SEARCH QUERY LANGUAGE GUIDE

Version 2

Document last updated: July 4, 2024

Reference: sls-en_search_query_language_gde

Table of contents

Change log 5

Getting started 6

Simple Search 7

- Single word 7
- Multiple words 7
- Phrases 7
- Field values 7
- Logical operators 8
  - And 8
  - Or 8
  - Not 8
- Parentheses 9
- Wildcards 9
- Step 9
- Lower and Upper 10
- Time Functions 10
  - second 10
  - minute 10
  - hour 11
  - day 11
  - day of week 11
  - month 11
- List 12
- Table 12

Aggregators 14

- chart 14
- timechart 16
- Available Aggregators 17
  - avg() 17
  - count() 17
  - distinct_count() 17
  - distinct_list() 18
  - list() 19
  - max() and min() 20
  - sum() 20
  - var() 21

One-to-One Commands 22

- rex 22
- norm 23
- fields 23
- rename 24

Process Commands 25

- AsciiConverter 25
- Clean Char 25
- Codec 26

Compare 26
Compare Network 27
Count Char 27
CountOf 28
Current Time 29
DatetimeDiff 30
Difference 30
DNS Cleanup 31
DNS Process 31
Domain Lookup 32
Entropy 32
Eval 33
Experimental Median Quartile Quantile 33
GEOIP 34
Grok 35
InRange 36
IP Lookup 36
JQ Parser 37
JSON Expand 37
JSON Parser 38
ListLength 40
ListPercentile 41
Next 41
Percentile 42
Process lookup 42
Regex 42
SortList 43
String Concat 43
Summation 43
toList 44
toTable 44
WhoIsLookup 45

Filtering Commands 46

search 46
filter 46
latest 47
order by 47
limit <number> 47

Pattern Finding 49

Single Stream 49
Multiple Streams 50
Left Join 50
Right Join 51
Join 51
Followed by 51

Chaining of commands 53

Additional Notes 54

Process or Count 54

Conditional Expression 54
Forward Slash Expression 54
norm 54
timechart 54
Capturing normalized field values 54
Grok Patterns 55

Further reading 59

Change log

Date	Description
July 4, 2024	New document

Getting started

Welcome to the SLS version 2 Search Query Language Guide.

SLS's **Query Language** is extensive, intuitive, and user-friendly. It covers all the search commands, functions, arguments, and clauses. You can search the log messages in various formats depending on the query you use.

SLS also supports chaining of commands and multi-line queries. Use a pipe (|) to chain the commands and press **Shift + Enter** to add a new line in the query. The search keywords are not case-sensitive.

> **NOTE**
> The examples of some search queries provided in this section may not yield any result as the relevant logs may not be available in your system.

This guide provides the following information that you need to use the SLS Query Language:

- Learn about the types of simple queries to familiarize yourself with the SLS Query Language.
- Learn how to aggregate fields with **chart** and **timechart** commands.
- Learn about the one-to-one commands.
- Learn about the process commands.
- Learn how to filter the search results.
- Learn how to find one or multiple streams and patterns of data to correlate a particular event.
- Learn how to chain multiple commands into a single query.

In this document, Stormshield Log Supervisor is referred to in its short form SLS. Images used in this document are from the partner vendor's (Logpoint) software program. In your SLS, the graphics may vary but user experience is exactly the same.

Simple Search

You can use the following types of simple queries to familiarize yourself with the SLS Query Language.

Single word

Single word search is the most basic search that you can run in SLS. Enter a **single word** in the **Query Bar** to retrieve the logs containing the word.

```
login
```

This query searches for all the logs containing the word **login** in the message.

Multiple words

Searching with multiple words lets you search the original logs using a combination of words. For searches with multiple words, only the logs containing all the words are displayed.

NOTE
The order of the words is not important.

```
account locked
```

This query searches for all the logs containing both the search terms **account** and **locked** in the message.

Phrases

Phrase Search lets you search the exact phrase in the logs. You must enclose the words inside double-quotes (" ").

NOTE
The order of the words is important.

```
"account locked"
```

This query searches for all the logs containing the exact phrase **account locked**.

Field values

The normalized logs contain information in key-value pairs. You can use these pairs directly in the log search. To see all the logs from the user **Bob**, use the following query:

```
user = Bob
```

This query searches for all the logs from the user **Bob**.

```
device_ip = 192.168.2.1
```

This query searches for all the logs coming from the device with the IP Address **192.168.2.1**.

You can combine multiple field value pairs as:

```
device_ip = 192.168.2.1 sig_id = 10051
```

You can also combine this with a simple query as:

```
login device_ip = 192.168.2.1 sig_id = 10051
```

Logical operators

You can use various keywords to perform logical operations in the SLS search query.

And

Use the logical operator **and** to search for the slogs containing both the specified parameters.

```
login and successful
```

This query searches for all the messages containing the word **login and** the word **successful**.

The **and** operator can also be used for key-value search queries as follows:

```
login and device_ip=192.168.2.2
```

Or

Use the logical operator **or** to search for the logs containing either of the specified parameters.

```
login or logout
```

This query searches for all the messages containing either the word **login or** the word **logout**.

This operator can also be used with the key-value search query as follows:

```
device_ip = 192.168.2.1 or device_ip = 127.0.0.1
```

Not

You can use the hyphen (-) symbol for the logical negation in your searches.

```
login -Bob
```

This query searches for the log messages containing the word **login** but not the word **Bob**.

```
-device_ip = 192.168.2.243
```

This query returns the logs containing all the **device_ip**s except **192.168.2.243**.

NOTE

- While searching with field-names, you can also use **!=** and **NOT** to denote negation.

```
device_ip != 192.168.2.243
NOT device_ip = 192.168.2.243
```

- By default, the **or** operator binds stronger than the **and** operator. When performing the **login or logout and MSWinEventLog**, SLS returns the log messages containing either **login** or **logout** with **MsWinEventLog**.

Parentheses

In SLS, the **or** operator has a higher precedence by default. You can use parentheses to override the default binding behavior when using the logical operators in the search query.

```
"login failed" or (denied and locked)
```

This query returns the log messages containing **login failed** or both **deniedandlocked**.

Wildcards

You can use wildcards as replacements for a part of the query string. Use the following characters as wildcards:

- **?** - Replacement for single character.
- ***** - Replacement for multiple characters.

If you want all the log messages containing the word **login** or **logon**, use the following:

```
log?n
```

NOTE
This query also searches for the log messages containing other variations such as **logan**, **logbn**, and **logcn**.

```
log*
```

This query returns the logs containing the words starting with **log** such as **sls**, **logout**, and **login**.

NOTE
You can also use **Wildcards** while forming a search query with field names. To get all the usernames that end in **t**, use the following. `username = *t`

Step

You can use the **step** function to group fields. To see the log messages with **destination_port** in steps of 100 as follows:

destination_port	count
0 - 100	50
100 - 200	32

```
step(destination_port,100) = 0 | chart count() by destination_port
```

This query searches for all the log messages containing the field **destination_port**, and groups them in steps of 100. The value at the end of the query specifies the starting value of the **destination_port** for grouping.

NOTE
You can use the **step** to group using multiple field names.

Lower and Upper

You can change type-case of your field values. Use the **lower** function to change the values to lower case. Similarly, use the **upper** function to change the field values to upper case. The **upper** and **lower** functions change the type-case of the values to the same case so that you can observe consistent results.

Use the **upper** and **lower** functions with **chart** and **timechart** commands.

```
| chart count() by upper(action)
```

```
| timechart count() by lower(action)
```

Time Functions

The **Time Functions** extract specied values from a time-based field. The following time functions are supported in the **Simple Search Query**:

- second
- minute
- hour
- day
- day of week
- month

The arguments taken by these functions are numeric. These functions parse **Unix Timestamps**.

In SLS, **col_ts** and **log_ts** carry Unix timestamps. However, you can create your own fields which contain the Unix timestamps using the **rex** or **norm** commands.

second

You can use the **second** function to search for the logs generated or collected in seconds.

The generic syntax for second is:

```
second(field) = value
```

The value for second ranges from 0 to 59.

```
second(log_ts) = 23
```

This query searches for the logs generated during the twenty third second.

minute

You can use the **minute** function to search for the logs generated or collected in minutes. The values for the minute range from 0 to 59.

```
minute(col_ts) = 2
```

This query searches for the logs generated during the second minute.

minute() can also be used in aggregation functions.

hour

You can use the **hour** function to search for the logs generated or collected in hours. The values for the hour range from 1 to 24.

Example:

```
hour(col_ts) = 1
```

This query displays the logs generated during the first hour.

day

You can use the **day** function to search for the logs generated or collected in days.

Example:

```
day(col_ts) = 4
```

This query displays the logs of the 4th day.

day of week

You can use the **day of week** function to search the logs for the specific day of the week. The value for **day_of_week** ranges from 1 (Sunday) to 7 (Saturday).

Example:

```
day_of_week(col_ts) = 7 OR day_of_week(col_ts) = 1
```

This query displays the logs in off days, i.e, Saturday and Sunday.

month

You can use the **month** function to search the logs generated or collected in months. The value of month ranges from 1 (January) to 12 (December).

Example:

```
month(col_ts) = 6
```

This query displays the log activity for June.

NOTE
You can use the relational operators (**>**, **<**, **=** and **!=**) with the time commands to create a sensible time-range for your search queries.

The following table summarizes the time functions:

Time functions	Working Examples	Value Range
second	second(cpl_ts) = 20	0 - 59
minute	minute(col_ts) = 18	0 - 59
hour	hour(col_ts) = 6	0 - 23
day	day(col_ts) = 14	1 - 31
day_of_week	day_of_week(col_ts) = 5	1 - 7 (Sun - Sat)

Time functions	Working Examples	Value Range
month	month(col_ts) = 11	1 - 12 (Jan - Dec)

List

You can create a static list with a number of values, and use this list in the search query instead of keying in all the values.

For example, if you create a list **EMPLOYEES** with the names of all the employees in a company, you can check whether a single user has logged into the system using the following query.

```
user in EMPLOYEES action=login
```

The search query matches the value of the field **user** with all the values in the EMPLOYEES list.

> **IMPORTANT**
> The name of the list must be provided in uppercase.

You can also use an **Inline List** while executing a search query.

The generic syntax for inline list is:

```
field in [value1, value2,....]
```

which is equivalent to **field = value1 OR field = value2**.

Example:

```
source_port in [21, 53, 88, 123]
```

In cases where the values have multiple words in the inline List, use quotation marks as shown below.

```
event in ["Process completed", "Process accomplished"]
```

Table

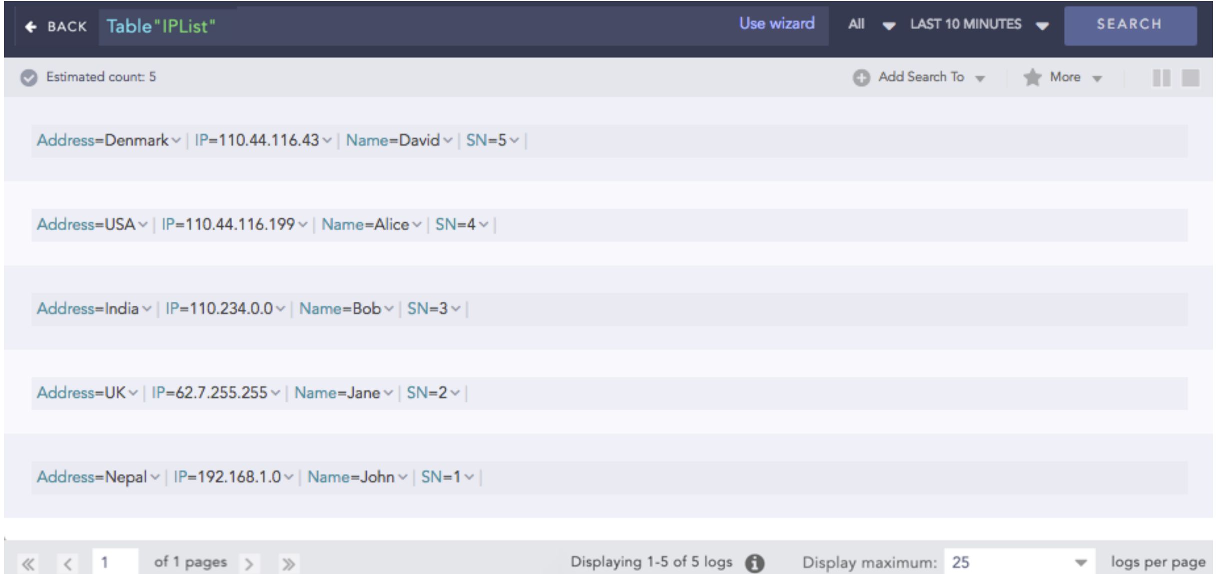
Tables are external file-formats which contain the information you may choose to associate with a search result. The file formats supported for the tables are CSV, ODBC, LDAP, and Threat Intelligence. The information obtained is prefixed with the table alias in the log messages.

For example:

IPList is a CSV table containing fields such as **Address**, **IP**, **Name**, and **SN**. To view the content of this external CSV table, use the following query:

```
table "IPList"
```

The following content is displayed:



To view all student entries in a table called **studentResult**, which contains **student_name**, **student_roll**, and **percentage** as fields, use:

```
table "studentResult"
```

To search for all the student entries in the table **studentResult** who have passed with distinction:

```
table "studentResult" percentage >= 80
```

To search for all the student entries in the table **studentResult** who have failed:

```
table "studentResult" percentage < 40
```

> **NOTE**
> In the **Data Privacy Module** enabled systems, when you use the **table** query, you can only see the values of the search results in the encrypted form. You cannot request a decryption for these values.

Aggregators

Aggregators are used with **chart** and **timechart** to aggregate fields. The search results can be formatted using fields, chart, or timechart commands.

- An aggregator displays 40 search results by default.
- Aggregators have an internal limit of 500K results by default. A single aggregator does not forward more than 500K results to subsequent aggregators or process commands. Use the :ref:limit command to set a higher limit of results to forward.
- Queries using an aggregator that results in large groupings can result in incomplete search results. To get complete results, use the :ref:order by to sort the search results in ascending order. There are also audit logs that you can use to check or confirm the results.
- Using free text queries within an aggregator results in raw log processing, a resource heavy operation. Only use free text within an aggregator when absolutely necessary.

chart

With **chart** command, you get log messages in a chart form. If you want to see all the messages containing **login** and group them by **device_ip**, use the following query.

```
login device_ip = * | chart count() by device_ip
```

This query searches for all the log messages containing the word **login**, and groups them by **device_ip**. It then displays the number of log messages for each **device_ip**.

You can also count by multiple fields. The log message count is then displayed for each field.

```
login | chart count() by destination_address, destination_port
```

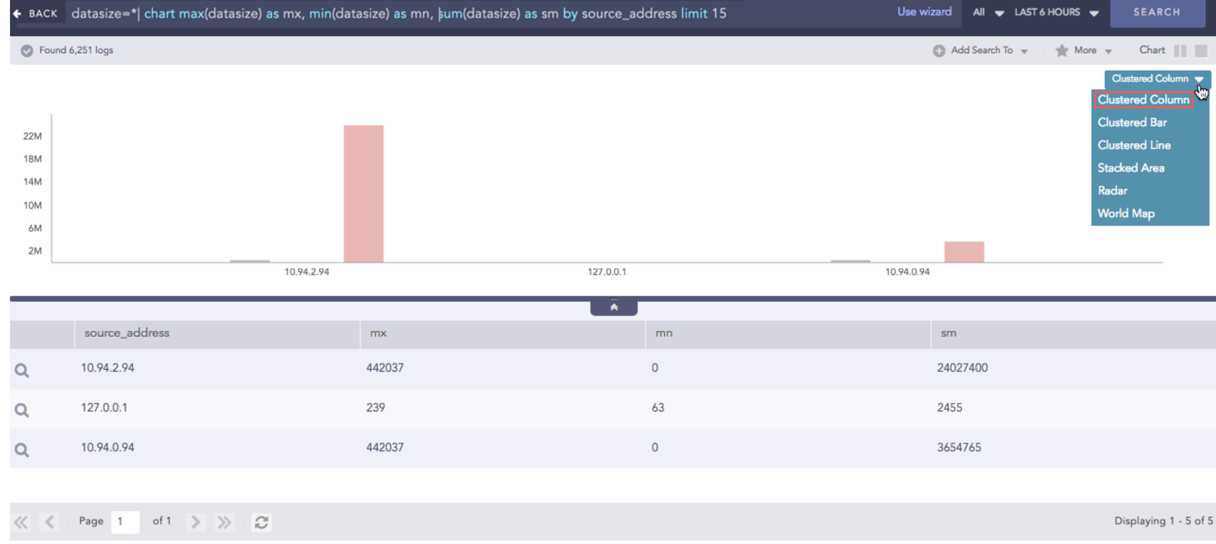
In this case, the count of the log messages for every combination of **destination_address** and **destination_port** is grouped and the corresponding count is shown.

You can use other aggregation functions such as **max** and **min** in place of **count**.

```
connection | chart max(datasize) by source_address
```

```
datasize=*| chart max(datasize) as mx, min(datasize) as mn, sum(datasize)
as sm by source_address limit 15
```

You can also display the chart in different forms such as Column, Bar, Line and Area.



BACK datasize=*| chart max(datasize) as mx, min(datasize) as mn, sum(datasize) as sm by source_address limit 15

Found 6,251 logs

Clustered Bar: Clustered Column, Clustered Bar, Clustered Line, Stacked Area, Radar, World Map

source_address	mx	mn	sm
10.94.2.94	442037	0	24027400
127.0.0.1	239	63	2455
10.94.0.94	442037	0	3654765

Page 1 of 1 — Displaying 1 - 5 of 5

BACK datasize=*| chart max(datasize) as mx, min(datasize) as mn, sum(datasize) as sm by source_address limit 15

Found 6,251 logs

Clustered Line: Clustered Column, Clustered Bar, Clustered Line, Stacked Area, Radar, World Map

source_address	mx	mn	sm
10.94.2.94	442037	0	24027400
127.0.0.1	239	63	2455
10.94.0.94	442037	0	3654765

Page 1 of 1 — Displaying 1 - 5 of 5

BACK datasize=*| chart max(datasize) as mx, min(datasize) as mn, sum(datasize) as sm by source_address limit 15

Found 6,251 logs

Stacked Area: Clustered Column, Clustered Bar, Clustered Line, Stacked Area, Radar, World Map

source_address	mx	mn	sm
10.94.2.94	442037	0	24027400
127.0.0.1	239	63	2455
10.94.0.94	442037	0	3654765

Page 1 of 1 — Displaying 1 - 5 of 5

You can also modify aggregation functions as follows:

```
object = connection | chart count(action=permitted) by source_address
```

In this query, only the log messages containing **action=permitted** are counted. You can write the same query as:

```
action = permitted object = connection | chart count() by source_address
```

Multiple counts or other aggregators can be used in a single query string.

```
object = connection | chart count(action=permitted), count(action=blocked)
by source_address
```

This query displays two columns. The first is the count of the connections with the permitted action and the second is the count of blocked actions.

timechart

You can use **timechart** to chart log messages as a time series data. It first displays logs according to the time they were collected or generated. Then, it returns the log results according to the collection time stamp (col_ts) or log generation time (log_ts).

The terms log_ts and col_ts have different functions.

log_ts	col_ts
Denotes the time present in log messages.	Denotes the time when SLS collected the log.

For example you can timechart all the messages with **login** shown below.

```
login | timechart count()
```

This plots the count of all the messages containing the word **login** into a graph with the horizontal axis as time. The total time-span is the time selected for the search query.

```
| timechart on log_ts count()
```

This query plots the count of the logs based on the **log_ts** field.

You can also use the **timechart** command to plot the data on a fixed time-interval. To have a **timechart** with bars for every 20 minutes, use the following query:

```
login | timechart count() every 20 minutes
```

You can use every x minutes, every x hours, or every x days with the **timechart**.

NOTE
When the limit of **timechart()** is not specified, the number of bars of the **timechart** depends on the nature of the query.

- The number is always equal to 30 if the time-range is less than 30 units. For example, if you provide a time range of 10 minutes SLS displays 30 bars in the span of 20 seconds.
- If the time-range is greater than 30 units, the number of bars is equal to the time-range. This holds true until the upper limit of the number of bars is reached, which is 59.
- There are also some special cases for the number of graphs. The number of bars is equal to the number of seconds specified and the time span of 1 day displays 24 bars in the span of one hour.

Available Aggregators

Aggregators are used with the **chart** and the **timechart** commands by joining them with the | symbol.

avg()

You can use **avg()** to calculate the average of all the values of the specified field.

Example:

```
| chart count(), avg(response_time, response_time=*)
```

This query calculates the average **response_time**.

count()

You can use **count** to get the total number of logs in the search results.

Example:

```
| chart count()
```

This query displays the total number of log messages in the search results.

```
login | chart count() by device_ip
```

This query searches for all the log messages containing the word **login**. It then groups the logs by their **device_ip**s and shows the count of the log messages for each of the **Device IP**.

You can also give filters to the **count()** as shown below.

```
login | chart count(event_id = 528) by device_ip
```

This query looks for all the log messages containing the word **login**. It then groups them by their **device_ip** s and shows the count of the messages containing the field value **event_id = 528**.

distinct_count()

You can use **distinct_count()** to get the number of distinct count of the object.

Example:

```
| chart distinct_count(destination_port) by destination_address
```

In this case, though different ports may have multiple counts, **distinct_count()** returns the count of the distinct ports for every destination address.

If the search results for a particular destination address had the following data:

port	count
21	20
25	30
901	15

The result for the **distinct_count()** is 3 for each of the ports 21, 25 and 901. However, the result of the **count()** is 65.

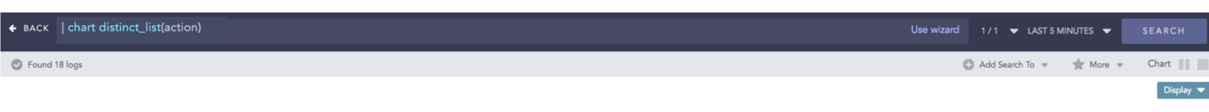
distinct_list()

You can use **distinct_list()** to return the list of all the distinct values of the field.

Example:

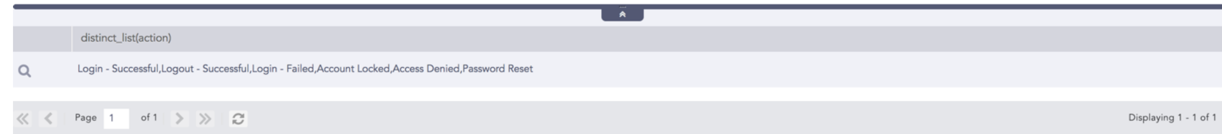
To view all the distinct values of the field **action** in the system, you can use the following query:

```
| chart distinct_list(action)
```



distinct_list(action)

Login - Successful,Logout - Successful,Login - Failed,Account Locked,Access Denied,Password Reset

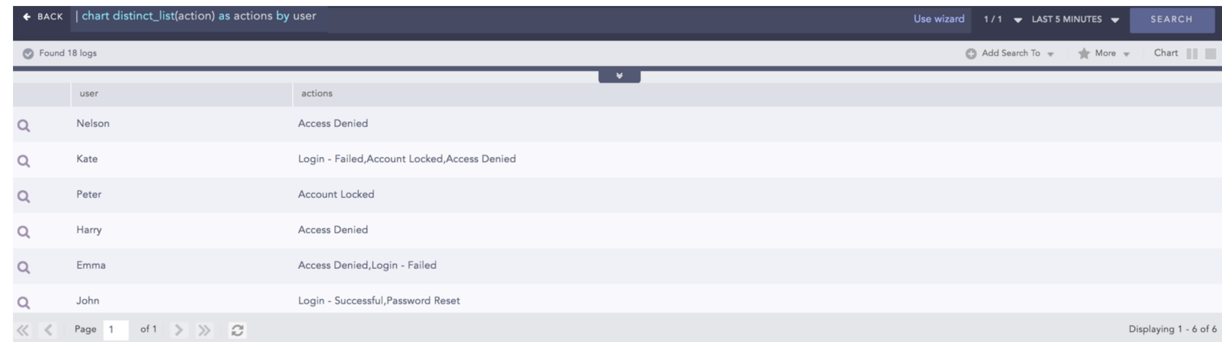


You can use a grouping parameter to group the distinct list.

Example:

```
| chart distinct_list(action) as actions by user
```

This query returns the list of every distinct value of the **action** field in the **actions** column grouped by the grouping parameter **user**. You can use this example to view all the actions performed and machines used by every user in your system.

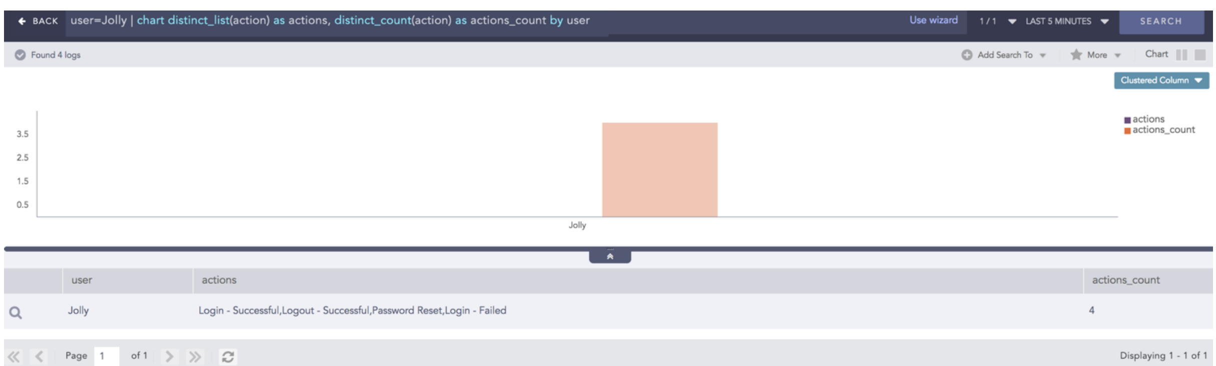


You can also use this aggregators with other aggregation commands.

Example:

```
user=Jolly | chart distinct_list(action) as actions, distinct_count
(action) as actions_count by user
```

This query returns the list of all the distinct actions with their counts for the user **Jolly**.



list()

list() takes a field as a parameter and returns the field values as a list in the search result. The duplicate field values are also included in the list.

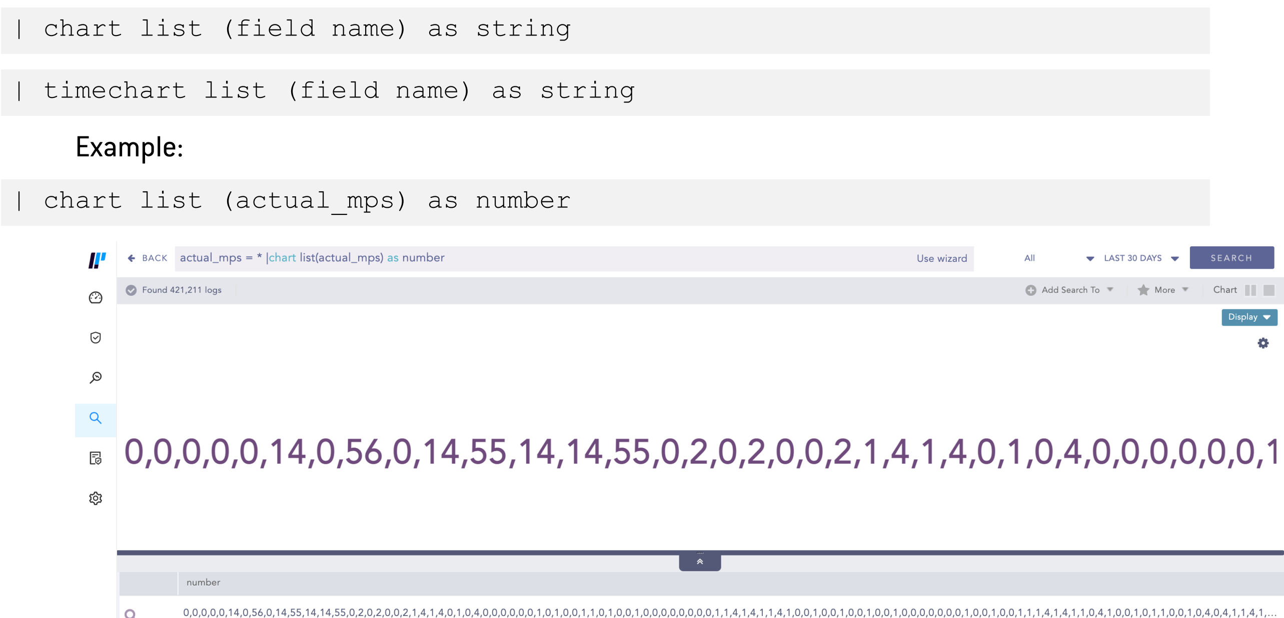
Syntax:

```
| chart list (field name) as string
```

```
| timechart list (field name) as string
```

Example:

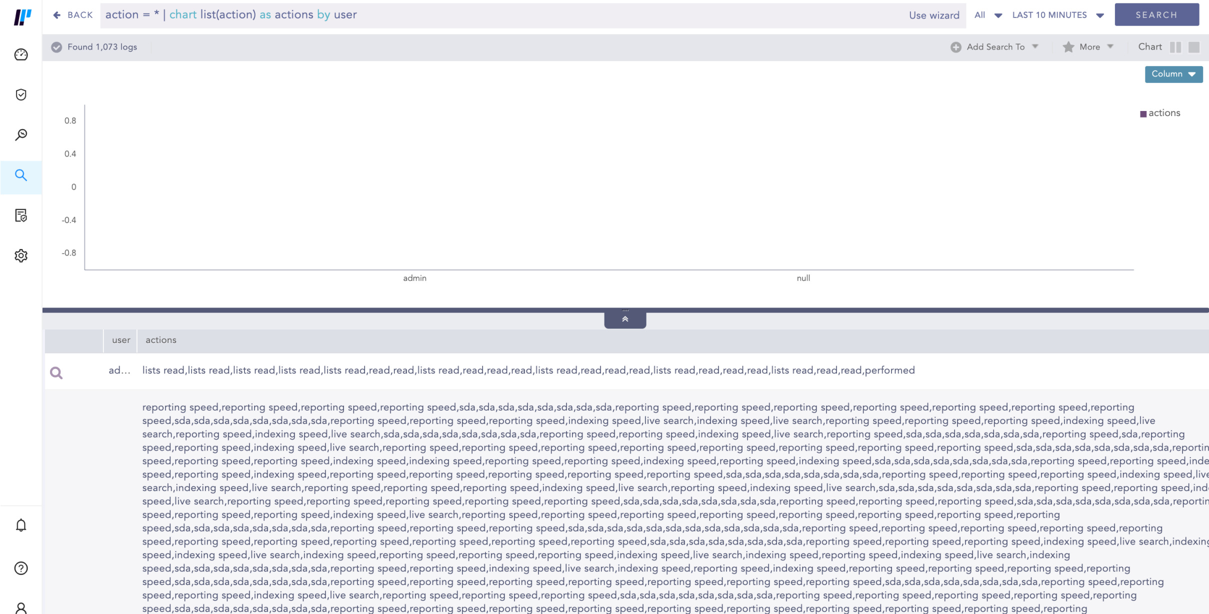
```
| chart list (actual_mps) as number
```



This query gives the list of the **actual_mps** field values and returns the list in the **number** field.

Example:

```
| chart list (action) as actions by user
```



This query gives the list of the **action** field values grouped by **user** grouping parameter and returns the list in the **actions** field.

max() and min()

These aggregators can be used to find the maximum or minimum value of the specified field.

Example:

```
| chart max(severity) by device_ip
```

This query displays the maximum **severity** value in each of the **device_ip**.

Example:

```
login | chart count(), max(col_ts) by device_ip, col_type
```

This query looks for all the log messages containing the word **login**. Then, it groups the search results by their **device_ip**s and the **col_type** and shows the count of the log messages and the latest **col_ts** for each of the groups.

The **max()** and **min()** also support filter expressions as:

```
| chart max(severity, severity < 5)
```

This query shows the maximum **severity** that is less than 5.

sum()

You can use the **sum()** to sum the values of the specified fields.

Example:

```
| chart sum(datasize) by device_ip
```

This query displays the sum of all the **datasize** fields for each **device_ip**.

You can also give filters to the **sum()** function.

Example:

```
| chart sum(datasize, datasize > 500)
```

This query only sums a datasize if it is greater than 500. The expression can be any valid query string but must not contain any view modifiers.

var()

You can use **var()** to calculate the variance of the field values. Variance describes how far the values are spread out from the mean value.

Execute the following query to visualize how the data fluctuates around the average value.

```
severity = * | chart count(),avg(severity),var(severity) by device_ip
```

NOTE
You can use **+**, **-**, *****, **/**, and **^** to add, subtract, multiply, divide, and to raise the power in the **min()**, **max()**, **sum()**, **avg()**, and **var()** functions.

Example:

```
avg(field1/field2^2+field3)
```

IMPORTANT
When using avg(), and min(), it is good to use a filter to discard log messages not containing the specified fields.

One-to-One Commands

The **One-to-one** commands take one value as input and provide one output.

For example, you can use the **rex** and the **norm** commands to extract specific parts of the log messages into an ad-hoc field name. This is equivalent to normalizing log messages during the search. However, the extracted values are not saved.

The **rex** and **norm** commands do not filter the log messages. They list all the log messages returned by the query and add the specified ad-hoc key-value pairs if possible.

> **IMPORTANT**
> Using the **rex** and **norm** commands or the **msg** field on large volume of logs may severely impact system performance. If a field you are processing already contains the required information and only needs further processing, we recommend you use **norm on** or **rex on** instead.

rex

You can use the **rex** command to recognize regex patterns in the **re2** format. The extracted variable is retained only for the current search scope. The result also shows the log messages that are not matched by the rex expression.

Example Log:

```
Oct 15 20:33:02 WIN-J2OVISWBB31.immuneaps.nfsserver.com MSWinEventLog 1
Security 169978 Sat Oct 15 20:33:01 2011 5156 Microsoft-Windows-Security-
Auditing N/A N/A Success Audit WIN-J2OVISWBB31.immuneaps.nfsserver.com
Filtering Platform Connection The Windows Filtering Platform has allowed a
connection. Application Information: Process ID: 4 Application Name:
System Network Information: Direction: Inbound Source Address:
192.168.2.255 Source Port: 138 Destination Address: 192.168.2.221
Destination Port: 138 Protocol: 17 Filter Information: Filter Run-Time ID:
67524 Layer Name: Receive/Accept Layer Run-Time ID: 44 169765
```

You can use the **rex** command to extract the protocol id into a field protocol_id with the following syntax:-

```
| rex Protocol:\s*(?P<protocol_id>\d+)
```

The query format is similar to the following:

```
| rex any regular expression:\s+(?P<field_name>expression to capture to
field)
```

> **IMPORTANT**
> The (?P< >) expression is part of the rex syntax to specify the field name.

You can also extract multiple fields from a single rex operation as shown below.

```
| rex Source Address:\s*(?P<src_address>\d+\.\d+\.\d+\.\d+)
```

The extracted values can be used to chart your results. For example,

```
| rex Protocol:\s+(?P<protocol_id>\d+) | chart count() by protocol_id
```

Since the rex command acts on the search results, you can add it to a query string as shown below:

```
Windows Filtering AND allowed | rex Protocol:\s+(?P<protocol_id>\d+)
```

```
user=* | rex on user:\s+(?P<account>\S+)@(?P<domain>\S+) | chart count()
by account, domain | search account=*
```

> **NOTE**
>
> Use Single quote to address inline normalization while using square bracket. For example:
>
> - This syntax works: | norm on user <my_user:\S+> | chart count() by my_user.
> - But this does not. | norm on user <my_user:[A-Z]+> | chart count() by my_user.
>
> If you use the box brackets (**[,]**), single quote (") is necessary in the syntax.

norm

You can use the **norm** command to extract variables from the search results into a field. The difference between the **rex** command and the **norm** command is that **norm** supports both normalization syntax and re2 syntax. The **rex** command only supports re2 syntax.

Example Log:

```
Dec 17 05:00:14 ubuntu sshd[7596]: Invalid user Bob from 110.44.116.194
```

To extract the value of the user into the field **user**, use the following syntax:-

```
| norm Invalid user <user:word>
```

And this can also be used to chart in the graph as follows.

```
| norm Invalid user <user:word>| chart count() by user
```

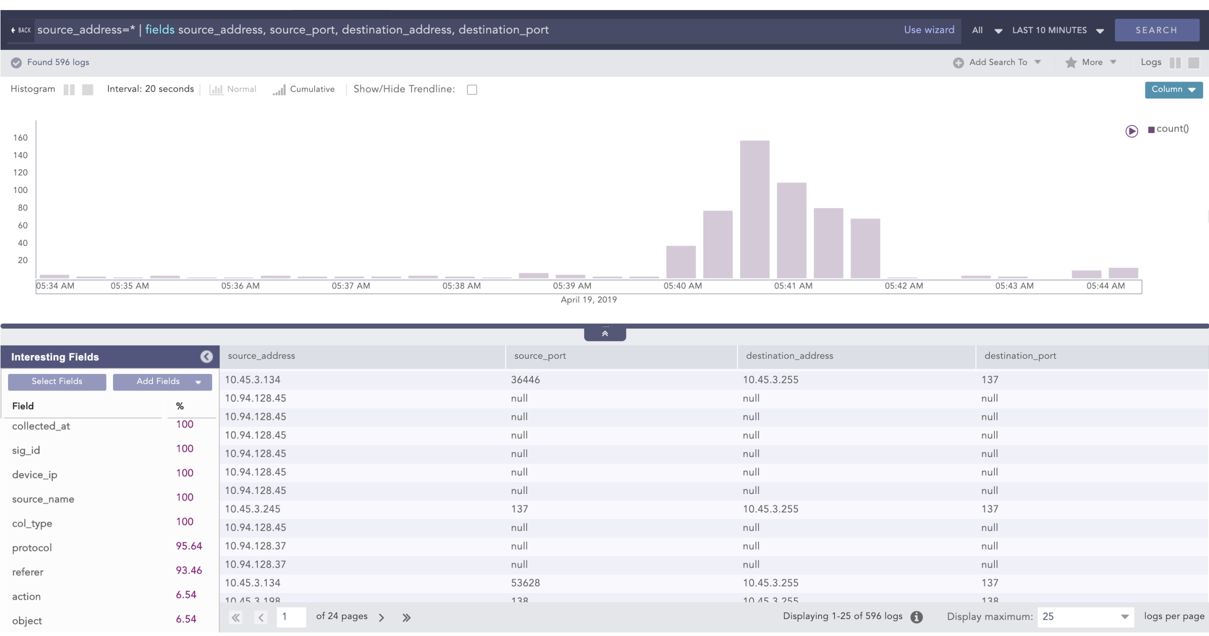
You can also use the **norm** command to extract multiple key-value pairs as shown below:

```
| norm Invalid user <user:word> from <source_ip:ip>
| chart count() by my_user, msg | search my_user=*
```

> **NOTE**
>
> Use Single quote to address inline normalization while using square bracket. For example:
>
> - This syntax works: | norm on user <my_user:\S+> | chart count() by my_user.
> - But this does not. | norm on user <my_user:[A-Z]+> | chart count() by my_user.
>
> If you use the box brackets (**[,]**), single quote (") is necessary in the syntax.

fields

You can use the **fields** command to display the search results in a tabular form. The table is constructed with headers according to the field-names you specify. SLS returns **null** if the logs do not contain the specified fields.




```
| fields source_address, source_port, destination_address, destination_
port
```

Here, the fields **source_address**, **source_port**, **destination_address**, and **destination_port** are displayed in a tabular form as shown above.

Any log message without the field **destination_port** has a corresponding row with the **destination_port** column value as -N/A-.

rename

You can use the **rename** command to rename the original field names.

Example:

```
| rename device_ip as host
```

When multiple fields of a log are renamed as the same name, the rightmost field takes precedence over others and only that field is renamed.

Example:

```
| rename source_address as ip, destination_address as ip
```

Here, if both the **source_address** and **destination_address** fields are present in a log, only the **destination_address** field is renamed as **ip** in search results.

The log messages after normalization can have different field-names for information carrying similar values. For example, different logs may have **name**, **username**, **u_name**, or **user_name** as keys for the same field username. To aggregate all the results and analyze them properly, you can use the rename command.

```
| rename target_user as user, caller_user as user | chart count() by user
```

In some cases, the field names can be more informative with the use of rename command as below:

```
label = Attack | rename source_address as attacking_ip | chart count() by
attacking_ip
```

Process Commands

You can use the **process** command to execute different one-to-one functions which produce one output for one input given.

SLS Process Commands are:

AsciiConverter

Converts hexadecimal (hex) value and decimal (dec) value of various keys to their corresponding readable ASCII values. It supports the Extended ASCII Table for processing decimal values.

Hexadecimal to ASCII

Syntax:

```
| process ascii_converter(fieldname,hex) as string
```

Example:

```
| process ascii_converter(sig_id,hex) as alias_name
```

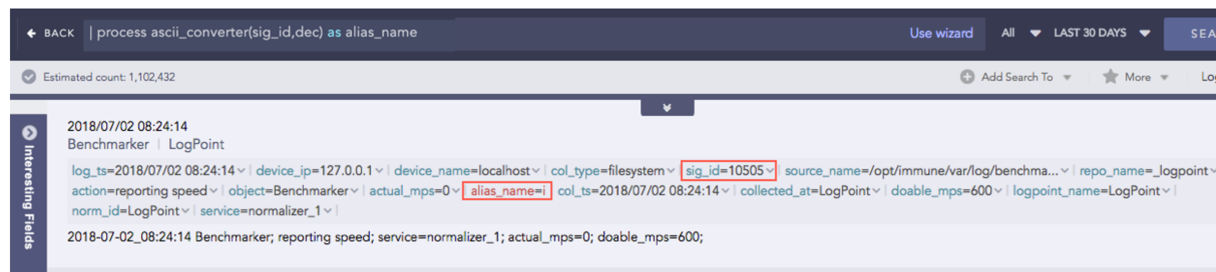
Decimal to ASCII

Syntax:

```
| process ascii_converter(fieldname,dec) as string
```

Example:

```
| process ascii_converter(sig_id,dec) as alias_name
```



Clean Char

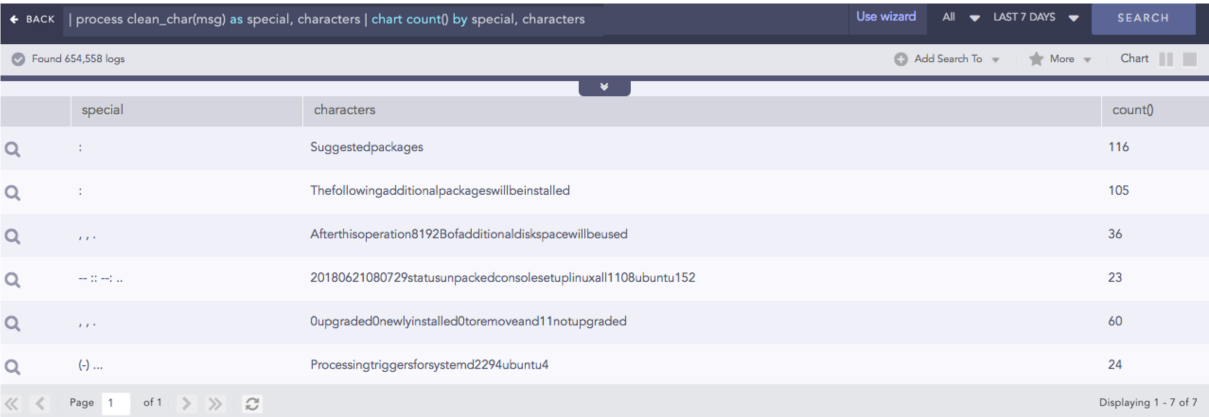
Removes all the alphanumeric characters present in a field-value.

Syntax:

```
| process clean_char(<field_name>) as <string_1>, <string_2>
```

Example:

```
| process clean_char(msg) as special, characters
| chart count() by special, characters
```



Codec

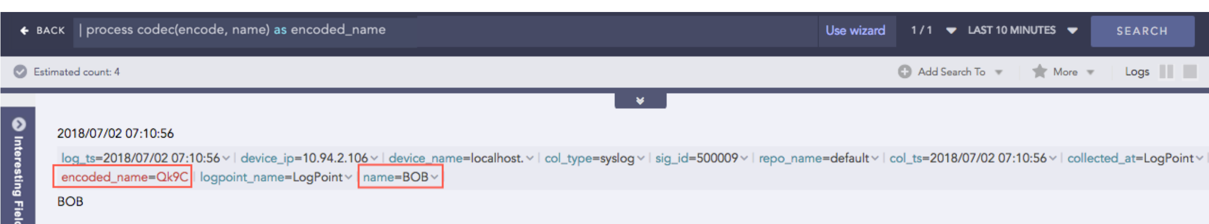
Codec is a compression technology with an encoder to compress the files and a decoder to decompress. This process command encodes the field values to ASCII characters or decodes the ASCII characters to their text value using the Base64 encoding/decoding method. Base64 encoding converts binary data into text format so a user can securely handle it over a communication channel.

Syntax:

```
| process codec(<encode/decode function>, <field to be encoded/decoded>)
as <attribute_name>
```

Example:

```
| process codec(encode, name) as encoded_name
```



Here, the "| process codec(encode, name) as encoded_name" query encodes the value of **name** field by applying **encode** function and displays encoded value in **encoded_name**.

Compare

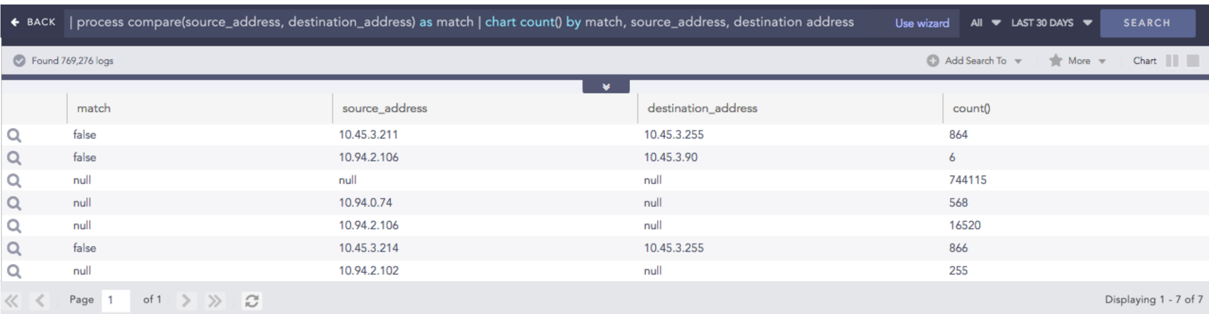
Compares two values to check if they match or not.

Syntax:

```
| process compare(fieldname1,fieldname2) as string
```

Example:

```
| process compare(source_address, destination_address) as match
| chart count() by match, source_address, destination address
```



match	source_address	destination_address	count()
false	10.45.3.211	10.45.3.255	864
false	10.94.2.106	10.45.3.90	6
null	null	null	744115
null	10.94.0.74	null	568
null	10.94.2.106	null	16520
false	10.45.3.214	10.45.3.255	866
null	10.94.2.102	null	255

Compare Network

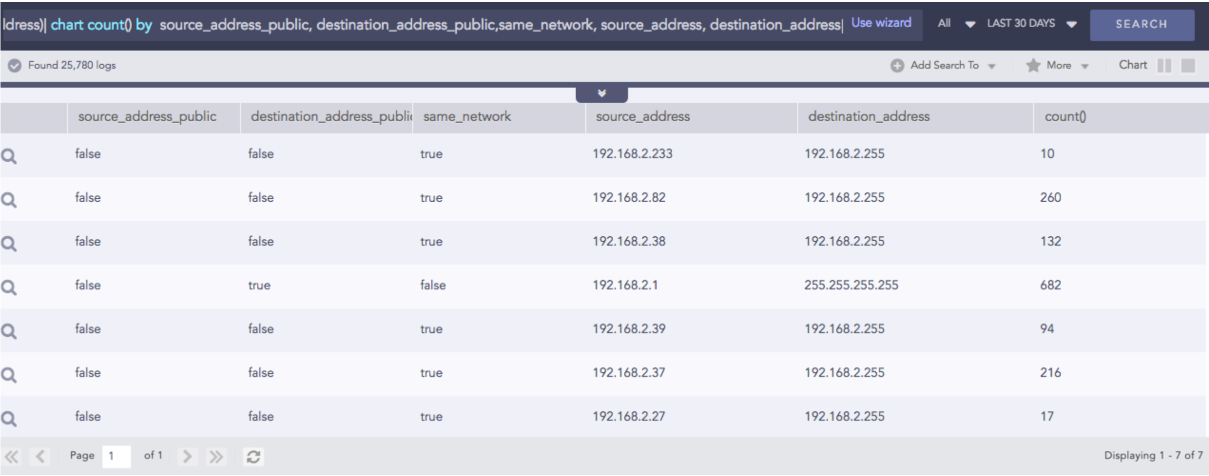
Takes a list of **IP addresses** as inputs and checks if they are from the same network or different ones. It also checks whether the networks are public or private. The comparison is carried out using either the default or the customized CIDR values.

Syntax:

```
| process compare_network(fieldname1,fieldname2)
```

Example: (Using default CIDR value)

```
source_address=* destination_address=*
| process compare_network (source_address, destination_address)
| chart count() by source_address_public, destination_address_public,
same_network, source_address, destination_address
```



source_address_public	destination_address_publi	same_network	source_address	destination_address	count()
false	false	true	192.168.2.233	192.168.2.255	10
false	false	true	192.168.2.82	192.168.2.255	260
false	false	true	192.168.2.38	192.168.2.255	132
false	true	false	192.168.2.1	255.255.255.255	682
false	false	true	192.168.2.39	192.168.2.255	94
false	false	true	192.168.2.37	192.168.2.255	216
false	false	true	192.168.2.27	192.168.2.255	17

Count Char

Counts the number of characters present in a field-value.

Syntax:

```
| process count_char(fieldname) as int
```

Example:

```
| process count_char(msg) as total_chars
| search total_chars >= 100
```

CountOf

Takes a field as a parameter and counts the number of times the element(s) occurred in the field's value.

Syntax:

```
| process count_of (source field name, string, kind)
```

Here, the **source** and **search** parameters are required.

Example:

```
| process count_of (device_address, "127") as cnt
```

This query counts the occurance of **127** string in the value of **device_address** field and displays it in **cnt**.

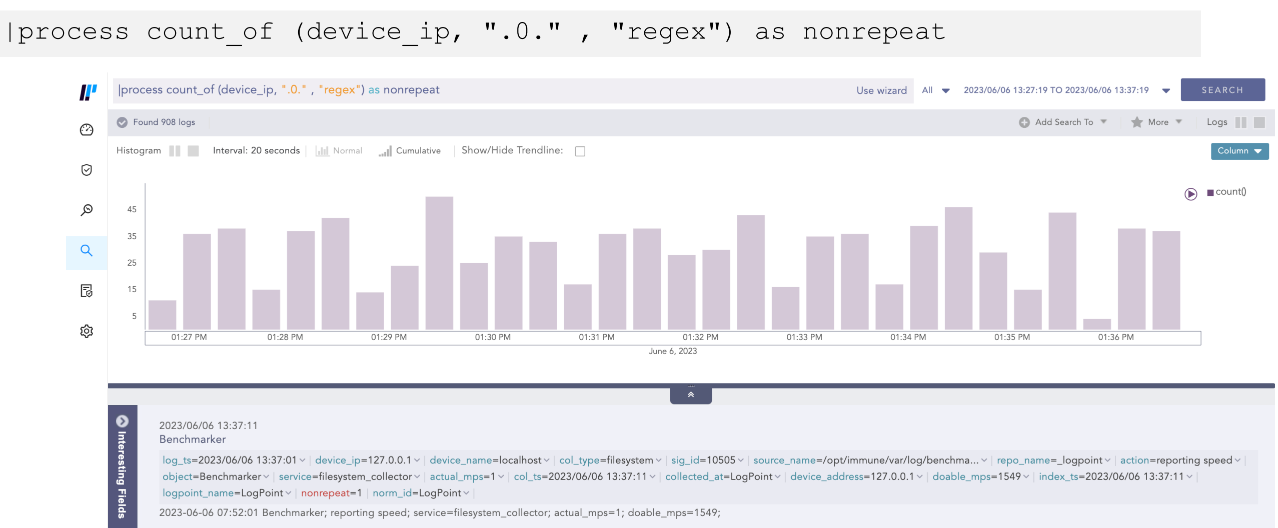
Example:

```
| process count_of (collected_at, "L") as occur
```

This query counts the occurance of **L** string in the value of **collected_at** field and displays it in **occur**.

Example:

```
|process count_of (device_ip, ".0." , "regex") as nonrepeat
```



This query counts the occcurance of **.0.** string by applying regex pattern in the value of **device_ip** field and displays it in **nonrepeat**.

Current Time

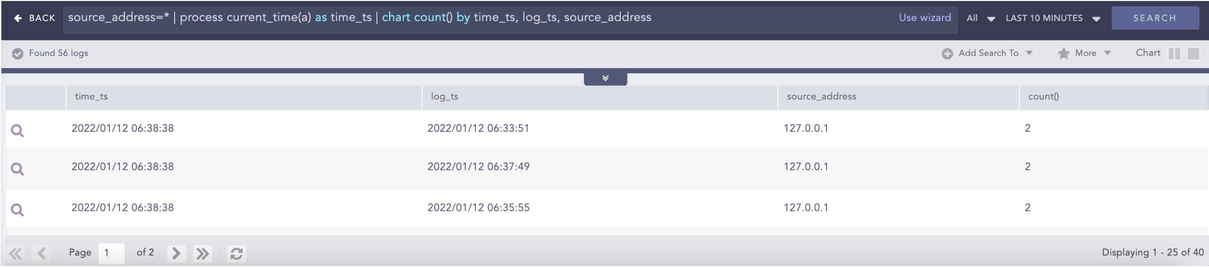
Gets the current time from the user and adds it as a new field to all the logs. This information can be used to compare, compute, and operate the timestamp fields in the log message.

Syntax:

```
| process current_time(a) as string
```

Example:

```
source_address=* | process current_time(a) as time_ts
| chart count() by time_ts, log_ts, source_address
```



DatetimeDiff

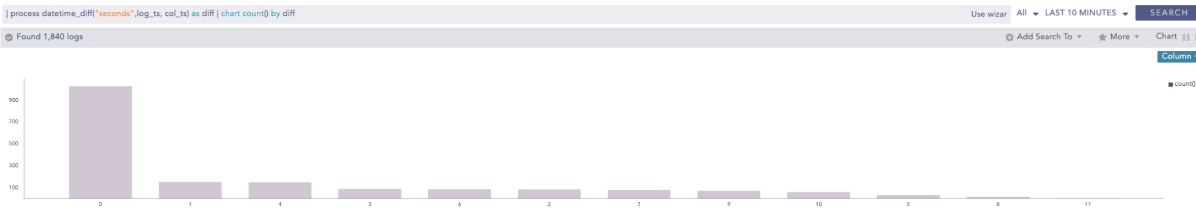
Processes two lists, calculates the difference between them, and returns the absolute value of the difference as the delta. The two lists must contain timestamps. It requires two first and second input parameters that are mandatory and can either be a list or a single field. The third parameter is mandatory and represents the required difference between the two input fields. This difference must be specified in either seconds, minutes or hours. The purpose of the third parameter is to determine how the difference between the two input fields can be represented. For instance, if the difference is specified in seconds, the output will show the absolute difference in seconds.

Syntax:

```
| process datetime_diff("seconds", ts_list1, ts_list2) as delta
```

Example:

```
| process datetime_diff("seconds",log_ts, col_ts) as diff | chart count()
by diff
```



Difference

Calculates the difference between two numerical field values of a search.

Syntax:

```
| process diff(fieldname1,fieldname2) as string
```

Example:

```
| process diff(sent_datasize,received_datasize) as difference
| chart count() by sent_datasize, received_datasize,difference
```

sent_datasize	received_datasize	difference	count()
403,582	394,542	9,040	1
403,567	394,528	9,039	1
null	null	null	1321
403,574	394,535	9,039	1
403,558	394,519	9,039	1
403,551	394,511	9,040	1

DNS Cleanup

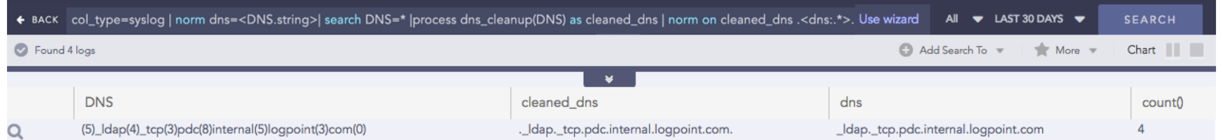
Converts a DNS from an unreadable format to a readable format.

Syntax:

```
| process dns_cleanup(fieldname) as string
```

Example:

```
col_type=syslog | norm dns=<DNS.string>| search DNS=*
|process dns_cleanup(DNS) as cleaned_dns
| norm on cleaned_dns .<dns:.*>.
| chart count() by DNS, cleaned_dns, dns
```



DNS	cleaned_dns	dns	count()
(5)_ldap(4)_tcp(3)pdc(8)internal(5)logpoint(3)com(0)	._ldap._tcp.pdc.internal.logpoint.com.	_ldap._tcp.pdc.internal.logpoint.com	4

DNS Process

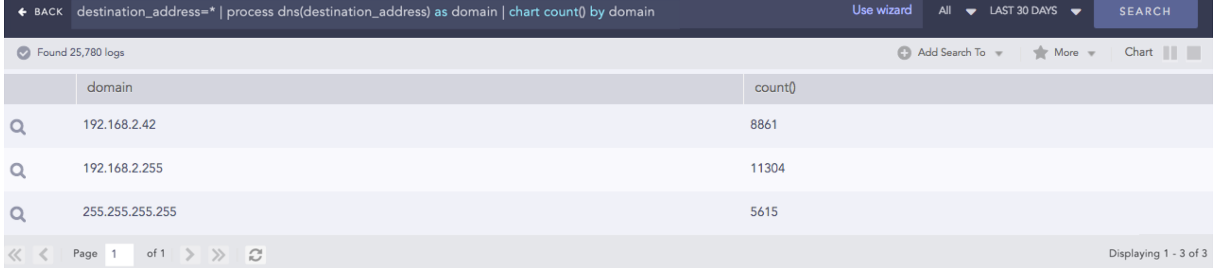
Returns the domain name assigned to an IP address and vice-versa. It takes an **IP address** or a **Domain Name** and a **Field Name** as input. The plugin then verifies the value of the field. If the input is an **IP Address**, it resolves the address to a **hostname** and if the input is a **Domain Name**, it resolves the address to an **IP Address**. The output value is stored in the **Field Name** provided.

Syntax:

```
| process dns(IP Address or Hostname)
```

Example:

```
destination_address=* | process dns(destination_address) as domain
| chart count() by domain
```



domain	count()
192.168.2.42	8861
192.168.2.255	11304
255.255.255.255	5615

Domain Lookup

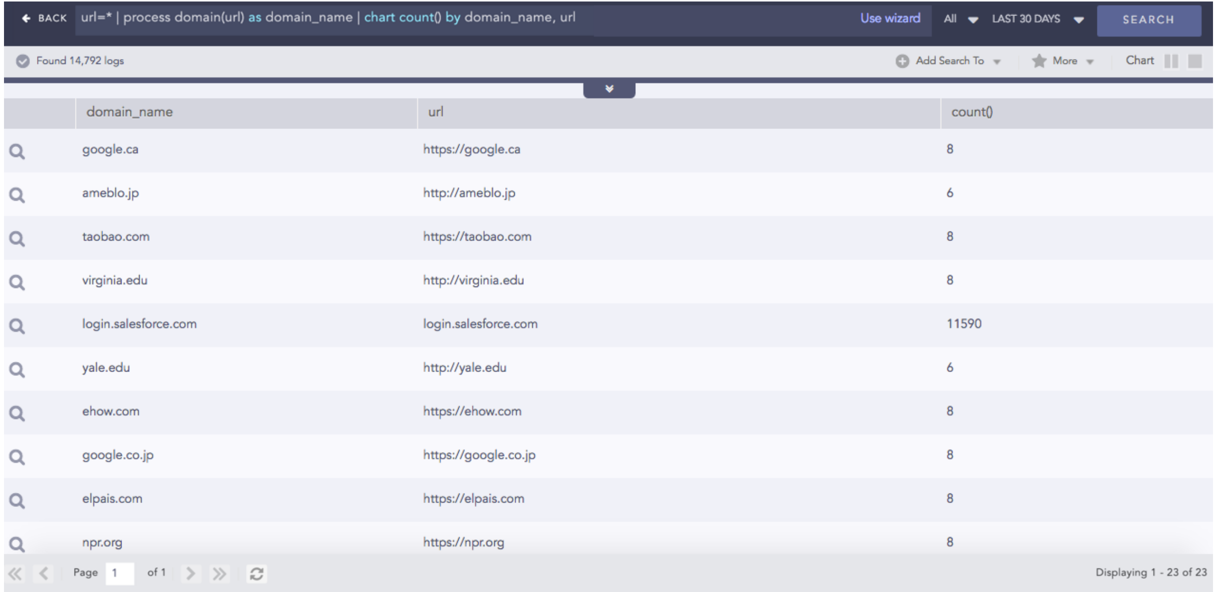
Provides the domain name from a URL.

Syntax:

```
| process domain(url) as domain_name
```

Example:

```
url=* | process domain(url) as domain_name |
chart count() by domain_name, url
```



domain_name	url	count()
google.ca	https://google.ca	8
ameblo.jp	http://ameblo.jp	6
taobao.com	https://taobao.com	8
virginia.edu	http://virginia.edu	8
login.salesforce.com	login.salesforce.com	11590
yale.edu	http://yale.edu	6
ehow.com	https://ehow.com	8
google.co.jp	https://google.co.jp	8
elpais.com	https://elpais.com	8
npr.org	https://npr.org	8

Entropy

Entropy measures the degree of randomness in a set of data. This process command calculates the entropy of a field using the Shanon entropy formula and displays data in the provided field. A higher entropy number denotes a data set with more randomness, which increases the probability that a system artificially generated the values and could potentially lead to a malicious conclusion.

Syntax:

```
| process entropy (field) as field_entropy
```

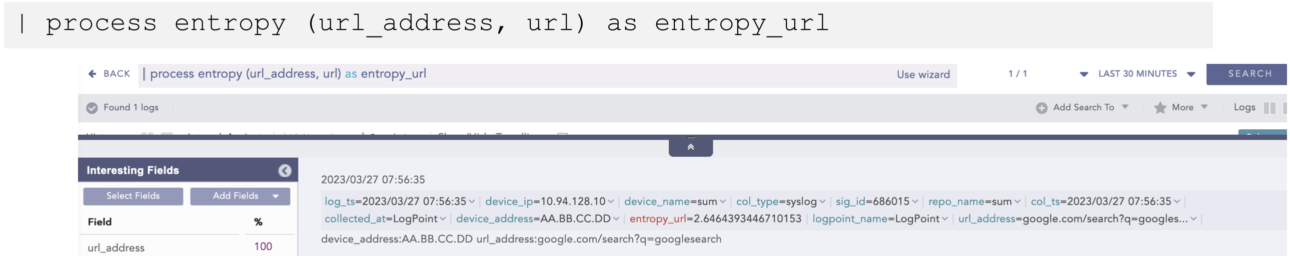
Example:

```
device_address = *| process entropy (device_address) as test
```

Here, the "| process entropy (device_address) as test" command calculates the entropy of the **device_address** field and displays it in **test**.

Example:

```
| process entropy (url_address, url) as entropy_url
```



Here, the "| process entropy (url_address, url) as entropy_url" command takes **url** as an optional parameter and extracts the domain name from the **url_address** to perform entropy calculation on it and displays it in **entropy_url**.

Example:

```
| process entropy ("google.com", string) as en
```

Here, the "| process entropy ("google.com", string) as en" command takes **string** as an optional parameter and calculates the entropy of **google.com** raw string field and displays it in **en**.

Eval

Evaluates mathematical, boolean and string expressions. It places the result of the evaluation in an identifier as a new field.

Syntax:

```
| process eval("identifier=expression")
```

Example:

```
| process eval("Revenue=unit_sold*Selling_price")
```

Experimental Median Quartile Quantile

Performs statistical analysis (median, quartile and quantile) of events based on fields. All these commands take numerical field values as input.

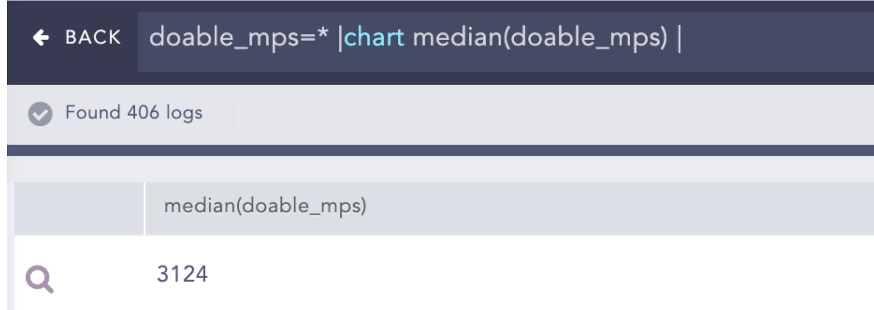
Median

Syntax:

```
| chart median(fieldname) as string
```

Example:

```
doable_mps=* |chart median(doable_mps)
```



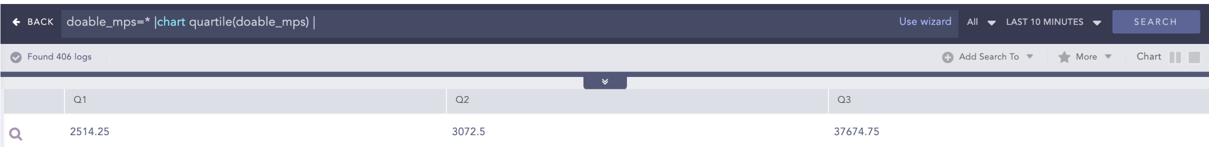
Quartile

Syntax:

```
| chart quartile(fieldname) as string1, string2, string3
```

Example:

```
doable_mps=* |chart quartile(doable_mps)
```



Quantile

Syntax:

```
| process quantile(fieldname)
```

Example:

```
doable_mps=* | process quantile(doable_mps)
|search quantile>0.99
|chart count() by doable_mps order by doable_mps desc
```

GEOIP

Gives the geographical information of a public IP address. It adds a new value "internal" to all the fields generated for the private IP supporting the RFC 1918 Address Allocation for Private Internets.

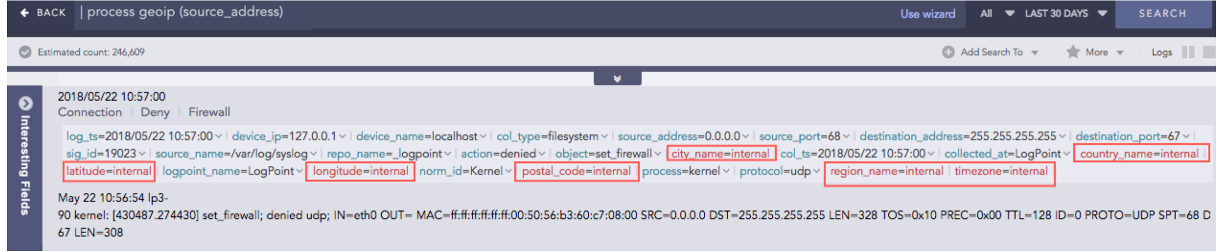
Syntax:

```
| process geoip (fieldname)
```

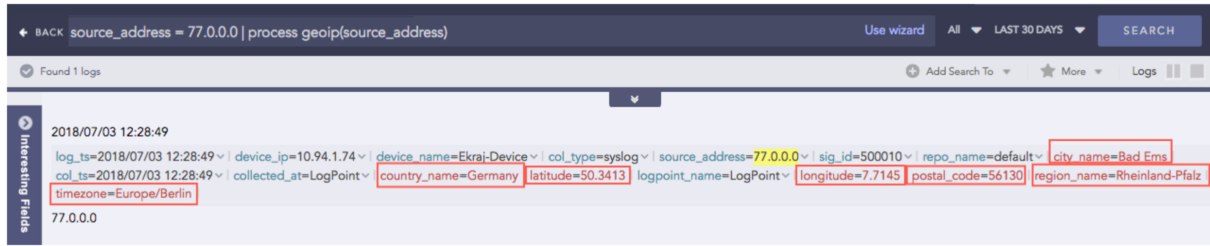
Example:

```
| process geoip (source_address)
```

For the Private IP:



For the Public IP:



Grok

Extracts key-value pairs from logs during query runtime using Grok patterns. Grok patterns are the patterns defined using regular expression that match with words, numbers, IP addresses, and other data formats.

Refer to **Grok Patterns** and find a list of all the Grok patterns and their corresponding regular expressions.

Syntax:

```
| process grok("<signature>")
```

A **signature** can contain one or more Grok patterns.

Example:

To extract the IP address, method, and URL from the log message:

```
192.168.3.10 GET /index.html
```

Use the command:

```
| process grok("%{IP:ip_address_in_log} %{WORD:method_in_log} %
{URIPATHPARAM:url_in_log}")
```

Using this command adds the **ip_address_in_log**, **method_in_log**, and **url_in_log** fields and their respective values to the log if it matches the signature pattern.

InRange

Determines whether a certain field-value falls within the range of two given values. The processed query returns TRUE if the value is in the range.

Syntax:

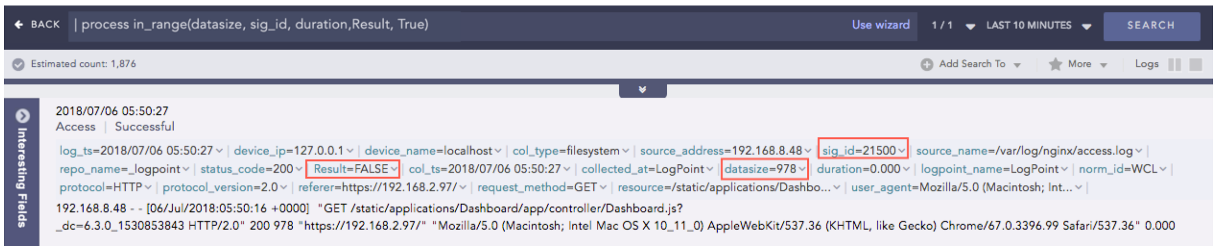
```
| process in_range(endpoint1, endpoint2, field, result, inclusion)

where,

endpoint1 and endpoint2 are the endpoint fields for the range, the field
is the fieldname to check whether its value falls within the given range,
result is the user provided field to assign the result (TRUE or FALSE),
inclusion is the parameter to specify whether the range is inclusive or
exclusive of given endpoint values. When this parameter is TRUE, the
endpoints will be included for the query and if it is FALSE, the endpoints
will be excluded.
```

Example:

```
| process in_range(datasize, sig_id, duration,Result, True)
```



IP Lookup

Enriches the log messages with the Classless Inter-Domain Routing (CIDR) address details. A list of CIDRs is uploaded in the CSV format during the configuration of the plugin. For any **IP Address** type within the log messages, it matches the IP with the content of the user-defined Lookup table and then enriches the search results by adding the CIDR details.

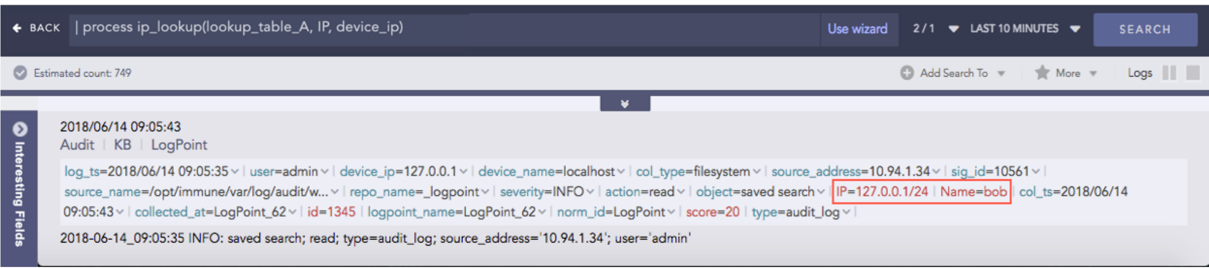
Syntax:

```
| process ip_lookup(IP_lookup_table, column, fieldname)
where IP_lookup_table is the lookup table configured in the plugin, Column
is the column name of the table which is to be matched with the fieldname
of the log message.
```

Example:

```
| process ip_lookup(lookup_table_A, IP, device_ip)
```

This command compares the IP column of the lookup_table_A with the device_ip field of the log and if matched, the search result is enriched.



JQ Parser

Applies the JQ filter to the fields with valid JSON field values of normalized logs and extracts key values from that field. The JQ filter defines a path for extracting the required data from a JSON file and has a wide variation and functionality.

Syntax:

```
| process jq_parser (field name, "filter") as field name
```

Example:

```
| process jq_parser (conditional_access_policies, ".[].result") as cap_
result
```

Here, the "| process jq_parser (conditional_access_policies, ".[].result") as cap_result" query applies **[]** (array filter) and **result** filter to the **conditional_access_policies** field and extracts the key values to the **cap_result** field.

JSON Expand

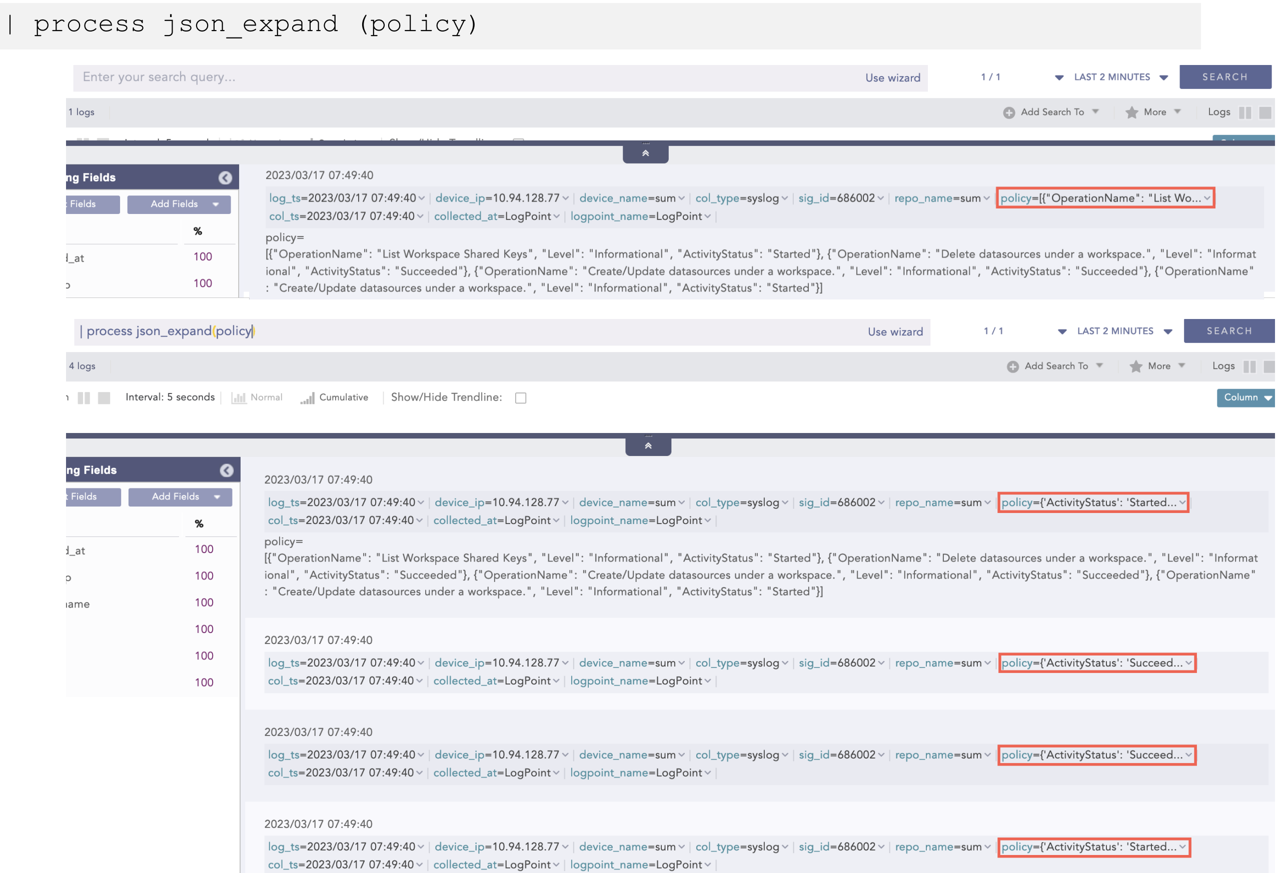
Takes the field with a valid JSON array value and creates separate log instances for individual array items of that field. Each array item takes the original field name.

Syntax:

```
| process json_expand (field name)
```

Example:

```
| process json_expand (policy)
```



Here, the "| process json_expand (policy)" query expands the **policy** field into four log instances. After expansion, each array item takes the **policy** as a field name.

JSON Parser

The JavaScript Object Notation (JSON) Parser reads JSON data and extracts key values from the fields with valid JSON field values of normalized logs. A string filter is applied to the provided field, which defines a path for extracting values from it. The filter contains a key, which can be alphanumeric and special characters except square brackets ([]), backtick (`) and tilde (~). These exceptional characters are reserved for essential use cases, such as mapping the list and selecting a condition in JSON Parser.

The supported filter formats for JSON Parser are:

- Chaining for nested JSON
  Example: .fields.user.Username
- Array access
  Example: .[1]

Syntax:

```
| process json_parser (field name, "filter") as field name
```

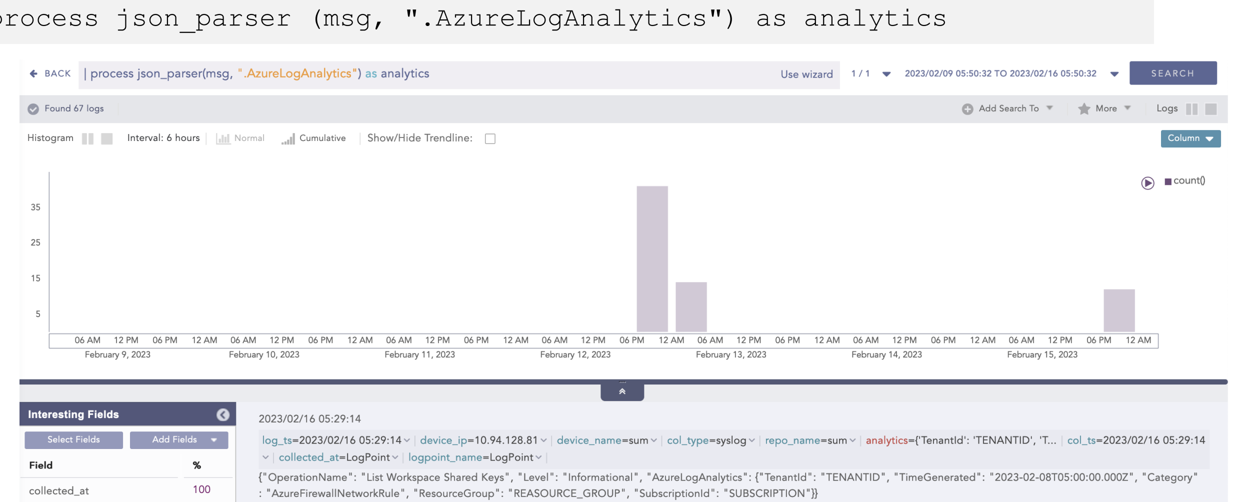
JSON Parser supports **map** and **select** functions for applying filters with true conditional statements. The supported conditional operators are: =, !=, >, < , >= and <=.

General syntax to use map and select functions:

```
| process json_parser(field name, ".[condition]") as field name
```

Example:

```
| process json_parser (msg, ".AzureLogAnalytics") as analytics
```



Here, the "| process json parser (msg, ".AzureLogAnalytics") as analytics" query applies the **AzureLogAnalytics** filter to the **msg** field and extracts the key values to the **analytics** field.

Example:

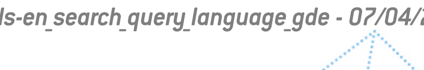
```
evidence = {
      "@odata.type": "#microsoft.graph.security.mailboxEvidence",
      "userAccount": {
        "account Name": "john.cena",
        "#userPrincipalName": "john.cena@lpbank.com",
      }
    }
                                                  Filter

Query: | process json_parser(evidence, ".@data\.type") as data_type
       | process json_parser(evidence, ".userAccount.display name") as display_name
       | process json_parser(evidence, ".userAccount.#userPrincipalName") as upn


Result:
    data_type = #microsoft.graph.security.mailboxEvidence
    display_name = john.cena
    upn = john.cena@lpbank.com
```

In filter, the backslash escaped the period before **type** and query applies the filter to the **evidence** field and extracts the key value to the **data type** field.

Example:



```
detail = [
{
"attachment": "CXmail",
"severity": 60
},
{
"attachment": "MalPE",
"severity": 80
},
{
"attachment": "Inject-IIJ",
"severity": 20
}
]
| process json_parser(detail, ".[.severity > 50]") as listWithSeverityGreaterThan50

Result:
listWithSeverityGreaterThan50 = [
{
"attachment": "CXmail",
"severity": 60
},
{
"attachment": "MalPE",
"severity": 80
}
]
```

In the **.[.severity>50]** filter, a conditional statement **severity>50** is used and the "| process json_parser(detail, ".[.severity > 50]") as listWithSeverityGreaterThan50" query applies the filter to the **detail** field and extracts the list of key values with the true condition to the **listWithSeverityGreaterThan50** field.

ListLength

Returns the number of elements in the list.

Syntax:

```
| process list_length(list) as length
```

Example:

```
| chart distinct_list(actual_mps) as lst | process list_length(lst) as
lst_length
```

| chart distinct_list(actual_mps) as lst | process list_length(lst) as lst_length

lst	lst_length
1,0,2,4,3,12,7,6	8

ListPercentile

Calculates the percentile value of a given list. It requires at least two input parameters. The first parameter is mandatory and must be a list. This command can also accept up to five additional parameters. The second parameter must be an alias, which is used in conjunction with the percentile percentage to determine the required percentile. The alias is concatenated with the percentile percentage to store the required percentile value.

Syntax:

```
| process list_percentile(list, 25, 75, 95, 99) as x

Result: x_25th_percentile = respective_value
 x_75th_percentile = respective_value
 x_95th_percentile = respective_value
 x_99th_percentile = respective_value

General:
 | process list_percentile(list,p) as aliasalias_pth_percentile
```

Example:

```
| actual_mps=* chart distinct_list(actual mps) as a | process list_
percentile(a, 50, 95,99) as x | chart count() by a, x_50th_percentile, x_
95th_percentile, x_99th_percentile
```

actual_mps=* | chart distinct_list(actual_mps) as a | process list_percentile(a, 50, 95, 99) as x | chart count() by a , x_50th_percentile, x_95th_percentile, x_99th_percentile — Use wizard

a	x_50th_percentile	x_95th_percentile	x_99th_percentile
[2, 0, 3, 1, 4, 7, 5]	3	6.399999999999999	6.879999999999999

Next

Takes a list and an offset as input parameters and returns a new list where the elements of the original list are shifted to the left by the specified offset. The maximum allowable value for the offset is 1024. For example, if the original list is [1, 2, 3, 4, 5, 6] and the offset is 1, the resulting list would be [2, 3, 4, 5, 6]. Similarly, if the offset is 2, the resulting list would be [3, 4, 5, 6]. This command requires two parameters as input. The first is mandatory and must be a list. The second parameter is mandatory and represents the offset value. An alias of 1 must be provided as input.

Syntax:

```
| process next(list, 1) as next_list| process next(list, 2) as next_list_2
```

Example:

```
| chart list(user) as list | process next(list, 1) as next_list | chart
count() by list next_list
```

| chart list(user) as list
| process next(list, 1) as next_list
| chart count() by list, next_list — Use wizard

list	next_list	count()
[1, 2, 3]	[2, 3]	1

Percentile

Percentiles are numbers below which a portion of data is found. This process command calculates the statistical percentile from the provided field and informs whether the field's value is high, medium or low compared to the rest of the data set.

Syntax:

```
| chart percentile (field name, percentage)
```

Example:

```
doable_mps = *| chart percentile (doable_mps, 99)
```

percentile(doable_mps, 99)
77,366.6

Here, the "| chart percentile (doable_mps, 99)" command calculates the percentile for the value of the **doable_mps** field.

Process lookup

This process command looks up related data from a user defined table.

Syntax:

```
| process lookup(table,field)
```

Example:

```
| process lookup(lookup_table, device_ip)
```

Regex

Extracts specific parts of the log messages into custom field names.

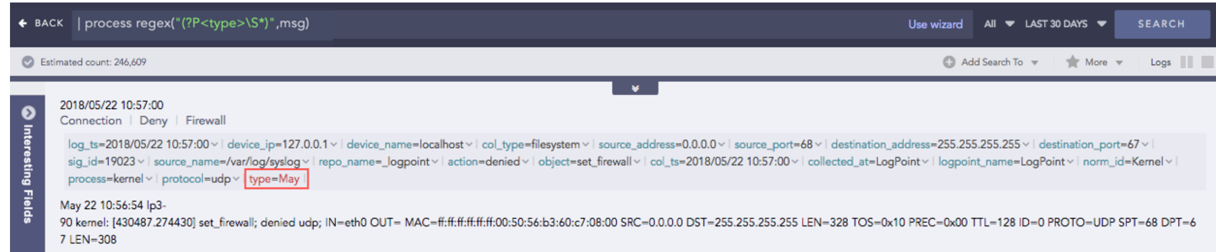
Syntax:

```
| process regex("_regexpattern", _fieldname)
| process regex("_regexpattern", "_fieldname")

Both syntaxes are valid.
```

Example:

```
| process regex("(?P<type>\S*)",msg)
```



SortList

Sorts a list in ascending or descending order. By default, the command sorts a list in ascending order. The first parameter is mandatory and must be a list. The second parameter desc is optional.

Syntax:

```
| process sort_list(list) as sorted_list
| process sort_list(list, "desc") as sorted_list
```

Example:

```
chart distinct list(actual_mps) as lst | process sort_list(lst) as LP_KB_
Dynamictable_Populate_Values | chart count by lst, sorted list
```

| chart distinct_list(actual_mps) as lst | process sort_list(lst) as sorted_list | chart count() by lst, sorted_list

lst	sorted_list
[2, 0, 1, 3, 4, 12, 7, 6]	[0, 1, 2, 3, 4, 6, 7, 12]

String Concat

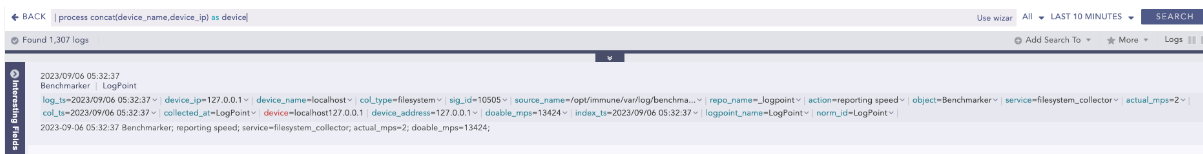
Joins multiple field values of the search results.

Syntax:

```
| process concat(fieldname1, fieldname2, ...., fieldnameN) as string
```

Example:

```
| process concat(device_name,device_ip) as device
```



Summation

Calculates the sum between two numerical field values of a search.

Syntax:

```
| chart sum(fieldname)
```

Example:

```
label = Memory | chart sum(used) as Memory_Used by col_ts
```

toList

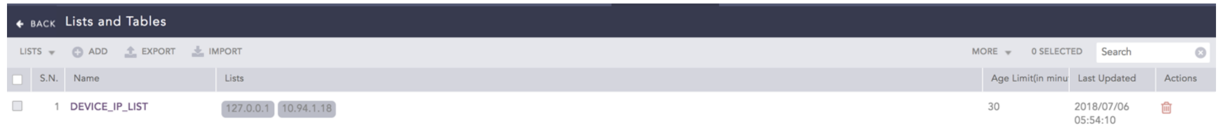
Populates the dynamic list with the field values of the search result.

Syntax:

```
| process toList (list_name, field_name)
```

Example:

```
device_ip=* | process toList(device_ip_list, device_ip)
```



toTable

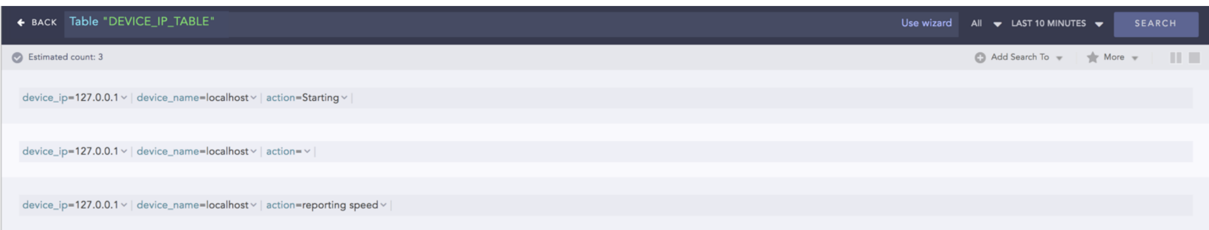
Populates the dynamic table with the fields and field values of the search result.

Syntax:

```
| process toTable (table_name, field_name1, field_name2,...., field_name9)
```

Example:

```
device_ip=* | process toTable(device_ip_table, device_name, device_ip,
action)
```



WhoIsLookup

Enriches the search result with the information related to the given field name from the WHOIS database.The WHOIS database consists of information about the registered users of an Internet resource such as registrar, IP address, registry expiry date, updated date, name server information and other information. If the specified field name and its corresponding value are matched with the equivalent field values of the WHOIS database, the process command enriches the search result, however, note that the extracted values are not saved.

Syntax:

```
| process whoislookup(field_name)
```

Example:

```
| chart distinct_list(log_ts) as log_ts_list, distinct_list(col_ts) as
col_ts_list
| process datetime_diff("seconds", log_ts_list, col_ts_list) as delta
| chart count() by log_ts_list, col_ts_list, delta`

  domain =* | process whoislookup(domain)
```

Filtering Commands

Filtering commands help you filter the search results.

search

To conduct searches on search results use the **search** command. It searches on dynamic fields returned from the norm, rex, and the table commands.

NOTE
It is not advised to use the search command unless absolutely necessary. The reason for this is that the search command uses heavy resources. So, it is always better to apply any kind of filtering before using the search command.

To search for users who have logged in more than 5 times:

```
login user = * | chart count() as count_user by user | search count_user >
5
```

If you create a dynamic field **new field** using norm command as,

```
| norm actual_mps = < new_field:int >
```

To view the logs which have **100** as the value of the new field, use the search command as:

```
| norm actual_mps = < new_field:int >|search new_field = 100
```

We recommend you to use the **search** command only in the following cases:

- When you need to filter the results for simple search (non key-value search).
  For example:
  ```
  | search error
  ```
- When you need to filter the results using the **or** logical operator.
  For example:
  ```
  | search device_name=localhost or col_type=filesystem
  ```

filter

The **filter** command lets you further filter the logs retrieved in the search results. SLS uses the **filter** command to drill-down on the search results. The **search** command is more efficient as it does not index intermediate fields.

NOTE

- The **filter** command filters the results based on dynamic fields returned from the **norm**, **rex**, and **table** commands as well.
- The **filter** command only works with expressions having the **=**, **>**, **<**, **>=**, and **<=** operators.
- To filter the results with more than one condition, you must chain multiple **filter** expressions.

Syntax:

```
<search query> | filter <condition>
```

For example, if you want to display only the domains that have more than 10 events associated with them in the search results, use the following query:

```
norm_id=*Firewall url=* | process domain(url) as domain | chart count() as
events by domain | filter events>10
```

The query searches for all the logs containing the fields **url** and **norm_id** with the value of **norm_id** having **Firewall** at the end. It then adds a new field **domain** to the logs based on the respective URLs and groups the results by their domains. Finally, the **filter** command limits the results to only those domains that have more than 10 events associated with them.

latest

The **latest** command finds the most recent log messages for every unique combination of provided field values.

```
| latest by device_ip | timechart count() by device_ip
```

This query searches for the latest logs of all the devices.

```
status = down port = 80 | latest on log_ts by device_ip
```

This query searches for all the latest devices based on the **log_ts** field whose web server running on the port number 80 is down.

order by

Use **order by** to sort the search results based on a numeric field in either **ascending** or **descending** order.

For simple searches that do not contain aggregation or correlation queries, the command can sort the search results based on only timestamp fields such as **log_ts** and **col_ts**. However, for other searches, all fields are supported.

Examples:

```
device_name= "John Doe" and col_type="syslog" | order by col_ts asc
```

This query searches for all the syslog messages generated from the device named **John Doe** and sorts them in the ascending order of their **col_ts** values.

```
device_name=* | order by log_ts desc
```

This query searches for the logs from all the devices in the system and sorts them in the descending order of their **log_ts** values.

NOTE
The sorting order of the search results is inconsistent when a search query does not contain an order sorting command. Use the **order by** command to make it consistent.

limit <number>

Use the **limit <number>** command to limit the number of results displayed. Additionally, you can add the **other** keyword at the end of the query to display the aggregation of the rest of the

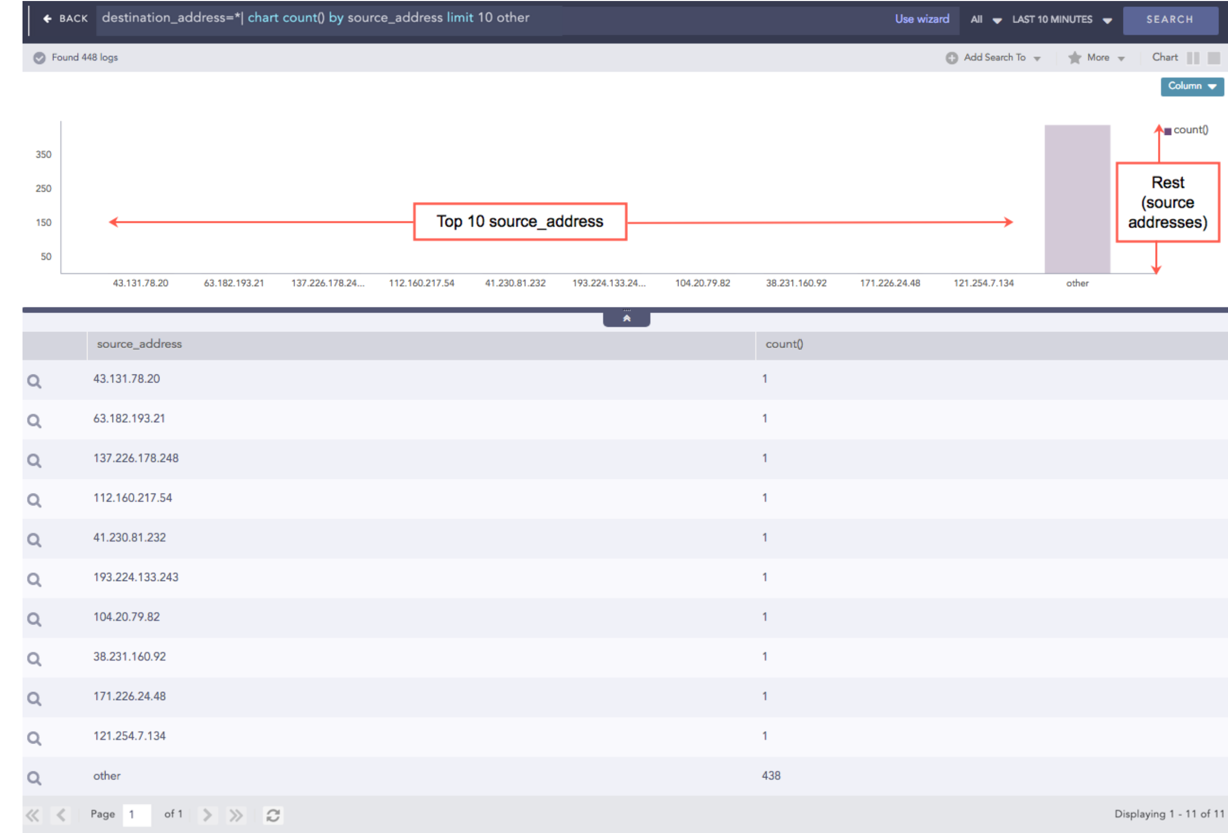
results.

> **NOTE**
> - The feature to display the **Top-10 and the Rest** graphs is supported for the aggregation queries.
> - While using the **limit <number>** command to retrieve a large volume of logs, make sure that your system has enough resources to load and render the data.

Example:

```
destination_address = * | chart count() by source_address limit 10 other
```

This query searches for all the logs having a **destination address**, filters the top 10 results by their **source address** and rolls-up all the remaining results in the eleventh line. The **source_address** field displays the word **other** in the table as shown in the figure below.



source_address	count()
43.131.78.20	1
63.182.193.21	1
137.226.178.248	1
112.160.217.54	1
41.230.81.232	1
193.224.133.243	1
104.20.79.82	1
38.231.160.92	1
171.226.24.48	1
121.254.7.134	1
other	438

Some other working examples:

```
device_ip=*| chart count() by action, source_address limit 5 other
| chart sum(actual_mps) by service limit 20 other
| chart count() by action limit 10 other
```

Pattern Finding

Pattern finding is a method of finding one or multiple streams and patterns of data to correlate a particular event. For example: five failed logins, followed by a successful login. It can be performed on the basis of the count and the time of occurrence of the stream. Use the **Pattern Finding** rules to detect complex event patterns in a large number of logs.

Correlation is the ability to track multiple types of logs and deduce meanings from them. It lets you look for a collection of events that make up a suspicious behavior and investigate further.

Single Stream

A stream consists of a count or occurrence of a query. The query can be a simple search query or an aggregating query. The stream can consist of a **having same** or a **within** expression. Stream has notion of time.

Syntax	Description
[ ]	For single streams, square brackets contain a stream of events.
within	Keyword to denote the notion of time frame
having same	Keyword

Following are the working examples for pattern finding using single stream:

To find 5 login attempts:

```
[5 action = "logged on"]

[5 login]
```

To find 5 login attempts within a timeframe of 2 minutes:

```
[5 action = "logged on" within 2 minutes]

[5 login within 2 minutes]
```

To find 5 login attempts by the same user:

```
[5 action = "logged on" having same user]
[5 login having same user]
```

To find 10 login attempts by the same user from the same source_address (multiple fields) within 5 minutes:

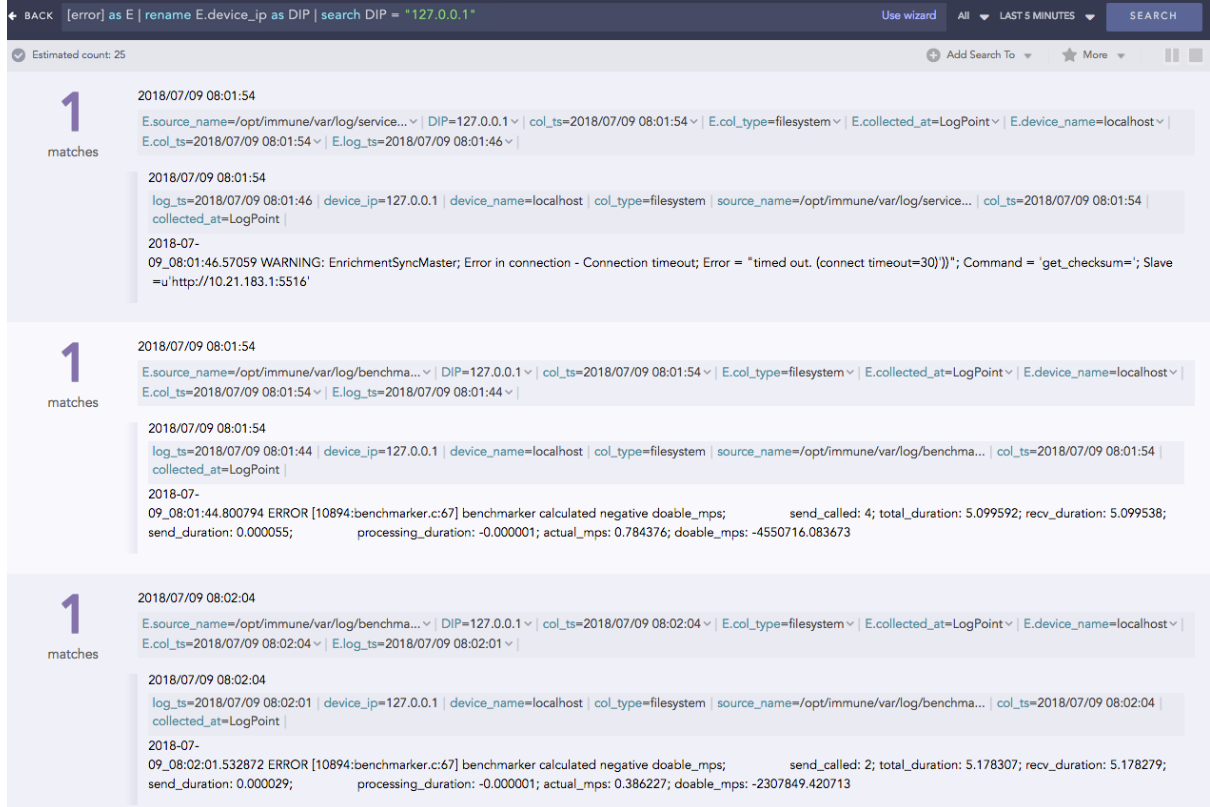
```
[10 action = "logged on" having same user, source_address within 5
minutes]
```

The time format for specifying timeframe are: second(s), minute(s), hour(s) and day(s).

```
[error] as E
```

This query finds the logs with errors. It then aliases the result as E and displays the fields prefixed with E such as E.severity, and E.device_ip. You can then use the aliased fields as shown below:

```
[error] as E | rename E.device_ip as DIP | search DIP = "127.0.0.1"
```



Pattern finding queries for different conditions:

10 login to localhost (source_address) by the same user for the last 15 minutes.

```
[10 login source_address = 127.0.0.1 having same user_name within 15
minutes]
```

The field of a log file with a norm command .

```
[2 login | norm <username:word> login successful having same username
within 10 seconds]
```

Multiple Streams

You can join multiple patterns by using **Pattern Finding by Joining Streams** and **Pattern Finding by Following Streams**.

Left Join

You can use a left join to return all the values from the table or stream on the left, and only the common values from the table or stream on the right.

Example:

```
[table event_prob] as s1
left join [event = * | chart count() by event] as s2
on s1.event = s2.event
```

Right Join

You can use a right join to return all the values from the table or stream on the right and only the common values from the table or stream on the left.

Example:

```
[5 transaction error having same user within 30 seconds] as s1
right join [transaction successful] as s2
on s1.user=s2.user
```

Join

Join queries are used to link the results from different sources. The link between two streams must have an **on** condition. The link between two lookup sources or any of the lookup and stream does not require a time-range. Join as a part of a search string, can link one data-set to another based on one or more common fields. For instance, two completely different data-sets can be linked together based on a username or event ID field present in both the data-sets.

The syntax for joining multiple patterns is as follows:

[stream 1] <aliased as s1> <JOIN> [stream 2] <aliased as s2> on <Join_conditions> | additional filter query.

```
[action = locked] as locked
join
[action = unlocked] as unlocked
on
locked.target_user = unlocked.target_user
| chart count() by locked.target_user, locked.caller_computer,
unlocked.caller_user
```

```
[login] as l join [table User] as u on l.user = u.user
```

To find the events where a reserved port of an Operating System (inside the PORT_MACHINE table) is equal to the blocked port (inside the BLOCKED_PORT table):

```
[table PORT_MACHINE port<1024] as s1 join [table BLOCKED_PORT] as s2 on
s1.port=s2.port
```

To find 5 login attempts by the same user within 1 minute followed by 5 failed login attempts by the same user within 1 minute

```
[5 login having same user within 1 minute] as s1
followed by
[5 failed having same user within 1 minute]
```

To find 5 login attempts by the same user within 1 minute followed by 5 failed attempts by the same user within 1 minute and users from both result are same

```
[5 login having same user within 1 minute] as s1
followed by
[5 failed having same username within 1 minute] as s2 on s1.username =
s2.username
```

Followed by

Pattern Finding by followed by is useful when two sequential streams are connected to an action.

For example:

```
[2 login success having same user] AS stream1
followed by
[login failure] as stream2
ON
stream1.user = stream2.user
```

Here,

Syntax	Description
[ ] AS stream1	A simple pattern finding query aliased as stream1
followed by	Keyword
[ ] AS stream2	A simple search aliased as stream2
ON	Keyword
stream1.user = stream2.user	Matching field from the 2 streams

The syntax for joining multiple patterns is as follows:

- [stream 1] <aliased as s1> <followed by> [stream 2] <aliased as s2> <within time limit> on <Join_conditions>| additional filter query.
- [stream 1] as s1 followed by [stream2] as s2 within time_interval on s1.field = s2.field
- [stream 1] as s1 followed by [stream2] as s2 on s1.field = s2.field
- [stream 1] as s1 followed by [stream2] as s2 within time_interval

The inference derived from the above queries:

- Streams can be labeled using alias. Here, the first stream is labeled as s1. This labeling is useful while setting the join conditions in the join query.
- The operation between multiple streams is carried out using "followed by" or "join".
- Use the **followed by** keyword to connect two sequential streams anticipating an action, e.g., multiple login attempts followed by successful login.
- Use the **join** keyword to view additional information in the final search. The **join** syntax is mostly used with tables for enriching the data.
- Time limit for occurrence can also be specified.
- If you use the **join** keyword, then specify the **on** condition.
- Join conditions are simple mathematical operations between the data-sets of two streams.
- Use additional filter query to mitigate false positives which are generally created while joining a stream and a table. Searching the query with a distinct key from the table displays an error-less result.

```
[| chart count() by device_ip] AS lookup
JOIN
[device_ip=*] AS log ON lookup.device_ip = log.device_ip
```

This query does not display histogram but displays the log table.

```
[device_ip=*] as log join [| chart count() by device_ip] as lookup on
log.device_ip=lookup.device_ip
```

This query displays both the histogram and the log table.

Chaining of commands

You can chain multiple commands into a single query by using the pipe (|) character. Any command except **fields** can appear before or after any other command. The **fields** command must always appear at the end of the command chain.

Example:

```
| chart count() as cnt by device_name | search cnt > 1000
```

This query displays the number of logs with the same **device_name** appearing more than 1000 times.

```
(label = logoff) AND hour (log_ts) > 8 AND hour (log_ts) <16 |
latest by user |
timechart count() by user
```

This query captures all the log messages labeled as **logoff** and those collected between 8 AM and 4 PM. It then displays the timechart of the recent users for the selected time-frame.

Additional Notes

Process or Count

Count and **process** are keywords and must be enclosed within double quotes.

```
MsWinEventLog product=* | chart count() as "Count" by product
order by count() desc limit 10
```

Similarly,

```
MsWinEventLog product=* "process"=* action=*
| fields product, "process", action, object
```

Conditional Expression

Conditional expression within parenthesis () must be separated explicitly by **or**.

```
| chart count(label = delete or label = remove) as remove
```

Forward Slash Expression

Any expression after the forward slash must be enclosed within double quotes.

```
source_name = "/opt/immune/var/log/audit/webserver.log"
| chart count() by source_address
```

norm

```
| norm doable_mps=<dmps:'['0-9']'+>
```

```
| norm <:'\['><my_field:word><:'\]'> | chart count() by my_field
```

timechart

Limit does not work with timechart.

```
| timechart count() by col_type
```

Capturing normalized field values

Use norm on command to capture normalized field value in log search result.

```
Suppose the log search result consists of a log value pair

source_name = /opt/immune/var/log/benchmarker

Now, if you want to capture the first two words of the path, you can write
the query as follows:

| norm on source_name <capture:'\/opt\/immune'>
```

This feature works well with rex command too.

```
user=* | rex on user:\s+(?P<account>\S+)@(?P<domain>\S+)
| chart count() by account, domain | search account=*
```

In the example above, the rex command is used on a field which captures email addresses. The email address is then broken into account and domain using the corresponding regex.

Grok Patterns

SLS search recognizes the following Grok patterns.

General Patterns

Pattern name	Regular expression
USERNAME	[a-zA-Z0-9._-]+
USER	%{USERNAME}
INT	(?:[+-]?(?:[0-9]+))
BASE10NUM	(?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:.[0-9]+)?)|(?:.[0-9]+)))
NUMBER	(?:%{BASE10NUM})
BASE16NUM	(?<![0-9A-Fa-f])(?:[+-]?(?:0x)?(?:[0-9A-Fa-f]+))
BASE16FLOAT	\b(?<![0-9A-Fa-f.])(?:[+-]?(?:0x)?(?:(?:[0-9A-Fa-f]+(?:.[0-9A-Fa-f]*)?)|(?:.[0-9A-Fa-f]+)))\b
POSINT	\b(?:[1-9][0-9]*)\b
NONNEGINT	\b(?:[0-9]+)\b
WORD	\b\w+\b
NOTSPACE	\S+
SPACE	\s*
DATA	.*?
GREEDYDATA	.*
QUOTEDSTRING	(?>(?<!\)(?>”(?>.|[^"]+)+”|””|(?>'(?>\\.|[^\\']+)+')|“|(?>` (?>.|[^`]+)+` )|` `))
UUID	[A-Fa-f0-9]{8}-(?:[A-Fa-f0-9]{4}-){3}[A-Fa-f0-9]{12}
DOMAINTLD	[a-zA-Z]+
EMAIL	%{NOTSPACE}@%{WORD}.%{DOMAINTLD}
QS	%{QUOTEDSTRING}

Networking-related Patterns

Pattern name	Regular expression
MAC	(?:%{CISCOMAC}|%{WINDOWSMAC}|%{COMMONMAC})
CISCOMAC	(?:(?:[A-Fa-f0-9]{4}.){2}[A-Fa-f0-9]{4})
WINDOWSMAC	(?:(?:[A-Fa-f0-9]{2}-){5}[A-Fa-f0-9]{2})

<table>
<tr><th>Pattern name</th><th>Regular expression</th></tr>
<tr><td>COMMONMAC</td><td>(?:(?:[A-Fa-f0-9]{2}:){5}[A-Fa-f0-9]{2})</td></tr>
<tr><td>IPV6</td><td>((([0-9A-Fa-f]{1,4}:){7}([0-9A-Fa-f]{1,4}|:))|(([0-9A-Fa-f]{1,4}:){6}(:[0-9A-Fa-f]{1,4}|((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3})|:))|(([0-9A-Fa-f]{1,4}:){5}(((:[0-9A-Fa-f]{1,4}){1,2})|:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3})|:))|(([0-9A-Fa-f]{1,4}:){4}(((:[0-9A-Fa-f]{1,4}){1,3})|((:[0-9A-Fa-f]{1,4})?:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){3}(((:[0-9A-Fa-f]{1,4}){1,4})|((:[0-9A-Fa-f]{1,4}){0,2}:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){2}(((:[0-9A-Fa-f]{1,4}){1,5})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){1}(((:[0-9A-Fa-f]{1,4}){1,6})|((:[0-9A-Fa-f]{1,4}){0,4}:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3}))|:))|(:(((:[0-9A-Fa-f]{1,4}){1,7})|((:[0-9A-Fa-f]{1,4}){0,5}:((25[0-5]|2[0-4]d|1dd|[1-9]?d)(.(25[0-5]|2[0-4]d|1dd|[1-9]?d)){3}))|:)))(%.+)?</td></tr>
<tr><td>IPV4</td><td>(?<![0-9])(?:(?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2}))(?![0-9])</td></tr>
<tr><td>IP</td><td>(?:%{IPV6}|%{IPV4})</td></tr>
<tr><td>HOSTNAME</td><td>b(?:[0-9A-Za-z][0-9A-Za-z-]{0,62})(?:.(?:[0-9A-Za-z][0-9A-Za-z-]{0,62}))*(.?|b)</td></tr>
<tr><td>HOST</td><td>%{HOSTNAME}</td></tr>
<tr><td>IPORHOST</td><td>(?:%{HOSTNAME}|%{IP})</td></tr>
<tr><td>HOSTPORT</td><td>%{IPORHOST}:%{POSINT}</td></tr>
</table>

Path-related patterns

<table>
<tr><th>Pattern name</th><th>Regular expression</th></tr>
<tr><td>PATH</td><td>(?:%{UNIXPATH}|%{WINPATH})</td></tr>
<tr><td>UNIXPATH</td><td>(?>/(?>[w_%!$@:.,-]+|.)*)+</td></tr>
<tr><td>TTY</td><td>(?:/dev/(pts|tty([pq])?)(w+)?/?(?:[0-9]+))</td></tr>
<tr><td>WINPATH</td><td>(?>[A-Za-z]+:|\)(?:\[^\?*]*)+</td></tr>
<tr><td>URIPROTO</td><td>[A-Za-z]+(+[A-Za-z+]+)?</td></tr>
<tr><td>URIHOST</td><td>%{IPORHOST}(?::%{POSINT:port})?</td></tr>
<tr><td>URIPATH</td><td>(?:/[A-Za-z0-9$.+!*'(){},~:;=@#%_-]*)+</td></tr>
<tr><td>URIPARAM</td><td>?[A-Za-z0-9$.+!*'|(){},~@#%&/=:;_?-[]]*</td></tr>
<tr><td>URIPATHPARAM</td><td>%{URIPATH}(?:%{URIPARAM})?</td></tr>
<tr><td>URI</td><td>%{URIPROTO}://(?:%{USER}(?::[^@]*)?@)?(?:%{URIHOST})? (?:%{URIPATHPARAM})?</td></tr>
</table>

Date and time patterns

<table>
<tr><th>Pattern name</th><th>Regular expression</th></tr>
<tr><td>MONTH</td><td>b(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|Jun(?:e)?|Jul(?:y)? |Aug(?:ust)?|Sep(?:tember)?|Oct(?:ober)?|Nov(?:ember)?|Dec(?:ember)?)b</td></tr>
</table>

Pattern name	Regular expression
MONTHNUM	(?:0?[1-9]|1[0-2])
MONTHNUM2	(?:0[1-9]|1[0-2])
MONTHDAY	(?:(?:0[1-9])|(?:[12][0-9])|(?:3[01])|[1-9])
DAY	(?:Mon(?:day)?|Tue(?:sday)?|Wed(?:nesday)?|Thu(?:rsday)?|Fri(?:day)? |Sat (?:urday)?|Sun(?:day)?)
YEAR	(?>dd){1,2}
HOUR	(?:2[0123]|[01]?[0-9])
MINUTE	(?:[0-5][0-9])
SECOND	(?:(?:[0-5]?[0-9]|60)(?:[:.,][0-9]+)?)
TIME	(?!<[0-9])%{HOUR}:%{MINUTE}(?::%{SECOND})(?![0-9])
DATE_US	%{MONTHNUM}[/-]%{MONTHDAY}[/-]%{YEAR}
DATE_EU	%{MONTHDAY}[./-]%{MONTHNUM}[./-]%{YEAR}
ISO8601_TIMEZONE	(?:Z|[+-]%{HOUR}(?::?%{MINUTE}))
ISO8601_SECOND	(?:%{SECOND}|60)
TIMESTAMP_ISO8601	%{YEAR}-%{MONTHNUM}-%{MONTHDAY}[T]%{HOUR}:?%{MINUTE} (?::?%{SECOND})?%{ISO8601_TIMEZONE}?
DATE	%{DATE_US}|%{DATE_EU}
DATESTAMP	%{DATE}[-]%{TIME}
TZ	(?:[PMCE][SD]T|UTC)
DATESTAMP_RFC822	%{DAY} %{MONTH} %{MONTHDAY} %{YEAR} %{TIME} %{TZ}
DATESTAMP_RFC2822	%{DAY}, %{MONTHDAY} %{MONTH} %{YEAR} %{TIME} %{ISO8601_TIMEZONE}
DATESTAMP_OTHER	%{DAY} %{MONTH} %{MONTHDAY} %{TIME} %{TZ} %{YEAR}
DATESTAMP_EVENTLOG	%{YEAR}%{MONTHNUM2}%{MONTHDAY}%{HOUR}%{MINUTE}%{SECOND}

Syslog patterns

Pattern name	Regular expression
SYSLOGTIMESTAMP	%{MONTH} +%{MONTHDAY} %{TIME}
PROG	(?:[w._/%-]+)
SYSLOGPROG	%{PROG:program}(?:[%{POSINT:pid}])?
SYSLOGFACILITY	<%{NONNEGINT:facility}.%{NONNEGINT:priority}>
HTTPDATE	%{MONTHDAY}/%{MONTH}/%{YEAR}:%{TIME} %{INT}
SYSLOGHOST	%{IPORHOST}

Log formats

Pattern name	Regular expression
SYSLOGBASE	%{SYSLOGTIMESTAMP:timestamp} (?:%{SYSLOGFACILITY})?%{SYSLOGHOST:logsource} %{SYSLOGPROG}:
COMMONAPACHELOG	%{IPORHOST:clientip} %{USER:ident} %{USER:auth} [%{HTTPDATE:timestamp}] "(?:%{WORD:verb} %{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})" %{NUMBER:response} (?:%{NUMBER:bytes}|-)
COMBINEDAPACHELOG	%{COMMONAPACHELOG} %{QS:referrer} %{QS:agent}

Further reading

Additional information and answers to questions you may have about SLS are available in the **Stormshield knowledge base** (authentication required).

documentation@stormshield.eu

All images in this document are for representational purposes only, actual products may differ.

Copyright © Stormshield 2024. All rights reserved. All other company and product names contained in this document are trademarks or registered trademarks of their respective companies.