known_private_ipv6()` function accepts an IPv6 address, and returns `true` if the IPv6 string address belongs to any of the following known set of private network IPs:

- FC00::/7
- FD00::/7

The IPv6 address can be either an explicit string using quotes (`""`), such as `"3031:3233:3435:3637:3839:4041:4243:4445"`, or a string field.

The `<IPv6_address>` must contain an IPv6 address in an IPv6 field. For production purposes, this IPv6 address will normally be carried in a field that you retrieve from a dataset. For manual usage, assign the IPv6 address to a field, and then use that field with this function.

Example

Data table for ips_test_raw dataset

The example provided is based on the following data table for a dataset called `ips_test_raw`:

_TIME	IP	_VENDOR	_PRODUCT
Mar 26th 2025 19:26:07	FC00::1	ips	test
Mar 26th 2025 19:26:07	192.168.1.100	ips	test
Mar 26th 2025 19:26:07	FF0E::1	ips	test
Mar 26th 2025 19:26:07	127.0.0.1	ips	test
Mar 26th 2025 19:26:07	172.32.0.1	ips	test
Mar 26th 2025 19:26:07	2606:4700:4700::1111	ips	test

Query: Filter the IPv6 addresses belonging to a set of known private network IPs

```
dataset = ips_test_raw
| alter IsKnownPrivateIpv6 = is_known_private_ipv6(ip)
| filter IsKnownPrivateIpv6
```

Output results table

Returns all the IPv6 addresses that belong to a set of known private network IPs from the `ip` field in the `ips_test_raw` dataset. When the `is_known_private_ipv6` function returns `true`, the results are displayed with a new IsKnownPrivateIpv6 column (field) indicating a true value. If the function returns `false`, no results are returned.

_TIME	IP	_VENDOR	_PRODUCT	ISKNOWNPRIVATEIPV6
Mar 26th 2025 19:26:07	FC00::1	ips	test	true

json_extract

Abstract

Learn more about the Cortex Query Language `json_extract()` function that accepts a string representing a JSON object, and returns a field value from that object.

Before using this JSON function, it's important that you understand how Cortex XDR treats a JSON in the Cortex Query Language. For more information, see JSON functions.

Syntax

Regular Syntax

```
json_extract(<json_object_formatted_string>, <json_path>)
```

When a field in the <json_path> contains characters, such as a dot (.) or colon (:), use the syntax:

```
json_extract(<json_object_formatted_string>, "['<json_field>']")
```

Syntactic Sugar Format

To make it easier for you to write your XQL queries, you can also use the following syntactic sugar format.

```
<json_object_formatted_string> -> <json_path>{}
```

When a field in the `<json_path>` contains characters, such as a dot (.) or colon (:), use the syntax:

```
<json_object_formatted_string> -> ["<json_field>"]{}
```

Description

The `json_extract()` function extracts inner JSON objects by retrieving the value from the identified field. The returned datatype is always a string. If the input string does not represent a JSON object, this function fails to parse. To convert a string field to a JSON object, use the to_json_string function.

JSON field names are case sensitive, so the key to field pairing must be identical in an XQL query for results to be found. For example, if a field value is `"TIMESTAMP"` and your query is defined to look for "timestamp", no results will be found.

The field value is always returned as a string. To return the scalar values, which are not an object or an array, use json_extract_scalar.

Examples

Return the `storage_device_name` value from the `action_file_device_info` field.

```
dataset = xdr_data
| fields action_file_device_info as afdi
| alter sdn = json_extract(to_json_string(afdi), "$.storage_device_name")
| filter afdi != null
```

Using Syntactic Sugar Format

The same example above with a syntactic sugar format.

```
dataset = xdr_data
| fields action_file_device_info as afdi
| alter sdn = to_json_string(afdi)->storage_device_name{}
| filter afdi != null
```

json_extract_array

Abstract

Learn more about the Cortex Query Language `json_extract_array()` function that accepts a string representing a JSON array, and returns an XQL-native array.

Before using this JSON function, it's important that you understand how Cortex XDR treats a JSON in the Cortex Query Language. For more information, see JSON functions.

Syntax

Regular Syntax

```
json_extract_array(<json_array_string>, <json_path>)
```

When a field in the `<json_path>` contains characters, such as a dot (.) or colon (:), use the syntax:

```
json_extract_array(<json_array_string>, "['<json_field>']")
```

Syntactic Sugar Format

To make it easier for you to write your XQL queries, you can also use the following syntactic sugar format.

```
<json_array_string> -> <json_path>[]
```

When a field in the `<json_path>` contains characters, such as a dot (.) or colon (:), use the syntax:

```
<json_array_string> -> ["<json_field>"][]
```

Description

The `json_extract_array()` function accepts a string representing a JSON array, and returns an XQL-native array. To convert a string field to a JSON object, use the to_json_string function.

JSON field names are case sensitive, so the key to field pairing must be identical in an XQL query for results to be found. For example, if a field value is `"TIMESTAMP"` and your query is defined to look for "timestamp", no results will be found.

Examples

Regular Syntax

Extract the first IPV4 address found in the first element of the `agent_interface_map` array.

```
dataset = xdr_data
| fields agent_interface_map as aim
| alter ipv4 = json_extract_array(to_json_string(arrayindex(aim, 0)) , "$.ipv4")
| filter aim != null
| limit 10
```

Syntactic Sugar Format

The same example above with a syntactic sugar format.

```
dataset = xdr_data
| fields agent_interface_map as aim
| alter ipv4 = to_json_string(aim)->[0].ipv4[0]
| filter aim != null
| limit 10
```

json_extract_scalar

Abstract

Learn more about the Cortex Query Language `json_extract_scalar()` function.

Before using this JSON function, it's important that you understand how Cortex XDR treats a JSON in the Cortex Query Language. For more information, see JSON functions.

Syntax

Regular Syntax

```
json_extract_scalar(<json_object_formatted_string>, <field_path>)
```

When a field in the `<json_path>` contains characters, such as a dot (.) or colon (:), use the syntax:

```
json_extract_scalar(<json_object_formatted_string>, "['<json_field>']")
```

Syntactic Sugar Format

To make it easier for you to write your XQL queries, you can also use the following syntactic sugar format:

```
<json_object_formatted_string> -> <field_path>
```

When a field in the `<json_path>` contains characters, such as a dot (.) or colon (:), use the syntax:

```
<json_object_formatted_string> -> ["<json_field>"]
```

Description

The `json_extract_scalar()` function accepts a string representing a JSON object, and it retrieves the value from the identified field as a string. This function always returns a string. If the JSON field is an object or array, it will return a null value. To retrieve an XQL-native datatype, use an appropriate function, such as `to_float` or `to_integer`. If the input string does not represent a JSON object, this function fails to parse. To convert a string field to a JSON object, use the to_json_string function.

JSON field names are case sensitive, so the key to field pairing must be identical in an XQL query for results to be found. For example, if a field value is `"TIMESTAMP"` and your query is defined to look for "timestamp", no results will be found.

Examples

Return the `storage_device_drive_type` value from the `action_file_device_info` field, and return the record if it is 1.

There are two ways that you can build this query either with a filter using an XQL-native datatype or string.

Option A - Filter using an XQL-native datatype

```
dataset = xdr_data
| fields action_file_device_info as afdi
| alter sdn = to_integer(json_extract_scalar(to_json_string(afdi), "$.storage_device_drive_type"))
| filter sdn = 1
| limit 10
```

Option B - Filter using a string

```
dataset = xdr_data
| fields action_file_device_info as afdi
| alter sdn = json_extract_scalar(to_json_string(afdi), "$.storage_device_drive_type")
| filter sdn = "1"
| limit 10
```

Using Syntactic Sugar Format

The same example above with a syntactic sugar format.

```
dataset = xdr_data
| fields action_file_device_info as afdi
| alter sdn = to_integer(to_json_string(afdi)->storage_device_drive_type)
| filter sdn = 1
| limit 10
```

json_extract_scalar_array

Abstract

Learn more about the Cortex Query Language `json_extract_scalar_array()` function.

Before using this JSON function, it's important that you understand how Cortex XDR treats a JSON in the Cortex Query Language. This function doesn't have a syntatic sugar format. For more information, see JSON functions.

Syntax

```
json_extract_scalar_array(<json_array_string>, <json_path>)
```

A field in the `<json_path>` that contains characters, such as a dot (.) or colon (:) and should be escaped as it's an invalid JSON path, is currently unsupported.

Description

The `json_extract_scalar_array()` function accepts a string representing a JSON array, and returns an XQL-native array. This function is equivalent to the json_extract_array except that the final output isn't displayed in double quotes ("..."). To convert a string field to a JSON object, use the to_json_string function.

JSON field names are case sensitive, so the key to field pairing must be identical in an XQL query for results to be found. For example, if a field value is `"TIMESTAMP"` and your query is defined to look for "timestamp", no results will be found.

Example

Extract the first IPV4 address found in the first element of the `agent_interface_map` array. The values of the IPv4 addresses in the array will not contain any double quotes.

```
dataset = xdr_data
| fields agent_interface_map as aim
| alter ipv4 = json_extract_scalar_array(to_json_string(arrayindex(aim, 0)) , "$.ipv4")
| filter aim != null
| limit 10
```

Final output with 1 row from the results table. Notice that the IPV4 column doesn't contain any double quotes (`" "`) around the IP address `172.16.15.42`:

_TIME	AIM	_PRODUCT	_VENDOR	INSERT_TIMESTAMP	IPV4
Aug 9th 2023 10:04:39	`[{"ipv4":["172.16.15.42"], "ipv6":[], "mac":"00:50:56:9f:30:a9"}]`	XDR agent	PANW	Aug 17th 2023 19:25:48	172.16.15.42

In contrast, compare the above results to the same query using the `json_extract_array()` function. The final output with the same row from the results table has in the IPV4 column the IP address in double quotes `"172.16.15.42"`.

_TIME	AIM	_PRODUCT	_VENDOR	INSERT_TIMESTAMP	IPV4
Aug 9th 2023 10:04:39	`[{"ipv4":["172.16.15.42"], "ipv6":[], "mac":"00:50:56:9f:30:a9"}]`	XDR agent	PANW	Aug 17th 2023 19:25:48	"172.16.15.42"

lag

Abstract

Learn more about the Cortex Query Language `lag()` navigation function that is used with a `windowcomp` stage.

Syntax

windowcomp lag(<field>) [by <field> [,<field>,...]] sort [asc|desc] <field1> [, [asc|desc] <field2>,...] [as <alias>]

Description

The `lag()` function is a navigation function that is used in combination with a windowcomp stage. This function is used to return a single value of a field on a preceding row for each row in the group of rows using a combination of the `by` clause and `sort` (mandatory).

Example

Retrieve for each event the timestamp of the previous successful login since the last one.

```
preset = authentication_story
| filter auth_identity not in (null, """""""") and auth_outcome = """SUCCESS"""
| alter ep = to_epoch(_time)
| limit 100
| windowcomp lag(_time) by auth_identity sort asc ep as previous_login
```

last

Abstract

Learn more about the Cortex Query Language **`last`** aggregate comp function that returns the last field value found in the dataset with the matching criteria.

Syntax

comp last(*<field>*) [as *<alias>*] by *<field_1>*,*<field_2>* [addrawdata = true|false [as *<target field>*]]

Description

The `last` aggregation is a comp function that returns the last value found for a field in the dataset over a group of rows that has matching values for the fields identified in the `by` clause. This function is dependent on a time-related field, so for your query to be considered valid, ensure that the dataset running this query contains a time-related field. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Return the last timestamp found in the dataset for any given `action_total_download` value for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` fields. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields _time, actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp last(_time) as download_time by Process_Path, Process_CMD addrawdata = true as raw_data
```

last_value

Abstract

Learn more about the Cortex Query Language `last_value()` navigation function that is used with a `windowcomp` stage.

Syntax

windowcomp last_value(<field>) [by <field> [,<field>,...]] sort [asc|desc] <field1> [, [asc|desc] <field2>,...] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `last_value()` function is a navigation function that is used in combination with a windowcomp stage. This function is used to return a single value of a field for the last row of

each row in the group of rows in the current window frame, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort` (mandatory), and `between` window frame clause.

Example

Return the last IP address a user authenticated from successfully.

```
preset = authentication_story
| filter auth_identity not in (null, """""""") and auth_outcome = """SUCCESS""" and action_country != UNKNOWN
| alter et = to_epoch(_time), t = _time
| bin t span = 1d
| limit 100
| windowcomp last_value(action_local_ip) by auth_identity, t sort asc et between null and null as first_action_local_ip
| fields auth_identity , *action_local_ip
```

latest

Abstract

Learn more about the Cortex Query Language `latest` aggregate comp function that returns the latest field value found with the matching criteria.

Syntax

comp latest(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

Description

The `latest` aggregation is a comp function that returns a single chronologically latest value found for a field over a group of rows that has matching values for the fields identified in the `by` clause. This function is dependent on a time-related field, so for your query to be considered valid, ensure that the dataset running this query contains a time-related field. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Return the chronologically latest timestamp found for any given `action_total_download` value for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` fields. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields _time, actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp latest(_time) as download_time by Process_Path, Process_CMD addrawdata = true as raw_data
```

len

Abstract

Learn more about the Cortex Query Language `len` function that returns the number of characters contained in a string.

Syntax

```
len (<string>)
```

Description

The `len()` function returns the number of characters contained in a string.

Examples

Show domain names that are more than 100 characters in length.

```
dataset = xdr_data
| fields dns_query_name
| filter len(dns_query_name) > 100
| limit 10
```

list

Abstract

Learn more about the Cortex Query Language `list` aggregate comp function that returns an array for up to 100 values for a field in the result set.

Syntax

```
comp list(<field>) [as <alias>] by <field_1>,<field_2> [addrawdata = true|false [as <target field>]]
```

Description

The `list` aggregation is a comp function that returns a single array of up to 100 values found for a given field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause. The array values are all non-null, so null values are filtered out. The values returned in the array are non-unique, so if a value repeats multiple times it is included as part of the list of up to 100 values. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Examples

Return an array containing up to 100 values seen for the `action_total_download` field over a group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download
Download
| filter Download > 0
| limit 100
| comp list(Download) as list_download by Process_Path, Process_CMD addrawdata = true as raw_data
```

lowercase

Abstract

Learn more about the Cortex Query Language `lowercase()` function that converts a string field to all lowercase letters.

Syntax

```
lowercase (<string>)
```

Description

The `lowercase()` function converts a string field value to all lowercase.

Examples

Convert all `actor_process_image_name` field values that are not null to lowercase, and return a list of unique values.

```
dataset = xdr_data
| fields actor_process_image_name as apin
| dedup apin by asc _time
| filter apin != null
| alter apin = lowercase(apin)
```

ltrim, rtrim, trim

Abstract

Learn more about the Cortex Query Language `ltrim()`, `rtrim()`, and `trim()` functions that removes spaces or characters from the beginning or end of a string.

Syntax

trim (<string>,[trim_characters])

rtrim (<string>,[trim_characters])

ltrim (<string>,[trim_characters])

Description

The `trim()` function removes specified characters from the beginning and end of a string. The `rtrim()` removes specific characters from the end of a string. The `ltrim()` function removes specific characters from the beginning of a string.

If you do not specify trim characters, then whitespace (spaces and tabs) are removed.

Examples

Remove '.exe' from the end of the `action_process_image_name` field value.

```
dataset = xdr_data
| fields action_process_image_name as apin
| filter apin != null
| alter remove_exe_process = rtrim(apin, ".exe")
| limit 10
```

See also the replace function example.

max

Abstract

Learn more about the Cortex Query Language `max` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp max(*<field>*) [as *<alias>*] by *<field_1>*,*<field_2>* [addrawdata = true|false [as *<target field>*]]

windowcomp stage

windowcomp max(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `max()` function is used to return the maximum value of an integer field over a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `max` aggregation function is used with a comp stage, the function returns a single maximum value of an integer field for a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `max` aggregate function is used with a windowcomp stage, the function returns a single maximum value of an integer field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.

Examples

comp example

Return a single maximum value of the `action_total_download` field for a group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp max(Download) as max_download by Process_Path, Process_CMD addrawdata = true as raw_data
```

windowcomp example

Return the last login time. The query returns a maximum of 100 `authentication_story` records in a column called `action_user_agent`.

```
preset = authentication_story
| limit 100
| windowcomp max(_time) by action_user_agent
```

median

Abstract

Learn more about the Cortex Query Language `median` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp median(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

windowcomp stage

windowcomp median(<field>) [by <field> [,<field>,...]] [as <alias>]

Description

The `median()` function is used to return the median value of a field over a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `median` aggregation function is used with a comp stage, the function returns a single median value of a field for a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `median` aggregate function is used with a windowcomp stage, the function returns a single median value of a field for each row in the group of rows, for all records that contain matching values for the fields identified in the `by` clause. In a median function, the `sort` and `between` window frame clause are not used. The results are provided in a new column in the results table.

Examples

comp example

Return a single median value of the `action_total_download` field over a group of rows, for all records that have matching values for their `actor_process_image_path` and

`actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp median(Download) as median_download by Process_Path, Process_CMD
addrawdata = true as raw_data
```

windowcomp example

Return all events where the `Download` field is greater than the median by reviewing each individual event and how it compares to the median. The query returns a maximum of 100 `xdr_data` records in a column called `median_download`.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| windowcomp median(Download) by Process_Path, Process_CMD as median_download
| filter Download > median_download
```

min

Abstract

Learn more about the Cortex Query Language `min` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp min(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

windowcomp stage

windowcomp min(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `min()` function is used to return the minimum value of an integer field over a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `min` aggregation function is used with a comp stage, the function returns a single minimum value of an integer field for a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `min` aggregate function is used with a windowcomp stage, the function returns a single minimum value of an integer field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the by clause, sort, and between window frame clause. The results are provided in a new column in the results table.

Examples

comp example

Return a single minimum value of the `action_total_download` field for a group of rows, for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp min(Download) as min_download by Process_Path, Process_CMD addrawdata = true as raw_data
```

windowcomp example

Return the first login time. The query returns a maximum of 100 `authentication_story` records in a column called `action_user_agent`.

```
preset = authentication_story
| limit 100
| windowcomp min(_time) by action_user_agent
```

multiply

Abstract

Learn more about the Cortex Query Language `multiply()` function that multiplies two integers.

Syntax

multiply (<string> | <integer>, <string> | <integer>)

Description

The `multiply()` function multiplies two positive integers. Parameters can be either integer literals, or integers as a string type such as might be contained in a data field.

Example

```
dataset = xdr_data
| alter mynum = multiply(action_file_size, 3)
| fields action_file_size, mynum
| filter action_file_size > 0
| limit 1
```

object_merge

Abstract

Learn more about the Cortex Query Language `object_merge()` function.

Syntax

object_merge(<obj1>, <obj2>, <obj3>, ...)

Description

The `object_merge()` function returns a new object, which is created from a merge of a number of objects. When there is a key name that is duplicated in any of the objects, the value in the new object is determined by the latter argument.

Example

Two objects are created and merged, where some key names are duplicated, including `name`, `last_name`, and `age`. Since the `name` value is the same for both objects, the same name is used in the new object. Yet, the `last_name` and `age` key values differ, so the values from the second object are used in the new object.

```
dataset = xdr_data
| alter
  obj1 = object_create("name", "jane", "last_name", "doe", "age", 33),
  obj2 = object_create("name", "jane", "last_name", "simon", "age", 34, "city", "new-york")
| alter result = object_merge(obj1, obj2)
| fields result
```

The function returns the following new object in the RESULT column of the results table:

```
{"name": "jane", "last_name": "simon", "age": 34, "city": "new-york"}
```

object_create

Abstract

Learn more about the Cortex Query Language `object_create()` function.

Syntax

```
object_create ("<key1>", "<value1>", "<key2>", "<value2>",...)
```

Description

The `object_create()` function returns an object based on the given parameters defined for the key and value pairs. Accepts n > 1 even number of parameters.

Example

Returns a final object to a field called `a` that contains the key and value pair `{“2”:“password”}`, where the "password" value is comprised by joining 2 values together.

```
dataset = xdr_data
| alter a = object_create("2", concat("pass", "word"))
| fields a
```

parse_epoch

Abstract

Learn more about the Cortex Query Language `parse_epoch()` function that returns a Unix epoch TIMESTAMP object.

Syntax

```
parse_epoch("<format string>", <timestamp field>[, "<time zone>",] ["<time unit>"])
```

Description

The `parse_epoch()` function returns a Unix epoch TIMESTAMP object after converting a string representation of a timestamp. The `<time zone>` offset is optional to configure using an hours offset, such as “+08:00”, or using a time zone name from the List of Supported Time Zones, such as "America/Chicago". When you do not configure a timezone, the default is `UTC`. The `<time unit>` is optional to configure and indicates whether the Unix epoch integer value represents seconds, milliseconds, or microseconds. These values are supported, and the default is used when none is configured:

- SECONDS (default)
- MILLIS
- MICROS

The order of the `<time zone>` and `<time unit>` matters. The `<time zone>` must be defined first followed by the `<time unit>`. If the `<time zone>` is set after the `<time unit>`, the default time zone is used and the configured value is ignored.

Examples

With a time zone configured:

Returns a maximum of 100 `xdr_data` records, which includes a timestamp field called `new_time` in the format `MMM dd YYYY HH:mm:ss`, such as `Dec 25th 2008 04:30:00`. This `new_time` field is comprised by taking a character string representation of a timestamp "Thu Dec 25 07:30:00 2008" and adding to it +03:00 hours as the time zone format. This string timestamp is then converted to a Unix epoch TIMESTAMP object in milliseconds using the `parse_epoch` function, and this resulting value is converted to the final timestamp using the to_timestamp function.

```
dataset = xdr_data
| alter new_time = to_timestamp(parse_epoch("%c", "Thu Dec 25 07:30:00 2008", "+3", "millis"))
| fields new_time
```

- | limit 100

Without a time zone or time unit configured:

Returns a maximum of 100 `xdr_data` records, which includes a timestamp field called `new_time` in the format `MMM dd YYYY HH:mm:ss`, such as `Dec 25th 2008 04:30:00`. This `new_time` field is comprised by taking a character string representation of a timestamp "Thu Dec 25 07:30:00 2008" and adding to it a UTC time zone format (default when none configured). This string timestamp is then converted to a Unix epoch TIMESTAMP object in seconds (default when none configured) using the `parse_epoch` function, and this resulting value is converted to the final timestamp using the to_timestamp function.

```
dataset = xdr_data
| alter new_time = to_timestamp(parse_epoch("%c", "Thu Dec 25 07:30:00 2008"))
| fields new_time
```

- | limit 100

parse_timestamp

Abstract

Learn more about the Cortex Query Language `parse_timestamp()` function that returns a TIMESTAMP object.

Syntax

```
parse_timestamp("<format time string>", "<time string>" | format_string(<time field>) | <time string field>)
parse_timestamp("<format time string>", "<time string>" | format_string(<time field>) | <time string field>, "<time zone>")
```

Description

The `parse_timestamp()` function returns a TIMESTAMP object after converting a string representation of a timestamp. The `<time zone>` offset is optional to configure using an hours offset, such as “+08:00”, or using a time zone name from the List of Supported Time Zones, such as "America/Chicago". The `parse_timestamp()` function can include both an alter stage and format_string function. For more information, see the examples below. The `format_string` function contains the format elements that define how the `parse_timestamp` string is formatted. Each element in the `parse_timestamp` string must have a corresponding element in `format_string`. The location of each element in the `format_string` must match the location of each element in `parse_timestamp`.

Examples

Without a time zone configured

Returns a maximum of 100 `microsoft_dhcp_raw` records, which includes a TIMESTAMP object in the `p_t_test` field in the format MMM dd YYYY HH:mm:ss, such as Jun 25th 2021 18:31:25. This format is detailed in the `format_string` function, which includes merging both the `date` and `time` fields.

```
dataset = microsoft_dhcp_raw
| alter p_t_test = parse_timestamp("%m/%d/%Y %H:%M:%S", format_string("%s %s", date, time))
| fields p_t_test
| limit 100
```

-

With a time zone name configured

Returns a maximum of 100 `microsoft_dhcp_raw` records, which includes a TIMESTAMP object in the `p_t_test` field in the format MMM dd YYYY HH:mm:ss, such as Jun 25th 2021 18:31:25. This format is detailed in the `format_string` function, which includes merging both the `date` and `time` fields, and includes a "Asia/Singapore" time zone.

```
dataset = microsoft_dhcp_raw
| alter p_t_test = parse_timestamp("%m/%d/%Y %H:%M:%S", format_string("%s %s", date, time), "Asia/Singapore")
| fields p_t_test
| limit 100
```

-

With a time zone configured using an hours offset

Returns a maximum of 100 `microsoft_dhcp_raw` records, which includes a TIMESTAMP object in the `p_t_test` field in the format MMM dd YYYY HH:mm:ss, such as Jun 25th 2021 18:31:25. This format is detailed in the `format_string` function, which includes merging both the `date` and `time` fields, and includes a time zone using an hours offset of “+08:00”.

```
dataset = microsoft_dhcp_raw
| alter p_t_test = parse_timestamp("%m/%d/%Y %H:%M:%S", format_string("%s %s", date, time), "+08:00")
| fields p_t_test
| limit 100
```

-

Convert a time string that contains milliseconds
Returns a single `xdr_data` record, which includes both, a manually added time string, "Jun 25 2024 18:31:25.723", in the `time_string` field and a TIMESTAMP object in the `p_t_test` field, such as Jun 25 2024 18:31:25, as the result of the `parse_timestamp()` function. Notice that the format element `%E*S` is used to capture seconds including any level of factional precision, such as milliseconds.

```
dataset = xdr_data
| limit 1
| alter time_string = "Jun 25 2024 18:31:25.723"
| alter p_t_test = parse_timestamp("%h %d %Y %H:%M:%E3S", time_string)
```

- | fields p_t_test, time_string

pow

Abstract

Learn more about the Cortex Query Language `pow()` function that returns the value of a number raised to the power of another number.

Syntax

```
pow (<x,n>)
```

Description

The `pow()` function returns the value of a number (`x`) raised to the power of another number (`n`).

rank

Abstract

Learn more about the Cortex Query Language `rank()` numbering function that is used with a `windowcomp` stage.

Syntax

```
windowcomp rank() [by <field> [,<field>,...]] sort [asc|desc] <field1> [, [asc|desc] <field2>,...] [as <alias>]
```

Description

The `rank()` function is a numbering function that is used in combination with a windowcomp stage. This function is used to return a single value for the ordinal (1-based) rank for each row in the group of rows using a combination of the `by` clause and `sort` (mandatory).

Example

Return an average ranking for the avgerage CPU usage on `metric_type=HOST`. Allows you to see changes in the CPU usage compared to all hosts in the environment. The query returns a maximum of 100 `it_metrics` records. The results are ordered by `ft` in decending order in the `rank` column.

```
 dataset = it_metrics
| filter metric_type = HOST
| alter cpu_avg_str = to_string(cpu_avg)
| alter ft = date_floor(_time, "w")
| alter dt = date_floor(_time, "d")
| limit 100
| windowcomp rank() by ft sort desc cpu_avg_str as rank
| filter (agent_hostname contains $host_name)
| comp avg(rank) by dt
```

regexcapture

Abstract

Learn more about the Cortex Query Language `regexcapture()` function used in Parsing Rules to extract data from fields using regular expression named groups from a given string.

The `regexcapture()` function is only supported in the XQL syntax for Parsing Rules.

Syntax

```
regexcapture(<field>, "<pattern>")
```

Description

In Parsing Rules, the `regexcapture()` function is used to extract data from fields using regular expression named groups from a given string and returns a JSON object with captured groups. This function can be used in any section of a Parsing Rule. The `regexcapture()` function is useful when the regex pattern is not identical throughout the log, which is required when using the regextract function.

XQL uses RE2 for its regular expression implementation. When using the `(?i)` syntax for case-insensitive mode in your query, this syntax should be added only once at the beginning of the inline regular expression.

Example

Parsing Rule to ceate a dataset called `my_regexcapture_test`, where the vendor and product that the specified Parsing Rules applies to is called `regexcapture_vendor` and `regexcapture_product`. The output results includes a new field called

`regexcaptureResult`, which extract data from the `_raw_log` field using regular expression named groups as defined and returns the captured groups.

Parsing Rule:

```
[INGEST:vendor="regexcapture_vendor", product="regexcapture_product", target_dataset="my_regexcapture_test"]
alter regexcaptureResult = regexcapture(_raw_log,"^(?P<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}) - (?P<user>\w+) \[(?P<timestamp>.+)\]
(?P<request>.+) (?P<status>\d{3}) (?P<bytes>\d+)");
```

Log:

```
192.168.1.1 - john [10/Mar/2024:12:34:56 +0000] GET /index.html HTTP/1.1 200 1234
```

XQL Query:

For the `my_regexcapture_test` dataset, returns the `regexcaptureResult` field output.

```
dataset = my_regexcapture_test
| fields regexcaptureResult
```

regexcaptureResult field output:

```
{
  "ip": "192.168.1.1",
  "user": "john",
  "timestamp": "10/Mar/2024:12:34:56 +0000",
  "request": "GET /index.html HTTP/1.1",
  "status": "200",
  "bytes": "1234"
}
```

regextract

Abstract

Learn more about the Cortex Query Language `regextract()` function that uses regular expressions to assemble an array of matching substrings from a string.

Syntax

```
regextract (<string_value>, <pattern>)
```

Description

The `regextract()` function accepts a string and a regular expression, and it returns an array containing substrings that match the expression.

Cortex Query Language (XQL) uses RE2 for its regular expression implementation. While capturing multiple groups is unsupported, capturing one group in queries is supported.

When using the `(?i)` syntax for case-insensitive mode in your query, this syntax should be added only once at the beginning of the inline regular expression.

Capturing multiple groups is supported in Parsing Rules when using the regexcapture function.

Examples

Without a capturing group

Extract the `Account Name` from the `action_evtlog_message`. Use the arrayindex and split functions to extract the actual account name from the array created by `regextract`.

```
dataset = xdr_data
| fields action_evtlog_message as aem
| filter aem != null
| alter account_name =
    arrayindex(
        split(
            arrayindex(
                regextract(aem, "Account Name:\t\t.*\r\n")
            ,0)
        , ":")
    ,1)
| filter account_name != null
| limit 10
```

Using one capturing group

Extract from the `log_example` field all of the values included for the id objects.

```
dataset = xdr_data
| limit 1
| alter
    log_example = "{\"events\":[{\"id\": \"1\", \"type\": \"process\", \"size\": 123, \"processID\": 40540},{\"id\": \"2\", \"type\": \"request\", \"size\": 456, \"srcOS\": \"MAC\"}],\"host\": \"LocalHost\",\"date\": {\"day\": 4, \"month\": 7, \"year\": 2024},\"tags\":[\"agent\", \"auth\", \"low\"]}"
| alter
    one_capture_group_usage = regextract(log_example, "\"id\":\s*\"([^\"]+)\"")
| fields log_example, one_capture_group_usage
```

replace

Abstract

Learn more about the Cortex Query Language `replace()` function that performs a substring replacement.

Syntax

```
replace (<field>, "<old_substring>", "<new_string>")
```

Description

The `replace()` function accepts a string field, and replaces all occurrences of a substring with a replacement string.

Examples

If '.exe' is present on the `action_process_image_name` field value, replace that substring with an empty string. This example uses the if and lowercase functions, as well as the contains operator to perform the conditional check.

```
dataset = xdr_data
| fields action_process_image_name as apin
| filter apin != null
| alter remove_exe_process = if(lowercase(apin) contains ".exe",
                          replace(lowercase(apin),".exe",""),
                          lowercase(apin))
| limit 10
```

See also the ltrim, rtrim, trim function example.

replex

Abstract

Learn more about the Cortex Query Language `replex()` function that uses a regular expression to identify and replace substrings.

Syntax

```
replex (<string>, <pattern>, <new_string>)
```

Description

The `replex()` function accepts a string, and then uses a regular expression to identify a substring, and then replaces matching substrings with a new string.

XQL uses RE2 for its regular expression implementation.

Examples

For any `agent_id` that contains a dotted decimal IP address, mask the IP address. Use the dedup stage to reduce the result set to first-seen `agent_id` values.

```
dataset = xdr_data
| fields agent_id
| alter clean_agent_id = replex(agent_id,
                          "[\d]+\.[\d]+\.[\d]+\.[\d]+",
                          "xxx.xxx.xx.xx")
| dedup agent_id by asc _time
```

round

Abstract

Learn more about the Cortex Query Language `round()` function that returns the input value rounded to the nearest integer.

Syntax

```
round (<float> | <integer>)
```

Description

The `round()` function accepts either a float or an integer as an input value, and it returns the input value rounded to the nearest integer.

Example

```
dataset = xdr_data
| alter mynum = divide(action_file_size, 7)
| alter mynum2 = round(mynum)
| fields action_file_size, mynum, mynum2
| filter action_file_size > 3
| limit 1
```

row_number

Abstract

Learn more about the Cortex Query Language `row_number()` numbering function that is used with a `windowcomp` stage.

Syntax

```
windowcomp row_number() [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [as <alias>]
```

Description

The `row_number()` function is a numbering function that is used in combination with a windowcomp stage. This function is used to return a single value for the sequential row ordinal (1-based) for each row from a group of rows using a combination of the `by` clause and `sort`.

Example

Return a single value for the sequential row ordinal (1-based) for each row in the group of rows. The query returns a maximum of 100 `xdr_data` records. The results are ordered by the `source_ip` in ascending order in the `row_number_dns_query_name` column.

```
dataset = xdr_data
| limit 100
| windowcomp row_number() sort source_ip as row_number_dns_query_name
```

split

Abstract

Learn more about the Cortex Query Language `split()` function that splits a string and returns an array of string parts.

Syntax

split (<value> [, <string_delimiter>])

Description

The `split()` function splits a string using an optional delimiter, and returns the resulting substrings in an array. If no delimiter is specified or an empty string is specified as a delimiter, a space (' ') is used.

Examples

Split IP addresses into an array, each element of the array containing an IP octet.

```
dataset = xdr_data
| fields action_local_ip  as alii
| alter ip_octets = split(alii, ".")
| limit 10
```

stddev_population

Abstract

Learn more about the Cortex Query Language `stddev_population()` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp stddev_population(<field>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

windowcomp stage

windowcomp stddev_population(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `stddev_population()` function is used to return a single population (biased) variance value of a field for a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `stddev_population` aggregation function is used with a comp stage, the function returns a single population (biased) variance value of a field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `stddev_population` statistical aggregate function is used with a windowcomp stage, the function returns a single population (biased) variance value of a field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.

Examples

comp example

Calculates a maximum of 100 `metrics_source` records, where the `_broker_device_id` is `655AYUWF`, and include a single population (biased) variance value of the `total_size_rate` field for a group of rows.

```
dataset = metrics_source
| filter _broker_device_id = "655AYUWF"
| comp stddev_population(total_size_rate)
| limit 100
```

windowcomp example

Return maximum of 100 `metrics_source` records and include a single population (biased) variance value of the `total_size_rate` field for each row in the group of rows, for all records that contain matching values in the `_broker_device_id` field. The results are provided in the `stddev_population` column.

```
dataset = metrics_source
| limit 100
| windowcomp stddev_population(total_size_rate) by _broker_device_id as `stddev_population`
```

stddev_sample

Abstract

Learn more about the Cortex Query Language `stddev_sample()` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp stddev_sample(<field>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

windowcomp stage

windowcomp stddev_sample(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `stddev_sample()` function is used to return a single sample (unbiased) standard deviation value of a field for a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `stddev_sample` aggregation function is used with a comp stage, the function returns a single sample (unbiased) standard deviation value of a field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `stddev_sample` statistical aggregate function is used with a windowcomp stage, the function returns a single sample (unbiased) standard deviation value of a field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.

Examples

comp stage example

Calculate a maximum of 100 `metrics_source` records, where the `_broker_device_ip` is `172.16.1.25`, and include a single sample (unbiased) standard deviation value of the `total_size_bytes` field for a group of rows.

```
dataset = metrics_source
| filter _broker_device_ip = "172.16.1.25"
| comp stddev_sample(total_size_bytes)
| limit 100
```

windowcomp stage example

Return a maximum of 100 `metrics_source` records and include a single sample (unbiased) standard deviation value of the `total_size_rate` field for each row in the group of rows, for all records that contain matching values in the `_broker_device_id` field. The results are provided in the `stddev_sample` column.

```
dataset = metrics_source
| limit 100
| windowcomp stddev_sample(total_size_rate) by _broker_device_id as `stddev_sample`
```

string_count

Abstract

Learn more about the Cortex Query Language `string_count()` function that returns the number of times a substring appears in a string.

Syntax

```
string_count (<string>, <pattern>)
```

Description

The `string_count()` function returns the number of times a substring appears in a string.

Example

```
dataset = xdr_data
| fields actor_primary_username as apu
| filter string_count(apu, "e") > 1
```

subtract

Abstract

Learn more about the Cortex Query Language `subtract()` function that subtracts two integers.

Syntax

```
subtract (<string1> | <integer1>, <string2> | <integer2>)
```

Description

The `subtract()` function subtracts two positive integers by subtracting the second argument from the first argument. Parameters may be either integer literals, or integers as a string type such as might be contained in a data field.

Example

```
dataset = xdr_data
| alter mynum = subtract(action_file_size, 3)
```

```
| fields action_file_size, mynum
| filter action_file_size > 3
| limit 1
```

sum

Abstract

Cortex Query Language `sum` function used with both `comp` and `windowcomp` stages.

Syntax

comp stage

comp sum(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

windowcomp

windowcomp sum(<field>) [by <field> [,<field>,...]] [sort [asc|desc] <field1> [, [asc|desc] <field2>,...]] [between 0|null|<number>|-<number> [and 0|null|<number>|-<number>] [frame_type=range]] [as <alias>]

Description

The `sum()` function is used to return the sum of an integer field over a group of rows. The function syntax and application is based on the preceding stage:

comp stage

When the `sum` aggregation function is used with a comp stage, the function returns a single sum of an integer field for a group of rows, for all records that contain matching values for the fields identified in the `by` clause.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

windowcomp stage

When the `sum` aggregate function is used with a windowcomp stage, the function returns a single sum of an integer field for each row in the group of rows, for all records that contain matching values for the fields identified using a combination of the `by` clause, `sort`, and `between` window frame clause. The results are provided in a new column in the results table.

Examples

comp example

Return a single sum of the `action_total_download` field for a group of rows, for all records that have matching values for their `actor_process_image_path` and

`actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing a single value for the results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp sum(Download) as total_download by Process_Path, Process_CMD addrawdata = true as raw_data
```

windowcomp

Return the download to upload ratio per process. The query returns a maximum of 100 `xdr_data` records in new columns called `sum_upload` and `sum_download`.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download, action_total_upload as Upload
| filter Download > 0
| limit 100
| windowcomp sum(Download) by Process_Path, Process_CMD as sum_download
| windowcomp sum(Upload) by Process_Path, Process_CMD as sum_upload
| fields - Download ,Upload
| dedup Process_CMD, Process_Path, sum_download ,sum_upload
| alter ration = divide(sum_download ,sum_upload)
```

time_frame_end

Abstract

Learn more about the Cortex Query Language `time_frame_end()` function that returns the end time of the time range specified for the query.

Syntax

```
time_frame_end(<time frame>)
```

Description

The `time_frame_end()` function returns the timestamp object for the string representation of the end of the time frame configured for the query in the format MMM dd YYYY HH:mm:ss, such as Jun 8th 2022 15:20:06. You can configure the time frame using the config timeframe function, where the range can be relative or exact.

If the time frame is relative, for example last 24H, the function returns the current_time. This function is useful when the query uses a custom time frame whose end time is in the past.

Example 1 - Relative Time

For the last 5 days from when the query is sent, returns a maximum of 100 `xdr_data` records with the events of the _time field with a new field called "x". The "x" field lists the final timestamp

at the end of 5 days from when the query was sent for the events in descending order. For more information on this relative timeframe range, see the config timeframe function.

```
config timeframe = 5d
| dataset = xdr_data
| alter x = time_frame_end()
| fields x
| sort desc x
```

Example 2 - Relative Time

For the last 5 days from when the query is run until now, returns a maximum of 100 xdr_data records with the events of the _time field with a new field called "x". The "x" field lists the final timestamp at the end of 5 days from when the query runs for the events in descending order. For more information on this relative time frame range, see the config timeframe function.

```
config timeframe = between "5d" and "now"
| dataset = xdr_data
| alter x = time_frame_end()
| fields x
| sort desc
```

timestamp_diff

Abstract

Learn more about the Cortex Query Language `timestamp_diff()` function that returns the difference between two timestamp objects.

Syntax

```
timestamp_diff (<timestamp1>, <timestamp2>, <part>)
```

Description

The `timestamp_diff()` function returns the difference between two timestamp objects. The units used to express the difference is identified by the `part` parameter. The second timestamp is subtracted from the first timestamp. If the first timestamp is greater than the second, a positive value is returned. If the result of this function is between 0 and 1, 0 is returned.

Supported parts are:

- `DAY`
- `HOUR`
- `MINUTE`
- `SECOND`
- `MILLISECOND`
- `MICROSECOND`

Example

```
dataset = xdr_data
| filter story_publish_timestamp != null
| alter ts = to_timestamp(story_publish_timestamp, "MILLIS")
| alter ct = current_time()
| alter diff = timestamp_diff(ct, ts, "MINUTE")
| fields ts, ct, diff
| limit 1
```

timestamp_seconds

Abstract

Learn more about the Cortex Query Language `timestamp_seconds()` function.

Syntax

```
timestamp_seconds (<integer>)
```

Description

The `timestamp_seconds()` function converts an epoch time Integer value in seconds to a TIMESTAMP compatible value.

Endpoint Detection and Response (EDR) columns store epoch milliseconds values so this function is more useful for values that you insert.

Example

Display a human-readable timestamp for the `action_file_access_time` field.

```
alter access_timestamp = timestamp_seconds(1611882205) | limit 1
```

to_boolean

Abstract

Learn more about the Cortex Query Language `to_boolean()` function that converts a string to a boolean.

Syntax

```
to_boolean(<string>)
```

Description

The `to_boolean()` function converts a string that represents a boolean to a boolean value.

The input value to this string must be either `TRUE` or `FALSE`, case insensitive.

Example

to_epoch

Abstract

Learn more about the Cortex Query Language `to_epoch()` function that converts a timestamp value for a field or function to the Unix epoch timestamp format.

Syntax

```
to_epoch (<timestamp>, <time unit>)
```

Description

The `to_epoch()` function converts a timestamp value for a particular field or function to the Unix epoch timestamp format. This function requires a `<time unit>` value, which indicates whether the integer value for the Unix epoch timestamp format represents seconds (default), milliseconds, or microseconds. If no `<time unit>` is configured, the default is used. Supported values are:

- SECONDS
- MILLIS
- MICROS

Example

Returns a maximum of 100 `xdr_data` records with the events of the `_time` field, which includes a timestamp field in the Unix epoch format called `ts`. The ts field contains the equivalent Unix epoch values in milliseconds for the timestamps listed in the `_time` field.

```
dataset = xdr_data
| filter _time != null
| alter ts = to_epoch(_time, "MILLIS")
| fields ts
| limit 100
```

to_float

Abstract

Learn more about the Cortex Query Language `to_float()` function that converts a string to a floating point number.

Syntax

```
to_float(<string>)
```

Description

The `to_float()` function converts a string that represents a number to a floating point number. This function is identical to the to_number function.

Examples

Display the first 10 IP addresses that begin with a value greater than `192`. Use the split function to split the IP address by '.', and then use the arrayindex function to retrieve the first value in the resulting array. Convert this to a number and perform an arithmetic compare to arrive at a result set.

```
dataset = xdr_data
| fields action_local_ip  as alii
| filter to_float(arrayindex(split(alii, "."),0))  > 192
| limit 10
```

to_integer

Abstract

Learn more about the Cortex Query Language `to_integer()` function that converts a string field to an integer.

Syntax

```
to_integer(<string>)
```

Description

The `to_integer()` function converts a string value that represents a number of a given field to an integer. A good application of using the `to_integer` function is when querying for USB vendor IDs and USB product IDs, which are usually provided in a hex format.

It is an error to provide a string to this function that contains a floating point number.

Examples

Display the first 10 IP addresses that begin with a value greater than 192. Use the split function to split the IP address by '.', and then use the arrayindex function to retrieve the first value in the resulting array. Convert this to a number and perform an arithmetic compare to arrive at a result set.

```
dataset = xdr_data
| fields action_local_ip  as alii
| filter to_integer(arrayindex(split(alii, "."),0))  > 192
| limit 10
```

to_json_string

Abstract

Learn more about the Cortex Query Language `to_json_string()` function that accepts all data types and returns its contents as a JSON formatted string.

Syntax

```
to_json_string(<data type>)
```

Description

The `to_json_string()` function accepts all data types, such as integers, booleans, strings, and returns it as a JSON formatted string. This function always returns a string. When the input is an object or an array, the function returns a JSON formatted string of the input. When the input string is a string, it returns the string as is. You can then use the JSON formatted string or string returned by this function with the json_extract, json_extract_array, and json_extract_scalar functions.

Examples

Return the `action_file_device_info` field in JSON format.

```
dataset = xdr_data
| fields action_file_device_info as afdi
| filter afdi != null
| alter the_json_string = to_json_string(afdi)
| limit 10
```

to_number

Abstract

Learn more about the Cortex Query Language `to_number()` function that converts a string to a number.

Syntax

```
to_number (<string>)
```

Description

The `to_number()` function converts a string that represents a number to a floating point number. This function is identical to the to_float function.

Examples

Display the first 10 IP addresses that begin with a value greater than 192. Use the split function to split the IP address by '.', and then use the arrayindex function to retrieve the first value in the resulting array. Convert this to a number and perform an arithmetic compare to arrive at a result set.

```
dataset = xdr_data
| fields action_local_ip  as alii
| filter to_number(arrayindex(split(alii, "."),0))  > 192
| limit 10
```

to_string

Abstract

Learn more about the Cortex Query Language `to_string` function that converts a number value to a string.

Syntax

```
to_string (<field>)
```

Description

The `to_string()` function converts a number value of a given field to a string.

Examples

Display the first non-NULL `action_boot_time` field value. In a second column called `abt_string`, use the concat function to prepend "str: " to the value, and then display it.

```
dataset = xdr_data
| fields action_boot_time as abt
| filter abt != null
| alter abt_string = concat("str: ", to_string(abt))
| limit 1
```

to_timestamp

Abstract

Learn more about the Cortex Query Language `to_timestamp()` function that converts an integer to a timestamp.

Syntax

```
to_timestamp (<integer>, <units>)
```

Description

The `to_timestamp()` function converts an integer to a timestamp. This function requires a `units` value, which indicates whether the integer represents seconds, milliseconds, or microseconds since the Unix epoch. Supported values are:

- `SECONDS`
- `MILLIS`
- `MICROS`

Example

```
dataset = xdr_data
| filter story_publish_timestamp != null
| alter ts = to_timestamp(story_publish_timestamp, "MILLIS")
| fields ts
```

uppercase

Abstract

Learn more about the Cortex Query Language `uppercase()` function that converts a string field to all uppercase letters.

Syntax

```
uppercase (<string>)
```

Description

The `uppercase()` function converts a string field value to all uppercase.

Examples

Convert all `actor_process_image_name` field values that are not null to uppercase, and return a list of unique values.

```
dataset = xdr_data
| fields actor_process_image_name as apin
| dedup apin by asc _time
| filter apin != null
| alter apin = uppercase(apin)
```

values

Abstract

Cortex Query Language `comp values` aggregate returns an array for all the values seen for the field in the result set.

Syntax

comp values(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

Description

The `values` aggregation is a comp function that returns an array of all the values found for a given field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause. The array values are all non-null. Each value appears in the array only once, even if a given value repeats multiple times in the result set. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Example

Return an array containing all the values seen for the `action_total_download` field for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results. In addition, this example contains a number of fields defined as aliases: `actor_process_image_path` uses the alias `Process_Path`, `actor_process_command_line` uses the alias `Process_CMD`, `action_total_download` uses the alias `Download`, and `Download` uses the alias `values_download`.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp values(Download) as values_download by Process_Path, Process_CMD addrawdata = true as raw_data
```

var

Abstract

Learn more about the Cortex Query Language `var` aggregate comp function that returns the variance value of a field in the result set.

Syntax

comp var(<*field*>) [as <*alias*>] by <*field_1*>,<*field_2*> [addrawdata = true|false [as <*target field*>]]

Description

The `var` aggregation is a comp function that returns a single variance value of a field over a group of rows, for all records that contain matching values for the fields identified in the `by` clause. This function is used in combination with a `comp` stage.

In addition, you can configure whether the raw data events are displayed by setting `addrawdata` to either `true` or `false` (default), which are used to configure the final `comp` results. When including raw data events in your query, the query runs for up to 50 fields that you define and displays up to 100 events.

Example

Return the variance of the `action_total_download` field for all records that have matching values for their `actor_process_image_path` and `actor_process_command_line` values. The query calculates a maximum of 100 `xdr_data` records and includes a `raw_data` column listing the raw data events used to display the final `comp` results.

```
dataset = xdr_data
| fields actor_process_image_path as Process_Path, actor_process_command_line as Process_CMD, action_total_download as Download
| filter Download > 0
| limit 100
| comp var(Download) as variance_download by Process_Path, Process_CMD
addrawdata = true as raw_data
```