LOGPOINT

Integrations

Evaluation Process Plugin

*V*5.1.0

CONTENTS

CHAPTER

ONE

EVALUATION PROCESS PLUGIN

Evaluation Process Plugin installs the eval process command. This command evaluates mathematical, boolean and string expressions during a Logpoint search and adds the evaluation result in an identifier as a new field.

When you use the eval process command, make sure

- Identifier name is not the same as an existing field name. If it is, Logpoint discards the value of the identifier.
- When using a string value in an eval expression, always place the string within single quotes (‘’).
- Eval expressions use an existing key from an event.
- Invalid expression or syntax mismatches do not generate an exception or error.

Syntax:

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

- **Identifier**: Contains the result of evaluating expressions.
- **Expression**: A combination of numbers, variables, operators, functions, brackets and punctuation marks grouped to represent a value.

Example:

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

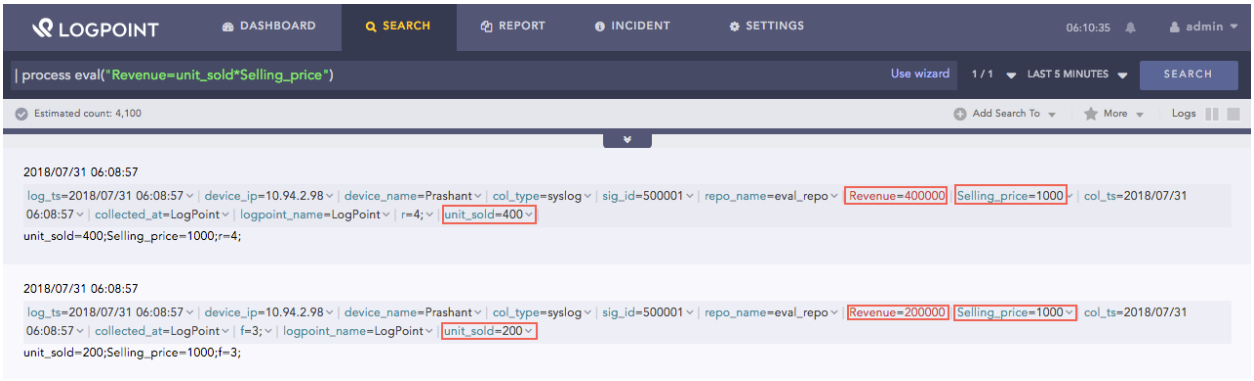


Fig. 1: Using eval Expression

Here, the query calculates the value of **Revenue** by multiplying the values of **unit_sold** and **Selling_price**.

CHAPTER

TWO

INSTALLING EVALUATION PROCESS PLUGIN

Prerequisite

Logpoint v6.2.0 or later

To install Evaluation Process Plugin:

1. Download the .pak file from the Help Center.
2. Go to *Settings >> System Settings* from the navigation bar and click **Applications**.
3. Click **Import**.
4. **Browse** to the downloaded .pak file.
5. Click **Upload**.

After installing Evaluation Process Plugin, you can find it under *Settings >> System Settings >> Plugins*.

CHAPTER

THREE

UNINSTALLING EVALUATION PROCESS PLUGIN

1. Go to *Settings >> System Settings* from the navigation bar and click **Applications**.
2. Click the **Uninstall** icon from **Actions** of Evaluation Process Plugin.
3. Click **Yes**.

CHAPTER

FOUR

ARITHMETIC EXPRESSIONS

Is the combination of numbers, operators and variables that results in a numeric value.

Generic Syntax:

```
| process eval("identifier = first_operand arithmetic_operator second_operand")
```

4.1 Addition (+)

Accepts numerical values as input for **addition** and generates the output in the destination field (identifier). It also accepts string values as input and returns the concatenation of values as the output.

Example:

```
| process eval("total_datasize=request_datasize+response_datasize")
| fields request_datasize, response_datasize, total_datasize
```

Here, the query calculates the value of the **total_datasize** identifier by adding the value of the **request_datasize** and **response_datasize** fields.

The *fields* command displays the corresponding values of all the three fields in a tabular form.

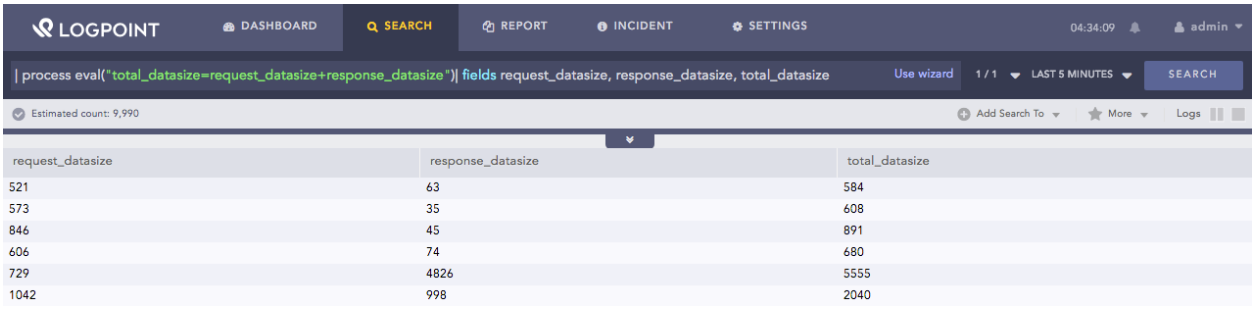


Fig. 1: Using addition function

4.2 Subtraction (-)

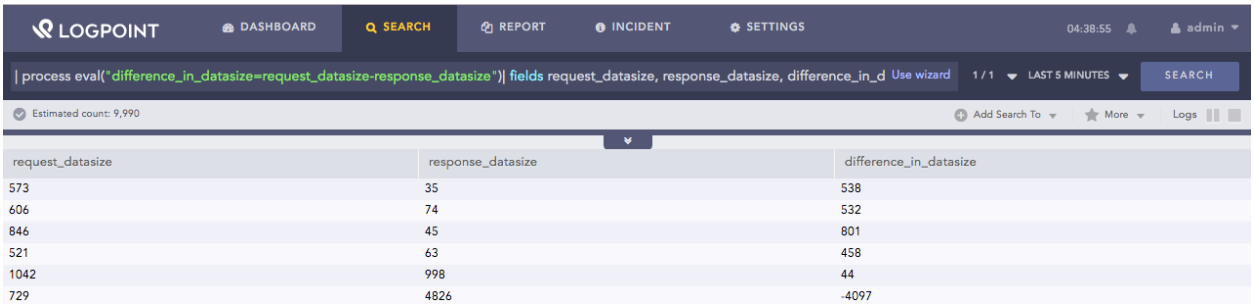
Accepts numerical values as input for **subtraction** and generates the output in the destination field (identifier).

Example:

```
process eval("difference_in_datasize=request_datasize-response_datasize")
fields request_datasize, response_datasize, difference_in_datasize
```

Here, the query calculates the value of the **difference_in_datasize** identifier by subtracting the values of the **response_datasize** field from the **request_datasize** field.

The *fields* command displays the corresponding values of all three fields in a tabular form.



request_datasize	response_datasize	difference_in_datasize
573	35	538
606	74	532
846	45	801
521	63	458
1042	998	44
729	4826	-4097

Fig. 2: Using subtraction function

4.3 Multiplication (*)

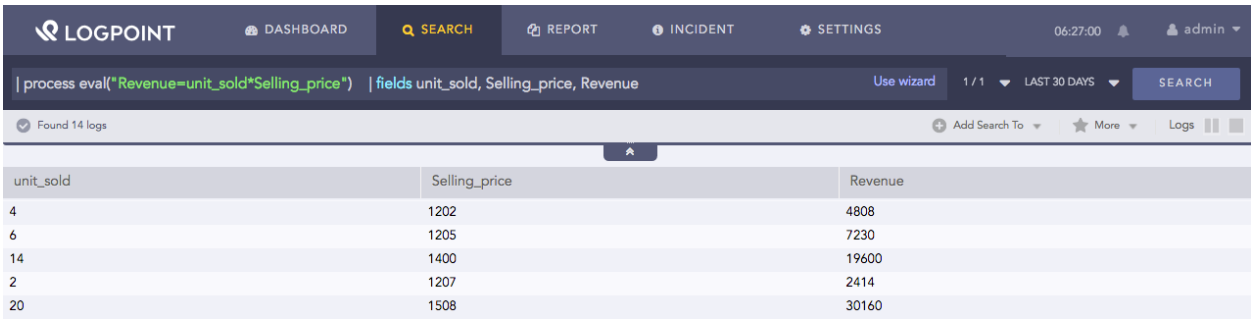
Accepts numerical values as input for **multiplication** and generates the output in the destination field (identifier).

Example:

```
process eval("Revenue=unit_sold*Selling_price")
fields unit_sold, Selling_price, Revenue
```

Here, the query calculates the value of the **Revenue** identifier by multiplying the values of the **unit_sold** and **Selling_price** fields.

The *fields* command displays the corresponding values of all the three fields in a tabular form.



unit_sold	Selling_price	Revenue
4	1202	4808
6	1205	7230
14	1400	19600
2	1207	2414
20	1508	30160

Fig. 3: Using multiplication function

4.4 Division (/)

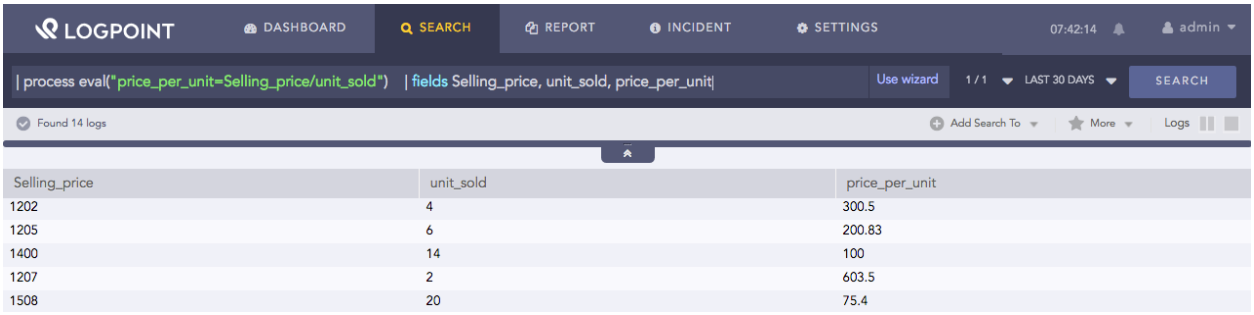
Accepts numerical values as input for the **division** and generates the output in the destination field (identifier).

Example:

```
| process eval("price_per_unit=Selling_price/unit_sold")
| fields Selling_price, unit_sold, price_per_unit
```

Here, the query calculates the value of the **price_per_unit** identifier by dividing the values of the **unit_sold** and **Selling_price** fields.

The *fields* command displays the corresponding values of all the three fields in a tabular form.



Selling_price	unit_sold	price_per_unit
1202	4	300.5
1205	6	200.83
1400	14	100
1207	2	603.5
1508	20	75.4

Fig. 4: Using division function

4.5 Modulus (%)

Accepts numerical values as input for the division and generates the remainder of the division as the output in the destination field (identifier).

Example:

```
process eval("modulo= Selling_price % cost_price ")
fields Selling_price, cost_price, remainder
```

Here, the query calculates the value of the **modulo** identifier by finding the remainder after dividing the value of the **Selling_price** field by **cost_price** field.

The *fields* command displays the corresponding values of all the three fields in a tabular form.

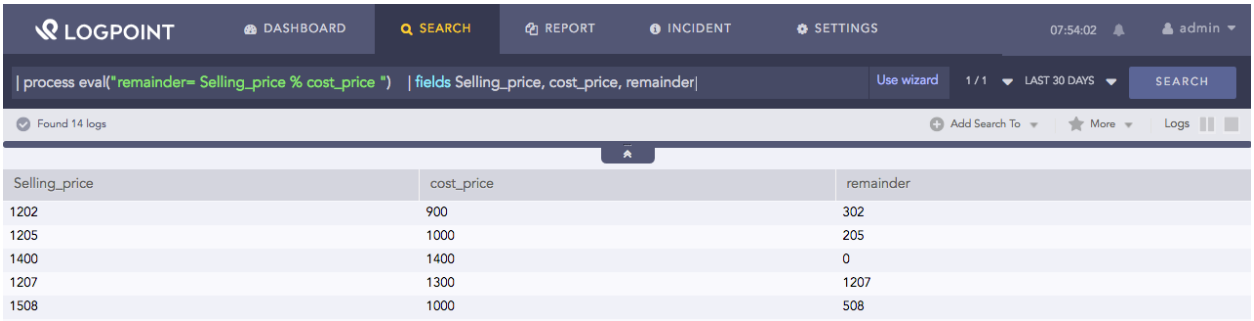


Fig. 5: Using modulus function

4.6 Power (^)

Accepts numerical values as input for the **power** operation and generates the output in the destination field (identifier).

Example:

```
process eval("area_square=length^2")
fields length, area_square
```

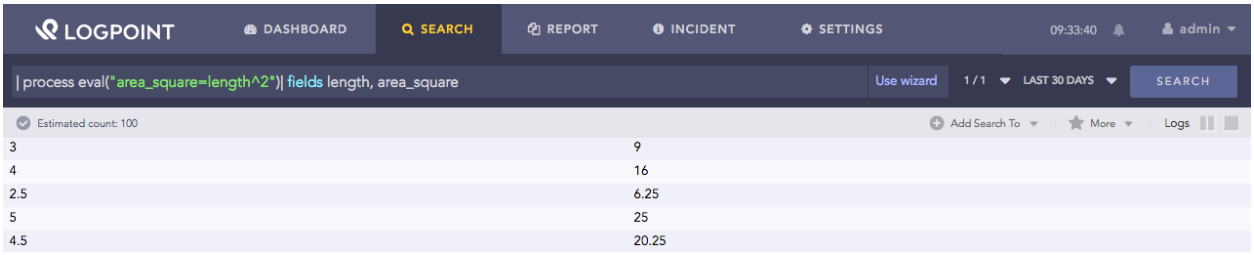


Fig. 6: Using power function

Here, the query calculates the value of the **area_square** identifier by squaring the value of the **length** field.

The *fields* command displays the corresponding values of the two fields in a tabular form.

CHAPTER

FIVE

CONDITIONAL AND COMPARISON FUNCTIONS

Evaluates conditions and compare values.

5.1 If Statement

The if statement takes a condition and a string value, X, and evaluates the condition; if it's true, it returns the value of X. When using a negative number in an if condition, it should be enclosed in parentheses '()' for clarity.

Syntax:

```
process eval("identifier=if(condition) {return X}")
```

Example:

```
process eval("User_severity=if(risk_score >= 5) {return 'Risk user'}")
chart count() by risk_score, User_severity
```

Fig. 1: Using if statement function

Here, the query checks if the **risk_score** field's value is greater than or equal to 5 and returns **Risk user** value in the **User_severity** identifier.

The *chart count()* command displays the count of the combination of **risk_score** and **User_severity** values as a chart and in a tabular form.

5.2 If-else Statement

The if-else statement takes a condition and two string values, X and Y, and evaluates the condition; if it's true, it returns the value of X; if it isn't, it returns the value of Y. When using a negative number in an if condition, it should be enclosed in parentheses '()' for clarity.

Syntax:

```
| process eval("identifier=if(condition) {return X} else {return Y}")
```

Example:

```
| process eval("is_profitloss=if((Selling_price%cost_price) == 0)
 {return 'No profit/loss'} else {return 'profit/loss'}")
| fields Selling_price, cost_price, is_profitloss
```

Fig. 2: Using if-else statement function

Here, the query checks if the remainder value when **Selling_price** field is divided by **cost_price** field is 0. It returns **No profit/lost** in the **is_profitloss** identifier if the condition is true, if it isn't it returns **profit/loss**.

The *fields* command displays the value of **Selling_price**, **cost_price** and **is_profitorloss** in a tabular form.

5.3 If-elseif-else Statement

It takes one or more alternating conditions and values. It compares the *condition* in the following order.

- If the first *condition* is true, it returns the value provided in *X*,
- If it isn't true, it compares the second *condition*;
- If the second *condition* is true, it returns the value provided in *Y*,
- If it isn't true, it returns the value provided in *Z*.

When using a negative number in an if condition, it should be enclosed in parentheses '()' for clarity.

Syntax:

```
| process eval("identifier=if(condition){return X} else-if(condition) {return Y} else { return Z}")
```

Example:

```
| process eval("User_severity=if(risk_score > 5) {return 'Risk user'}
 else-if(risk_score<=0) {return 'No risk'} else {return 'Normal user'}")
| fields risk_score, User_severity
```

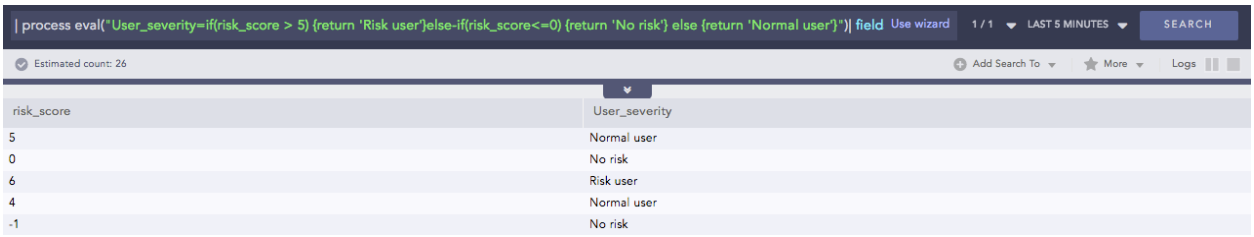


Fig. 3: Using if-elseif-else statement function

Here, the query checks if the **risk_score** field's value is greater than 5. It returns **Risk user** in the **User_severity** identifier if the condition is true, if it isn't it compares the second condition. It checks if the **risk_score** field's value is less than or equal to 0 and returns **No risk** if true. If both of these conditions is false, it returns **Normal user**.

The *fields* command displays the value of **risk_score** and **User_severity** in a tabular form.

5.4 Case Statement

Accepts one or more alternating conditions and values. It compares the *condition* with the following order.

- if *case_one* matches the value of the *data*, it returns the value provided in *X*,
- if it doesn't match, it checks if the *case_two* matches the value of the *data*;

- if the condition is true, it returns the value in *Y*,
- if it isn't it returns the value in *Z* by default.

Syntax:

```
| process eval("identifier=switch(data) {case(case_one) {return X}
 case(case_two) {return Y} default {return Z}}")
```

Example:

```
| process eval("Access_type=switch(action) {case('allow') {return 'Allow access'}
 case('deny') {return 'Deny access'} default {return 'Forward access'}}")
| fields action, Access_type
```

Fig. 4: Using case statement function

Here, the query returns **Allow access** in the **Access_type** identifier if the **action** field's value is **access**. If it isn't, it checks if the value of **action** is **deny** and returns **Deny access**. If both values don't match, it returns **Forward access** by default.

The *fields* command displays the value of **action** and **Access_type** in a tabular form.

5.5 cidrmatch

Accepts two arguments, a *CIDR* (Classless Inter-Domain Routing) notation, and an *IP* address. It returns *true* if the *IP* address matches the *CIDR* notation, if it doesn't it returns *false*.

Syntax:

```
| process eval("identifier=cidrmatch(CIDR, IP)")
```

Example:

```
| process eval("is_local_ip=cidrmatch('127.0.0.0/8', device_ip)")
```

Fig. 5: Using cidrmatch function

Here, the query returns **true** in the **is_local_ip** identifier if the **device_ip** field's value matches the CIDR notation 127.0.0.0/8, if it doesn't match it returns **false**.

5.6 coalesce

Accepts an arbitrary number of arguments as input and returns the value of the first argument that is not null.

Syntax:

```
| process eval("identifier=coalesce(X,Y,...)")
```

Example:

```
| process eval("ip_add=coalesce(ip_address,device_ip)")
| fields ip_address, device_ip, ip_add
```

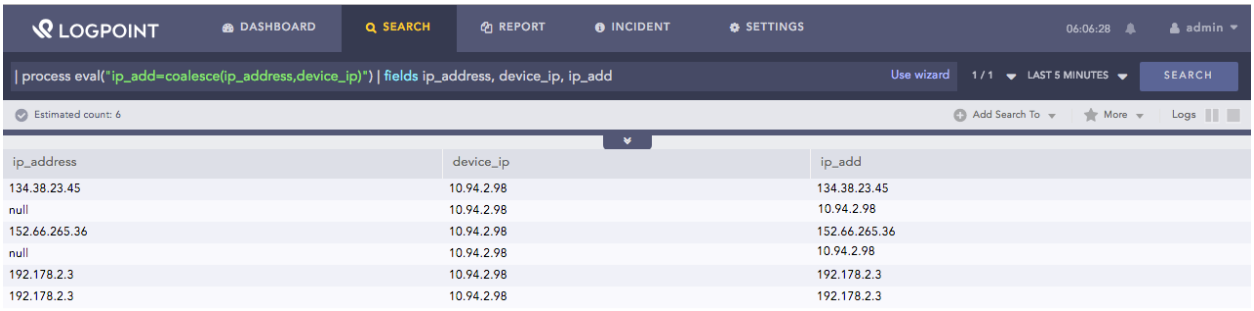


Fig. 6: Using coalesce function

Here, the query returns the **ip_address** field's value in the **ip_add** identifier if the value is not null. If it is null, it checks the value of the **device_ip** field. If the **device_ip** field's value is not null, it returns its value in the **ip_add** identifier.

The *fields* command displays the value of **ip_address**, **device_ip** and **ip_add** in a tabular form.

5.7 false

Returns *false*. The function in combination with other functions represents a condition that is absolutely false, 1==0. Unlike other functions, this function does not take any argument.

Syntax:

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

Example:

```
| process eval("is_profit=if(Selling_price > cost_price) {return true()} else {return false()}")
| chart count() by Selling_price, cost_price, is_profit
```

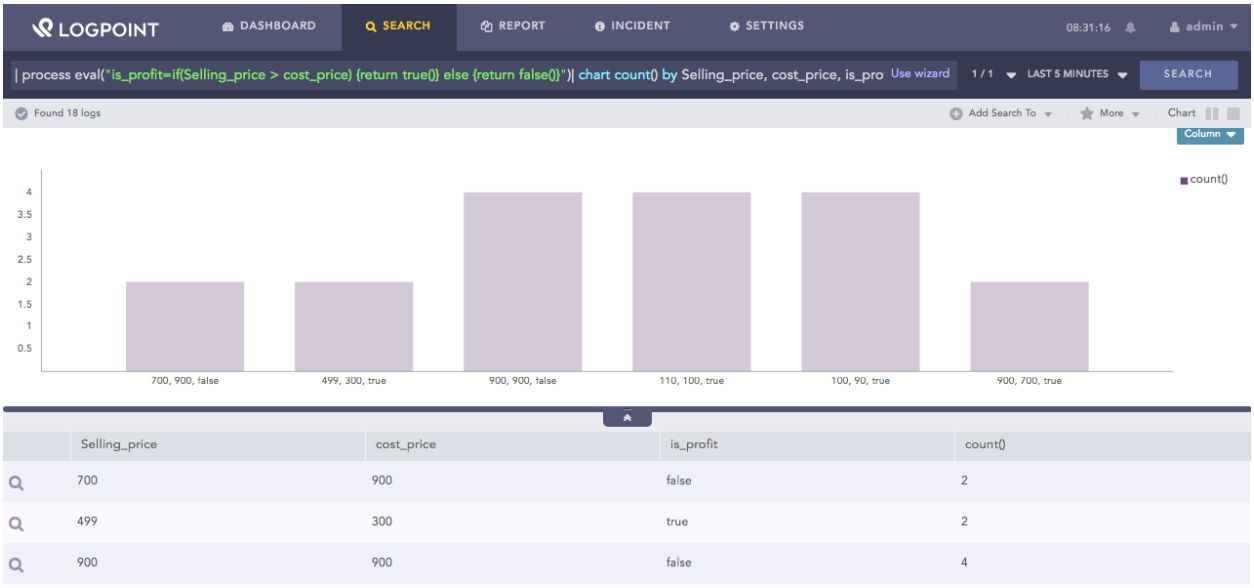


Fig. 7: Using false function

Here, the query checks the value in the **Selling_price** and **cost_price** fields. It returns **true** in the **is_profit** identifier if the **Selling_price** is greater than the **cost_price**, if it isn't it returns **false**.

The *chart count()* command displays the count of the combination **Selling_price** and **cost_price** values as a chart and in a tabular form.

5.8 in

Accepts a *field* of an event and a list of string values. It returns *true* if one of the values in the list matches the value specified in the *field*, if it doesn't it returns *false*.

Syntax:

```
| process eval("identifier=in(field, value1, value2, value3, ...)")
```

Example:

```
| process eval("isUserAdmin=in(user, 'Administrator', 'administrator', 'Admin', 'admin')")
| chart count() by user, isUserAdmin
```

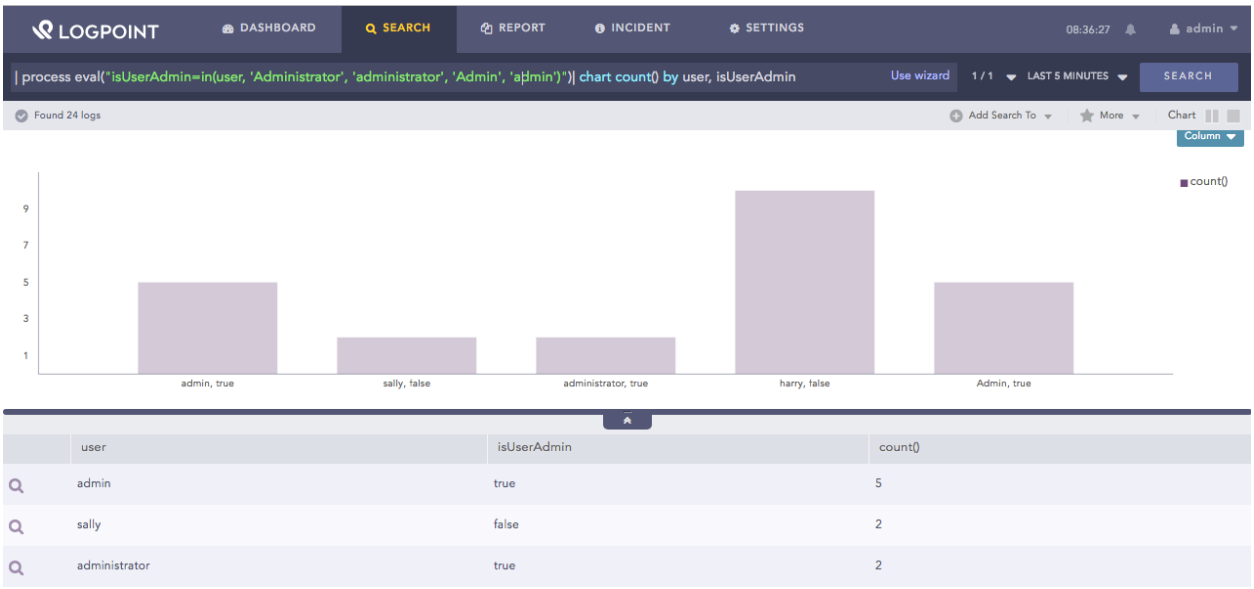


Fig. 8: Using in function

Here, the query returns **true** in the **isUserAdmin** identifier if the **user** field's value matches with any one value in the list: *Administrator*, *administrator*, *Admin* and *admin*, if it does't it returns *false*.

The *chart count()* command displays the count of the combination of **user** and **isUserAdmin** values as a chart and in a tabular form.

5.9 match

Accepts a text field *X* and a *regex* (regular expression) string. It returns *true* or *false* based on whether the given regular expression finds a match against any substring of the text in the field *X*.

It also returns *true* if the text in *regex* string exactly matches the text in the field *X*.

Syntax:

```
| process eval("identifier=match(X, regex)")
```

Example:

```
| process eval("is_coltype_filesystem=match(col_type,'file.*')") | chart count() by col_type, is_
↪coltype_filesystem
```

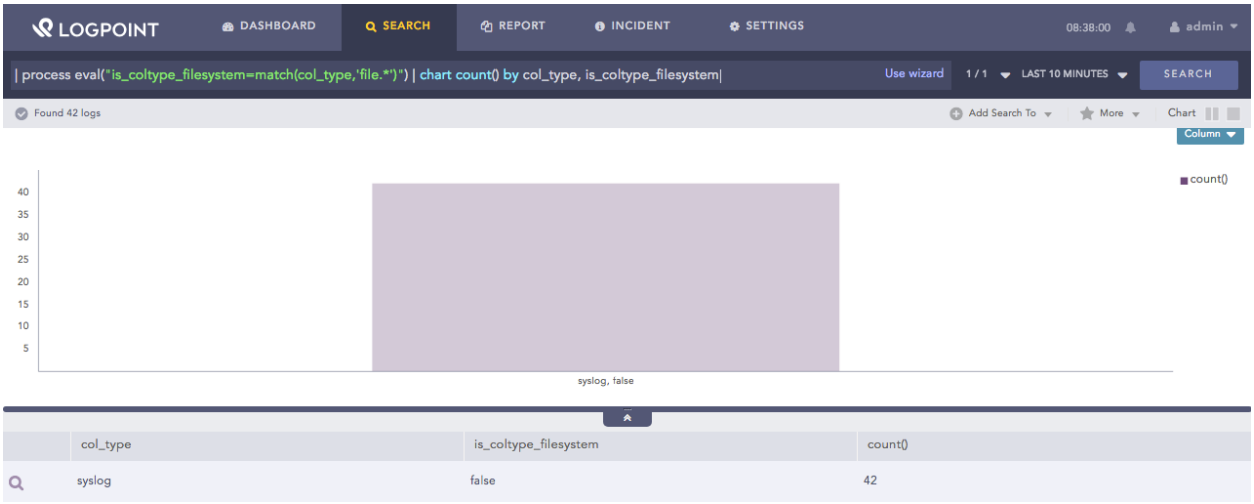


Fig. 9: Using match function

Here, the query compares the regex string **file.** with the value in the **col_type** field. It returns **true** in the **is_coltype_filesystem** identifier if the pattern is an exact match or is a substring of the **col_type** field's value, if it doesn't match it returns **false**.

The *chart count()* command displays the count of the combination of **col_type** and **is_coltype_filesystem** values as a chart and in a tabular form.

5.10 like

Accepts a text field *X* and a *pattern*. It returns *true* if the text in *X* matches the given *pattern*, if it doesn't match it returns *false*. This function also returns *true* if the text in the pattern exactly matches the text in the *X* field.

The *pattern* supports a regular expression as well as the percent character (%) for wildcards and an underscore character (_) for a single character match.

Syntax:

```
| process eval("identifier=like(X, pattern)")
```

Example:

```
process eval("is_coltype_syslog=like(col_type,'sys%')")
chart count() by col_type, is_coltype_syslog
```

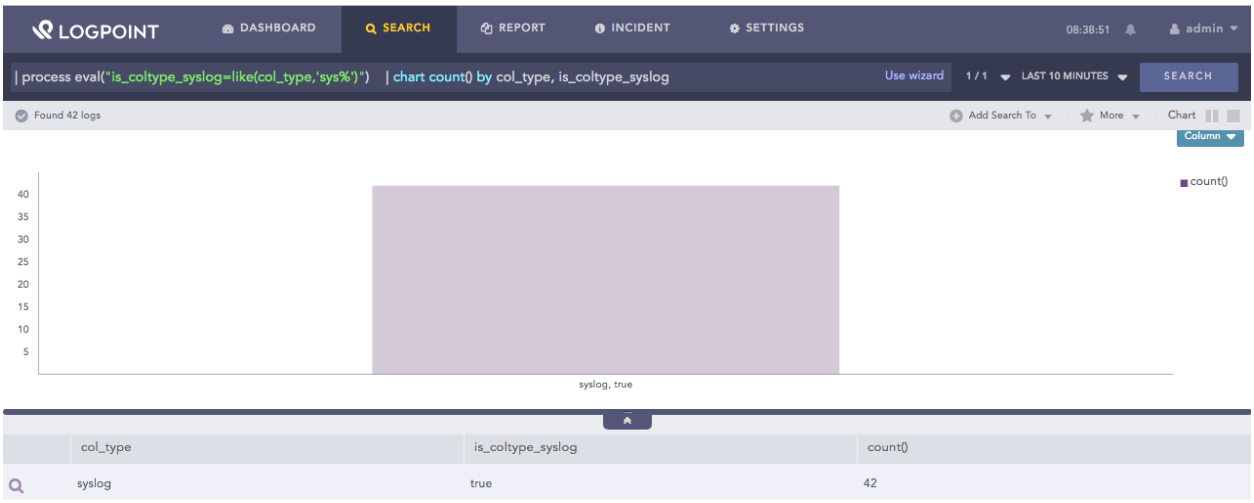


Fig. 10: Using like function

Here, the query compares the **sys%** pattern with the **col_type** field's value. It returns **true** in the **is_coltype_filesystem** identifier if the **sys%** pattern is an exact match or is a substring of the **col_type** field's value, if it doesn't match it returns **false**.

The *chart count()* command displays the count of the combination of **col_type* and **is_coltype_syslog** values as a chart and in a tabular form.

5.11 null

Returns null. You use the null function in combination with other functions. Use this function if you do not want any value returned in the user interface. Unlike other functions, this function does not take any argument.

Syntax:

```
process eval("identifier=null()")
```

Example:

```
process eval("User_severity=if(score <= 5) {return null() } else {return 'Risk user'}")
chart count() by score, User_severity
```

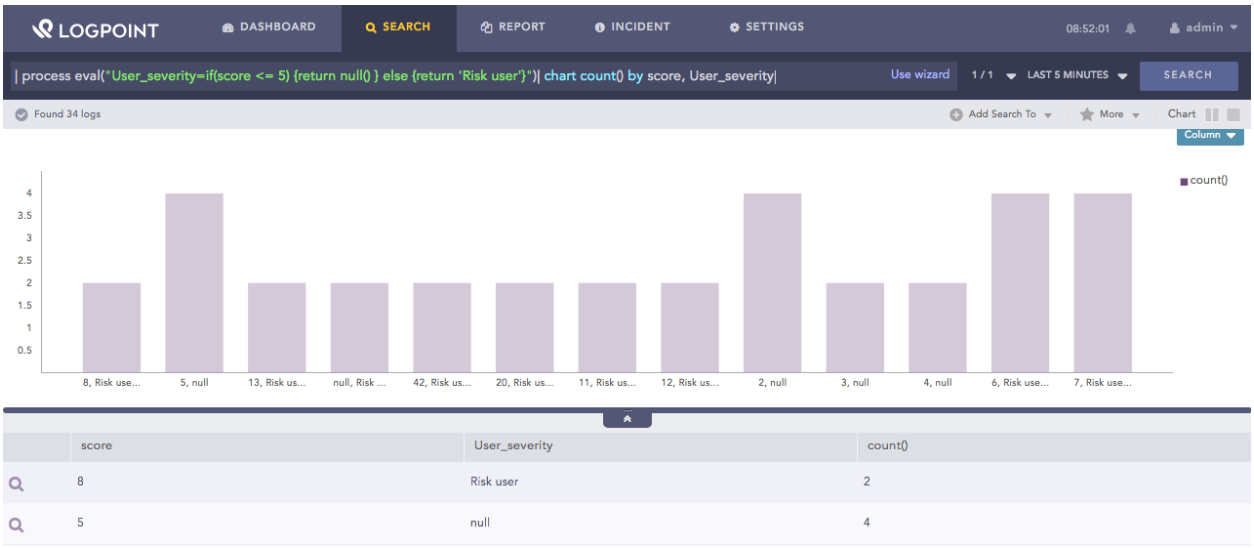


Fig. 11: Using null function

Here, the query returns **null** in the **User_severity** identifier if the **score** field's value is less or equal to 5, if it isn't it returns **Risk user**.

The *chart count()* command displays the count of the combination of **score** and **User_severity** values as a chart and in a tabular form.

5.12 nullif

Compares two arguments: *X* and *Y*. If *X* = *Y*, it returns null, if it isn't equal it returns the value of *X*.

Syntax:

```
| process eval("identifier=nullif(X, Y)")
```

Example:

```
| process eval("access_type=nullif(access,'DELETE')")
| chart count() by access, access_type
```

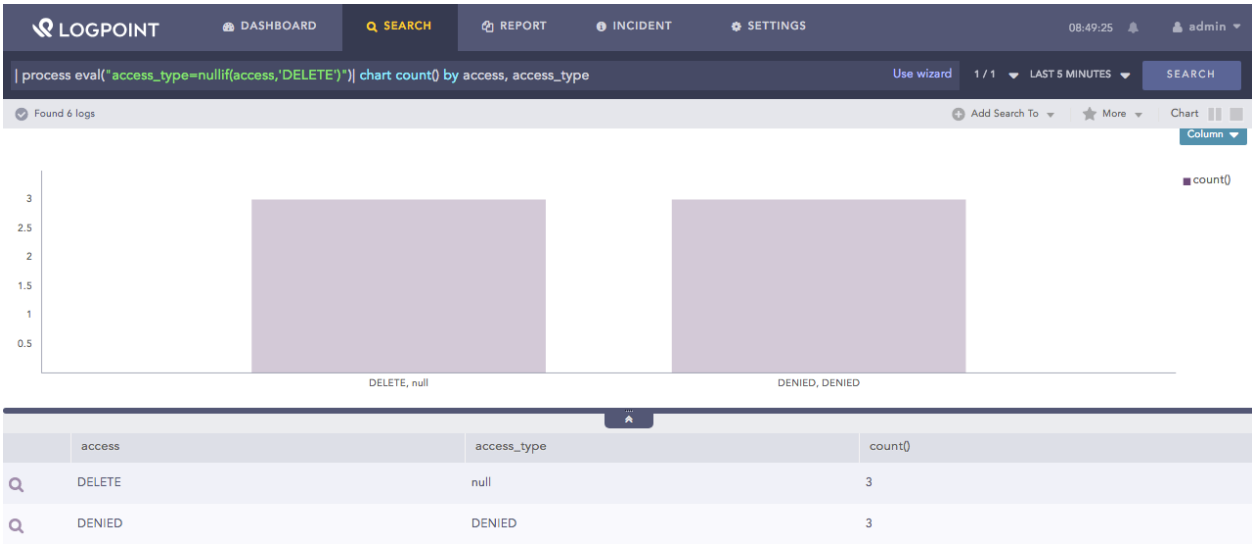


Fig. 12: Using nullif function

Here, the query returns **null** in the **access_type** identifier if the **access** field's value is **DELETE**, if it isn't it returns the value of the **access**.

The *chart count()* command displays the count of the combination of **access** and **access_type** values as a chart and in a tabular form.

5.13 searchmatch

Accepts a string field *X* as input. It returns *true* if the value of *X* matches the event type, if it doesn't match it returns *false*. You can use the pipe (|) symbol to separate multiple values of *X*.

Syntax:

```
| process eval("identifier=searchmatch(X)")
```

Example:

```
| process eval("is_authenticaion_event=searchmatch('Authentication | Access')")
```

Fig. 13: Using searchmatch function

Here, the query returns **null** in the **is_authentication_event** identifier if the **access** field's value is **DELETE**, if it isn't it returns **false**.

5.14 true

Returns *true*. It is often used in combination with other functions to represent a condition that is undoubtedly true, 1==1. Unlike other functions, this function does not take any argument.

Syntax:

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

Example:

```
| process eval("is_profit=if(Selling_price > cost_price) {return true()} else {return false()}")
| chart count() by Selling_price, cost_price, is_profit
```

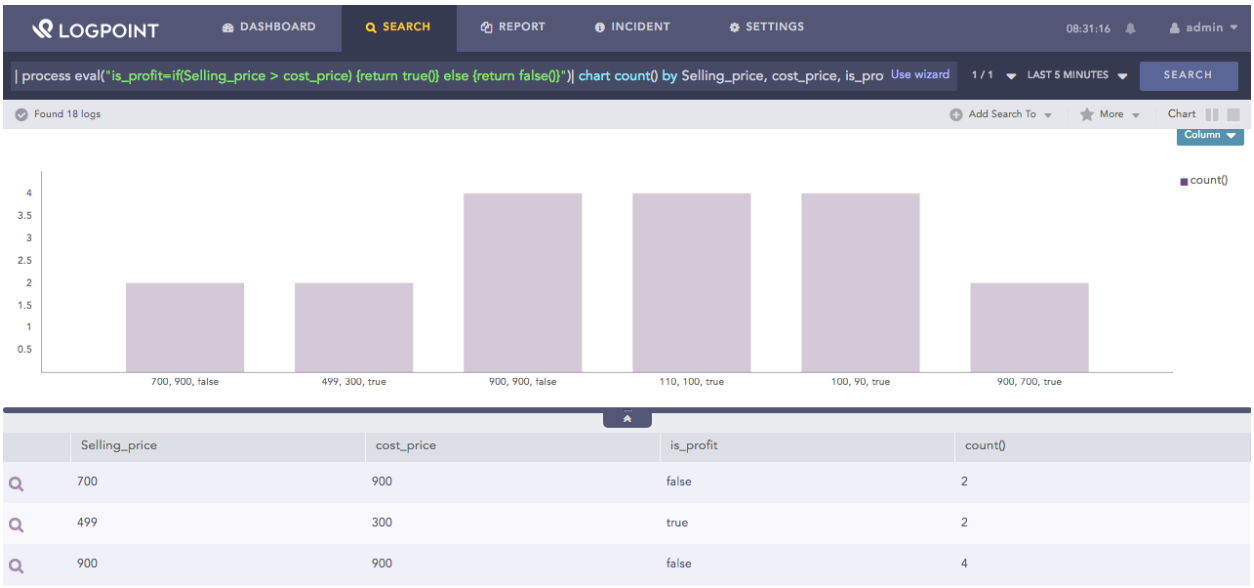


Fig. 14: Uisng true function

Here, the query returns **true** in the **is_profit** identifier if the **Selling_price** field's value is greater than the value of the **cost_price** field. If it isn't, it returns **false**.

The *chart count()* command displays the count of the combination of **Selling_price**, **cost_price** and **is_profit** values as a chart and in a tabular form.

5.15 contains

Accepts three arguments: *X*, *Y* and *Z*. It returns *true* if the value of *X* is within *Y* separated by a delimiter *Z*. If *X* is not listed, the function returns *false*. In the absence of a delimiter, the comma is a default delimiter.

Syntax:

```
| process eval("identifier=contains(X, Y, Z)")
```

Example:

```
| process eval("exists=contains('log','/var/log/syslog', '/') ")
```

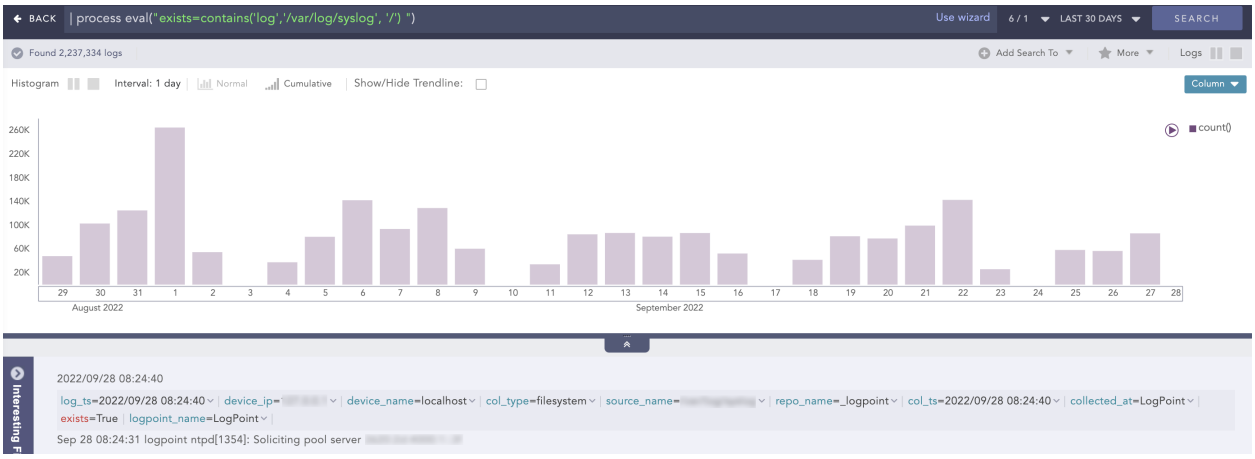


Fig. 15: Using contains function

Here, the query returns **true** in the **exists** identifier if the **log*** string is within **/var/log/syslog** string separated by **/** delimiter. If the **log** string is not listed, the function returns **false**.

5.16 has_any

Checks a string or its sub strings from data with any list of case-insensitive strings. It accepts two arguments: *X* and *Y*. It returns *true* if *X* or substring of *X* is present in *Y*, if it isn't present it returns *false*.

Syntax:

```
| process eval("identifier=contains(X, Y)")
```

X: It is a string.

Y: It is a list.

Example 1:

```
| process eval("result = has_any('hi.exe', '.exe,.dmg')")
```

Fig. 16: Using has_any function

Here, the query searches **hi.exe** string in the **.exe,.dmg** list and return **true** value in **result** field as the substring **.exe** is present in the list.

Example 2:

```
| process eval("result=has_any('This was a language.',kb_list)")
```

Fig. 17: Using has_any function

Here, the query searches **This was a language.** string or its sub string in the **kb_list** and returns **true** value in **result** field.

CHAPTER

SIX

CONVERSION FUNCTIONS

Converts numbers and strings to different formats.

6.1 printf

Accepts a string *format* and *arguments* as input and returns a formatted string value based on these input.

Syntax:

```
| process eval("identifier=printf(format, arguments)")
```

- format: A character string that comprises one or more format conversion specifiers. It must always be within single quotes (' ').
- arguments: Includes one or more string, number or field name.

Example:

```
| process eval("result=printf('Hello %s. Your user score is %d.',user,score)")
| fields user, score, result
```

Fig. 1: Using printf function

Here, the query assigns the value of the **user** field to **%s** and **score** field to **%d** and returns the defined sequence of strings in the **result** identifier.

The *fields* command displays the value of the **user**, **score** and **result** in a tabular form.

6.2 tonumber

Accepts a string value *X* and a *BASE* as input. It converts the string *X* to a number by the specified *BASE*.

Syntax:

```
| process eval("identifier=tonumber(X, BASE)")
```

- *X* can be a field name or a string value.
- The *BASE* defines the base number to convert the string value *X*. It can range from 2 to 36.

Example:

```
| process eval("result=tonumber('0A5',12)")
```

Fig. 2: Using tonumber function

Here, the query converts the string value **0A5** to a number by taking base 12 and returns it in the **result** identifier.

6.3 toint

Accepts an argument *Y* of string, long or double type as input and converts the value of *Y* to integer and assigns the converted value to *X* field.

Syntax:

```
| process eval("X=toint(Y)")
```

Example:

```
| process eval("x=toint(rule_trigger_id)")
```

Fig. 3: Using toint function

Here, the query converts the value of **rule_trigger_id** to integer and assigns the value to **x** field.

6.4 toreal

Accepts an argument *X* of string or long type as input and converts the value of *X* to double and assigns the converted value to *Y* field.

Syntax:

```
| process eval("Y=toreal(X)")
```

Example:

```
| process eval("y=toreal(severity)")
```

Fig. 4: Using toreal function

Here, the query converts the value of **severity** to double type and assigns the value in **y**.

6.5 tostring

Accepts at least one argument *X* as input and converts the input value *X* to a string. If *X* is a number, the second field *Y* can be "hex," "commas," or "duration." *Y* is optional.

Syntax:

```
| process eval("identifier=tostring(X, Y)")
```

- If X is a number, the function converts the number to a string,
- If X is a Boolean value, it returns the corresponding string value, True or False.

Syntax	Description
tostring(X, "hex")	Converts the input value X to hexadecimal.
tostring(X, "commas")	Formats the input value X with commas. If the number includes decimals, it is rounded to the nearest two decimal places.
tostring(X, "duration")	Converts the input value X (in seconds) to the readable time format HH:MM:SS.

Example 1:

```
| process eval("x=tostring(12)")
```

Here, the query converts the numeric value of **12** to string.

Example 2:

```
| process eval("result=tostring(score,'hex')")
```

Fig. 5: Using tostring function

Here, the query converts the numeric value of the **score** field to its corresponding hexadecimal string value and assigns it in the **result** identifier.

Example 3:

```
process eval("x=tostring(65,'duration')")
```

Here, the query converts the numeric value **65** into the readable time format **HH:MM:** and returns it in the **x** identifier.

Example 4:

```
process eval("x=tostring(65132.6789,'commas')")
```

Here, the query formats the numeric value **65132.6789** with a comma and rounds the decimal value to the two decimal place (hundredths position) and returns it in the **x** identifier.

CHAPTER

SEVEN

CRYPTOGRAPHIC FUNCTIONS

Evaluates hash functions and returns a fixed-size alphanumeric string.

7.1 md5

Accepts a string value *X* as input and returns the *md5* hash of the string value. The *md5* is a hash function that generates a 128-bit hash value of the string.

Syntax:

```
| process eval("identifier=md5(X)")
```

Example:

```
| process eval("hash=md5(device_name)")
```

Fig. 1: Using md5 function

Here, the query converts the **device_name** field value into its corresponding **md5** hash value and returns the value **421AA90E079FA326B6494F812AD13E79** in the **hash** identifier.

7.2 sha1

Accepts a string value *X* and returns the *sha1* hash of the string value. The *sha1* is a cryptographic hash function that generates a 160-bit (20-byte) hash value, typically rendered as a hexadecimal number, 40 digits long.

Syntax:

```
| process eval("identifier=sha1(X)")
```

Example:

```
| process eval("sha1_value=sha1(device_name)")
```

Fig. 2: Using sha1 function

Here, the query returns the corresponding **sha1** hash value of *device_name** in the **sha1_value** identifier.

7.3 sha256

Accepts a string value *X* and returns the *sha256* hash of a value. The *sha256* is a cryptographic hash function that generates an almost-unique 256-bit (32-byte) hash value, typically rendered as a hexadecimal number, 64 digits long.

Syntax:

```
| process eval("identifier=sha256(X)")
```

Example 1:

```
| process eval("sha256_value=sha256(device_name)")
```

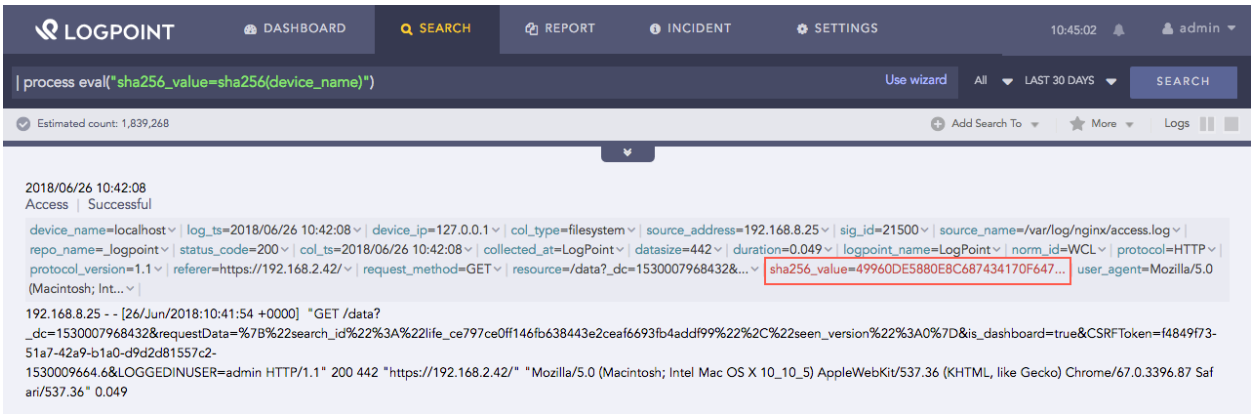


Fig. 3: Using sha256 function

Here, the query returns the corresponding **sha256** hash value of the **device_name** in the **sha256_value** identifier.

7.4 sha512

Accepts a string value *X* and returns the *sha512* hash of a value. The *sha512* is a cryptographic hash function that generates an almost-unique 512-bit (32-byte) hash value, typically rendered as a hexadecimal number, 128 digits long.

Syntax:

```
| process eval("identifier=sha512(X)")
```

Example:

```
| process eval("sha512_value=sha512(device_name)")
```

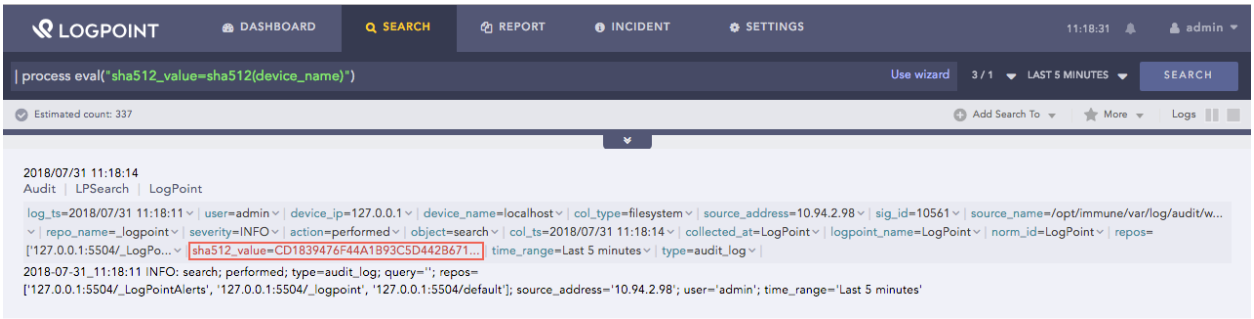


Fig. 4: Using sha512 function

Here, the query returns the corresponding **sha512** hash value of the **device_name** in the **sha512_value** identifier.

CHAPTER

EIGHT

DATE/TIME FUNCTIONS

Evaluates the date and time values in the YYYY/MM/DD format. You can customize the date/time format according the **Date/Time patterns** described in this page.

8.1 now

Returns the UNIX time when the search starts. Unlike other functions, this function does not take any argument.

Syntax:

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

Example:

```
| process eval("search_time=now()")
```

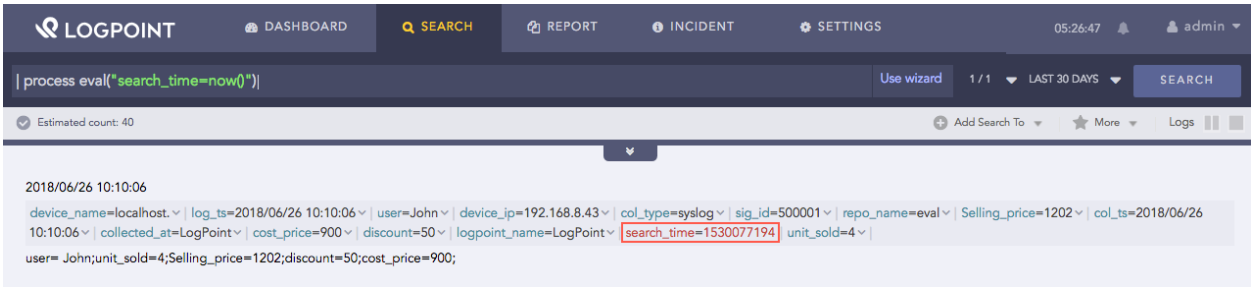


Fig. 1: Using now function

Here, the query returns the UNIX time of the search process in the **search** identifier.

8.2 relative_time

Accepts two arguments: a UNIX time *X* and a relative time specifier *Y* as input, and returns a UNIX time by adding or deducting the value of *Y* from the value of *X*.

Syntax:

```
| process eval("identifier=relative_time(X, Y)")
```

- The operator used in *Y* can be either + or -.
- The format specifier of time is *s* for a second, *m* for a minute, *h* for an hour, *d* for a day and *w* for a week.

Example:

```
| process eval("result=relative_time(now(), '+1d')")
```

Fig. 2: Using relative time function

Here, the query adds time equivalent of 1 day or 24 hours to the current UNIX time and returns it in the **result** identifier.

8.3 strftime

Accepts a UNIX time *X* and returns the time as a string using the date and time format specified in *Y*. The UNIX time value must be in seconds.

Syntax:

```
| process eval("identifier=strftime(X, Y)")
```

Example:

```
| process eval("search_date=strftime(now(), 'YYYY/MM/DD')")
```

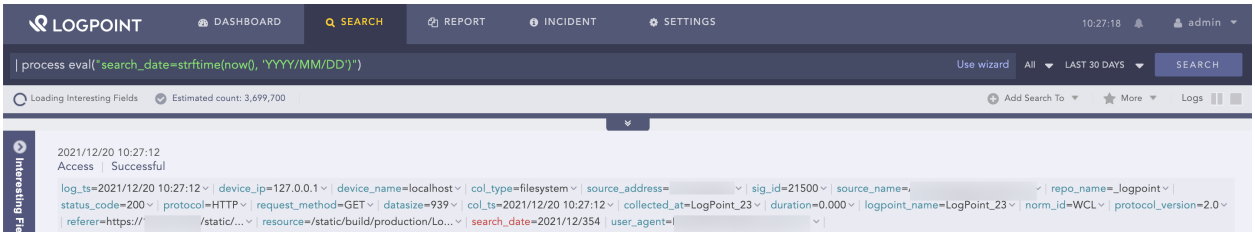


Fig. 3: Using strftime function for day in year

Here, the query accepts the current UNIX time and returns the time in **YYYY/MM/DD** format in the **search_date** identifier.

Example:

```
| process eval("search_date=strftime(now(), 'YYYY/MM/dd')")
```

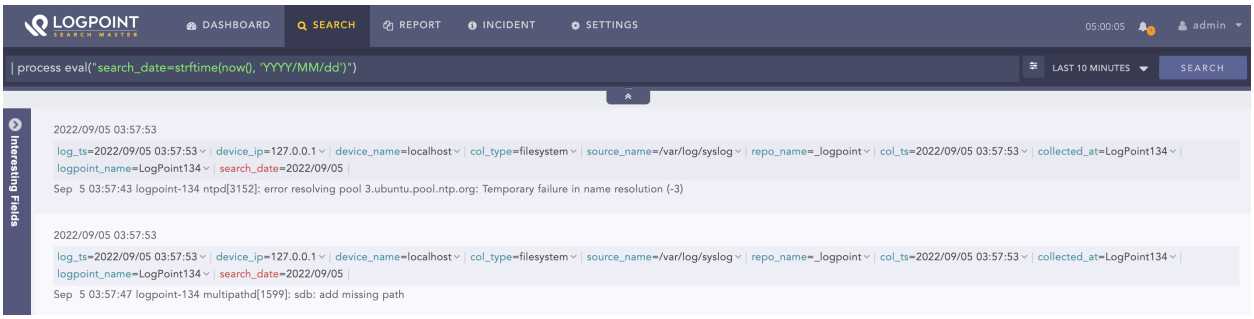


Fig. 4: Using strftime function for day in month

Here, the query accepts the current UNIX time and displays the time in **YYYY/MM/dd** format in the **search_date** identifier.

The **Timezone** parameter in the *strptime* function converts date and time values based on timezone, defined by a fixed offset from Greenwich Mean Time (GMT). By default, the timezone of a machine is used for the conversion. It is an optional parameter.

Syntax:

```
GMTOffsetTimeZone:
        GMT Sign Hours : Minutes
    Sign: one of
        + -
    Hours:
        Digit
        Digit Digit
    Minutes:
        Digit Digit
    Digit: one of
        0 1 2 3 4 5 6 7 8 9
```

Example:

```
| process eval("identifier=strftime(now(), 'yyyy-MM-dd hh:mm:ss', 'GMT+4:45')")
```

8.4 strptime

Accepts a human readable time specified in *X* and converts it into a UNIX timestamp using the date and time format specified in *Y*.

Syntax:

```
| process eval("identifier=strptime(X, Y)")
```

Example:

```
| process eval("searchtime=strptime('2017-12-12', 'yyyy-mm-dd')")
```

Fig. 5: Using strptime function

Here, the query accepts human readable time and converts it to a UNIX timestamp. It returns the converted time in the **searchtime** identifier.

The **Timezone** parameter in the strptime function converts date and time values based on timezone, defined by a fixed offset from Greenwich Mean Time (GMT). By default, the timezone of a machine is used for the conversion. It is an optional parameter.

Syntax:

```
GMTOffsetTimeZone:
        GMT Sign Hours : Minutes
  Sign: one of
        + -
  Hours:
        Digit
        Digit Digit
  Minutes:
        Digit Digit
```

(continues on next page)

(continued from previous page)

Digit: one of
0 1 2 3 4 5 6 7 8 9

Example:

```
| process eval("identifier=strptime('2022-08-12', 'yyyy-MM-dd', 'GMT-9:45')")
```

8.5 time

Takes no argument and returns the UNIX time on which the eval process command processes the command. The returned time is the time when the eval command processed the event.

Syntax:

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

Example:

```
| process eval("process_time=time()")
```

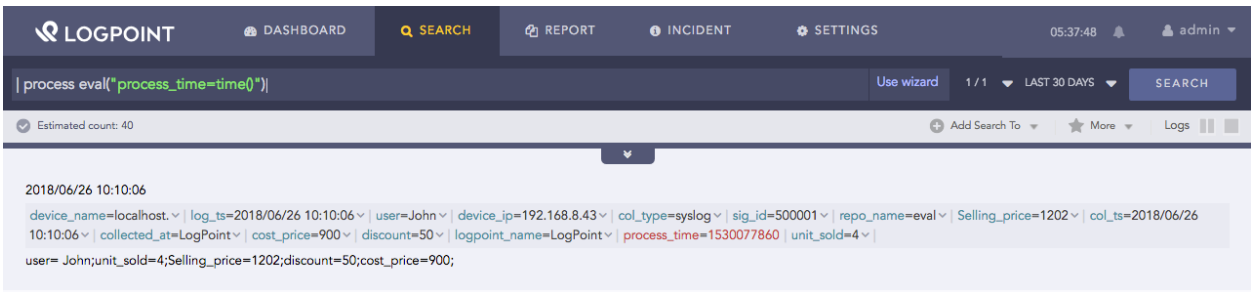


Fig. 6: Using time function

Here, the query returns the time of the eval command execution in the **process_time** identifier.

8.6 Date/Time patterns

The list of all possible patterns in the date/time function.

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, …, 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

CHAPTER

NINE

EXPRESSION WITH PARENTHESES

Parentheses clarify the order of expressions, for example the eval command first evaluates the parts of expression contained within the parentheses.

Example:

```
| process eval("Profit_percent=(Selling_price-cost_price)/cost_price * 100")
| fields Selling_price, cost_price, Profit_percent
```

Fig. 1: Using expression with parentheses

Here, the query calculates the specified arithmetic expression and returns its value in the **Profit_percent** identifier. While performing the calculation, the function first evaluates the part of the expression within the parentheses, **(Selling_price-cost_price)**, then it uses this result in the rest of the expression.

The *fields* command displays the value of **Selling_price**, **cost_price** and **Profit_percent** in a tabular form.

CHAPTER

TEN

INFORMATIONAL FUNCTIONS

Evaluates the information of arguments.

10.1 isbool

Accepts one argument *X* and returns *true* if the value of *X* is Boolean. If the value is not a Boolean, it returns *false*.

Syntax:

```
process eval("identifier=isbool(X)")
```

Example:

```
process eval("is_loss=Selling_price<cost_price")
process eval("boolean_result=isbool(is_loss)")
chart count() by Selling_price, cost_price, is_loss, boolean_result
```

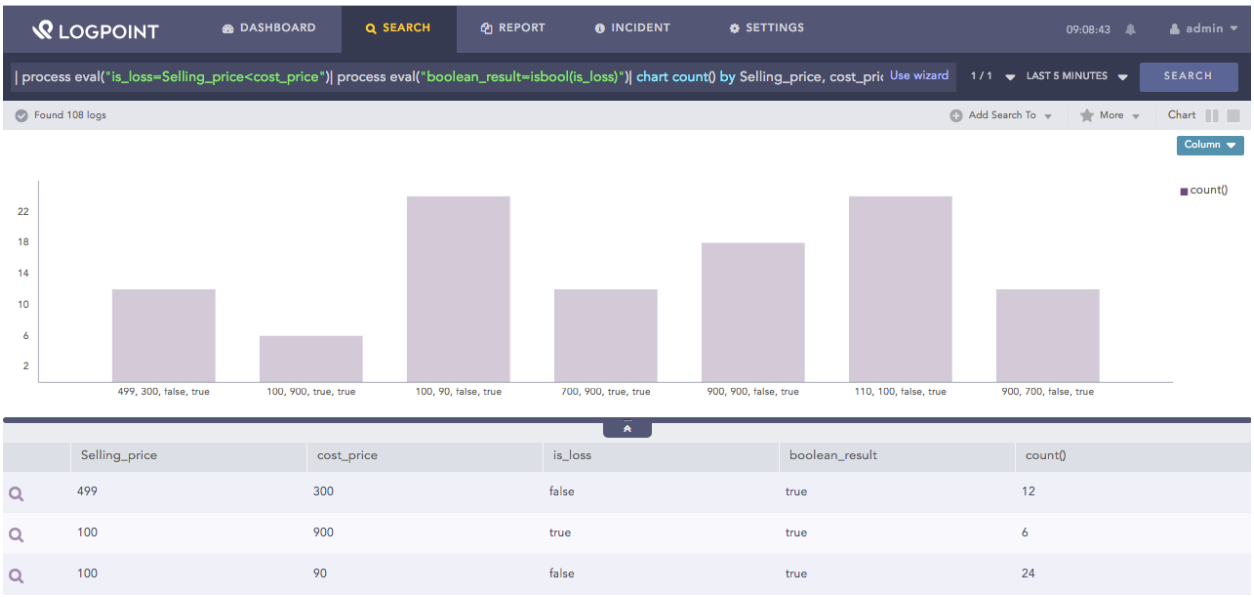


Fig. 1: Using isbool function

Here, the query first evaluates if the **Selling_price** is less than **cost_price** and returns its value in the **is_loss** identifier. Then, it returns **true** in the **boolean_result** identifier if the value in the **is_loss** field is Boolean. If the value is not a Boolean, the function returns **false**.

The *chart count()* command displays the count of the combination of **Selling_price**, **cost_price**, **is_loss** and **boolean_result** values as a chart and in a tabular form.

10.2 isint

Accepts one argument *X* and returns *true* if the value of *X* is an integer. If the value is not an integer, the function returns *false*.

Syntax:

```
| process eval("identifier=isint(X)")
```

Example:

```
| process eval("isscore_int=isint(score)") | fields score, isscore_int
```

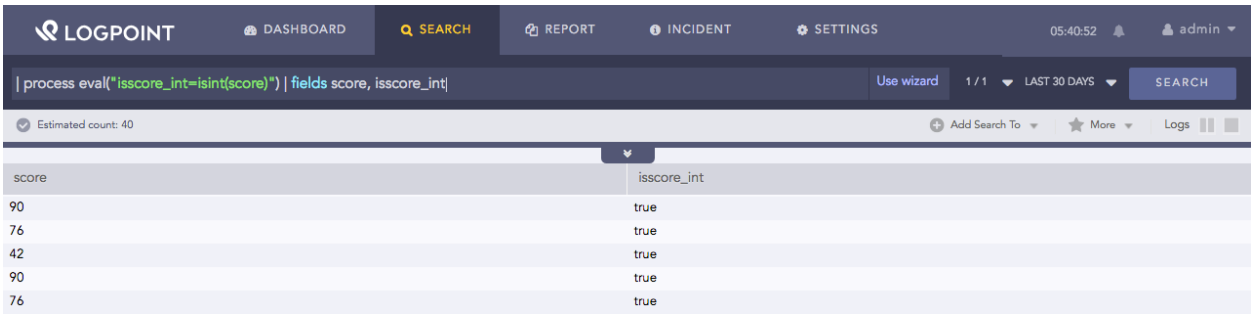


Fig. 2: Using isint function

Here, the query returns **true** in the **isscore_int** identifier if the value in the **score** field is an integer. If the value is not an integer, the function returns **false**.

The *fields* command displays the value of **score** and **isscore_int** in a tabular form.

10.3 isnotnull

Accepts one argument *X* and returns *true* if the value of *X* is not null. If the value is null, the function returns *false*.

Syntax:

```
| process eval("identifier=isnotnull(X)")
```

Example:

```
| process eval("is_loss=Selling_price<cost_price")
| process eval("notnull_result=isnotnull(is_loss)")
| chart count() by Selling_price, cost_price, is_loss, notnull_result
```

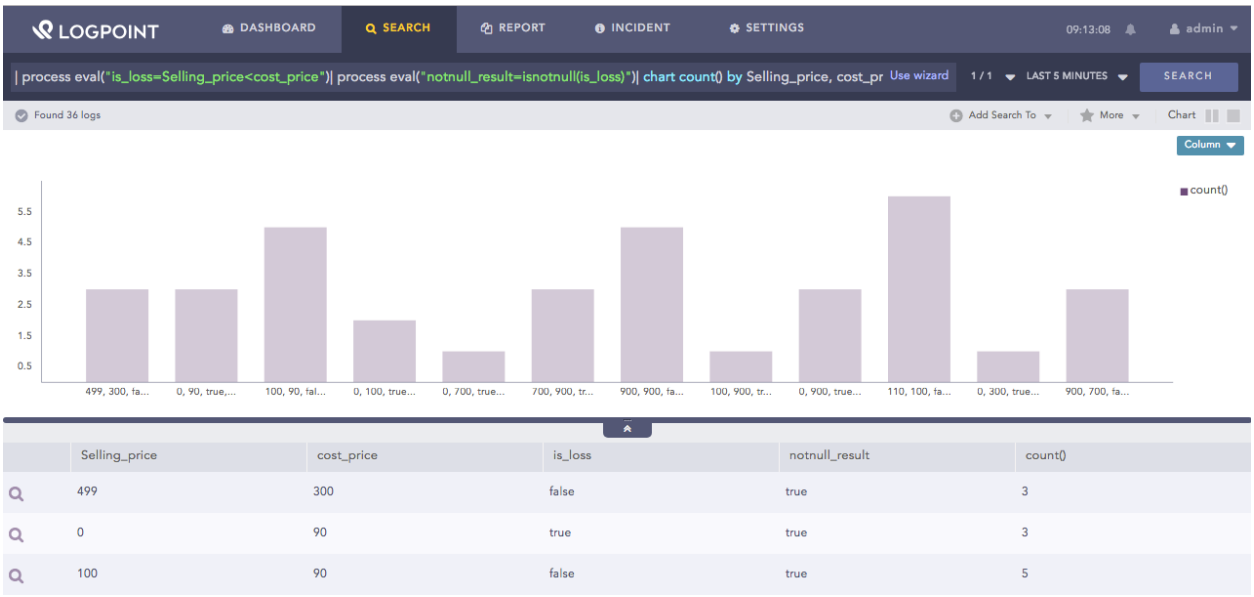


Fig. 3: Using isnotnull function

Here, the query first evaluates if the **Selling_price** is less than **cost_price** and returns its value in the **is_loss** identifier. Then, it returns **true** in the **notnull_result** identifier if the value in the **is_loss** field is not null. If the value is null, the function returns **false**.

The *chart count()* command displays the count of the combination of **Selling_price**, **cost_price**, **is_loss** and **notnull_result** values as a chart and in a tabular form.

10.4 isnull

Accepts one argument *X* and returns *true* if the value of *X* is null. If the value is not null, the function returns *false*.

Syntax:

```
| process eval("identifier=isnull(X)")
```

Example:

```
| process eval("is_loss=Selling_price<cost_price")
| process eval("null_result=isnull(is_loss)")
| chart count() by Selling_price, cost_price, is_loss, null_result
```

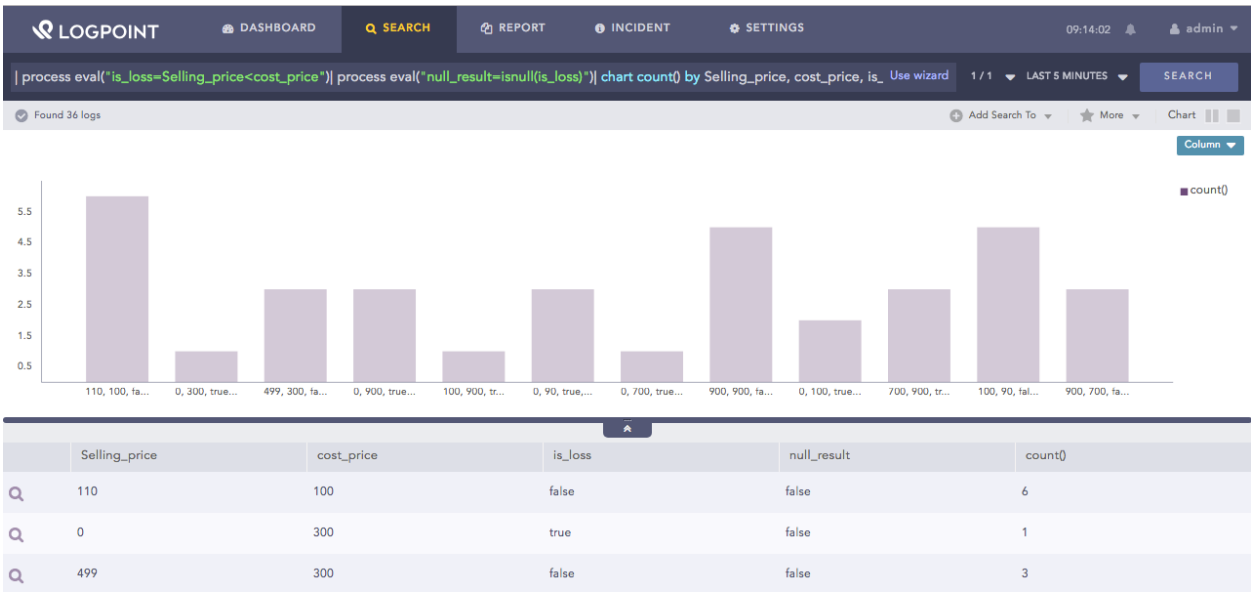


Fig. 4: Using isnull function

Here, the query first evaluates if the **Selling_price** is less than **cost_price** and returns its value in the **is_loss** identifier. Then, it returns **true** in the **null_result** identifier if the value in the **is_loss** field is null. If the value is not null, the function returns **false**.

The *chart count()* command displays the count of the combination of **Selling_price**, **cost_price**, **is_loss** and **null_result** values as a chart and in a tabular form.

10.5 isnum

Accepts one argument *X* and returns *true* if the value of *X* is a number. If the value is not a number, the function returns *false*.

Syntax:

```
| process eval("identifier=isnum(X)")
```

Example:

```
| process eval("num_result=isnum(cost_price)") | chart count() by cost_price, num_result
```

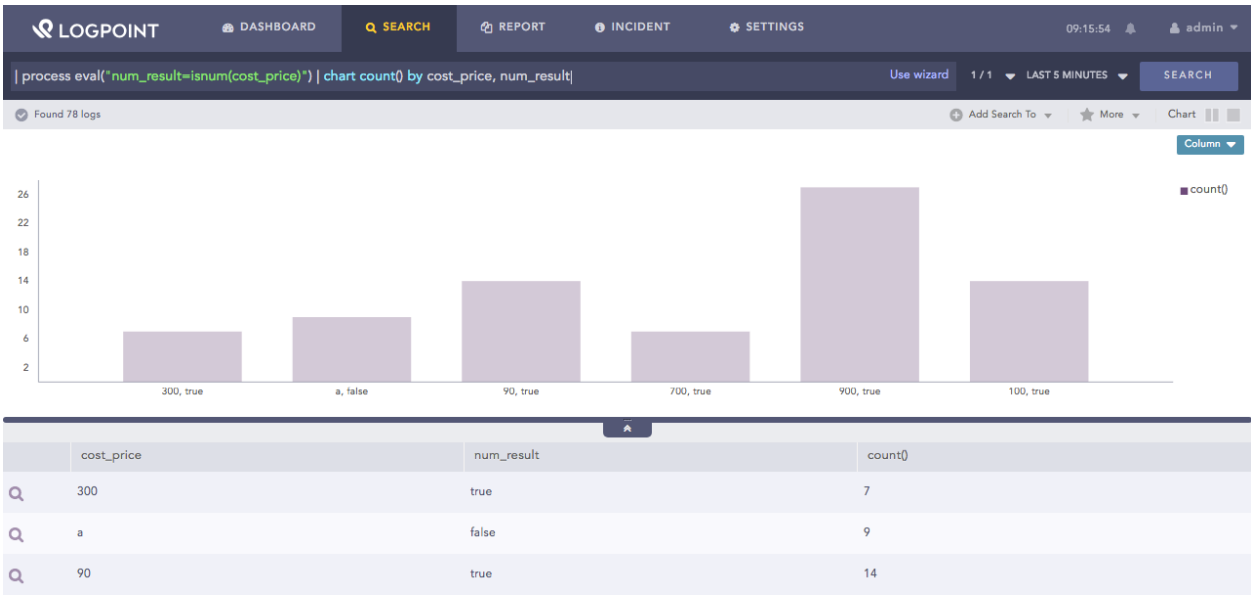


Fig. 5: Using isnum function

Here, the query returns **true** in the **num_result** identifier if the value in the **score** field is a number. If the value is not a number, the function returns **false**.

The *chart count()* command displays the count of the combination of **cost_price** and **num_result** values as a chart and in a tabular form.

10.6 isstr

Accepts one argument *X* and returns *true* if the value of *X* is a string. If the value is not a string, the function returns *false*.

Syntax:

```
| process eval("identifier=isstr(X)")
```

Example:

```
| process eval("str_result=isstr(cost_price)") | chart count() by cost_price, str_result
```

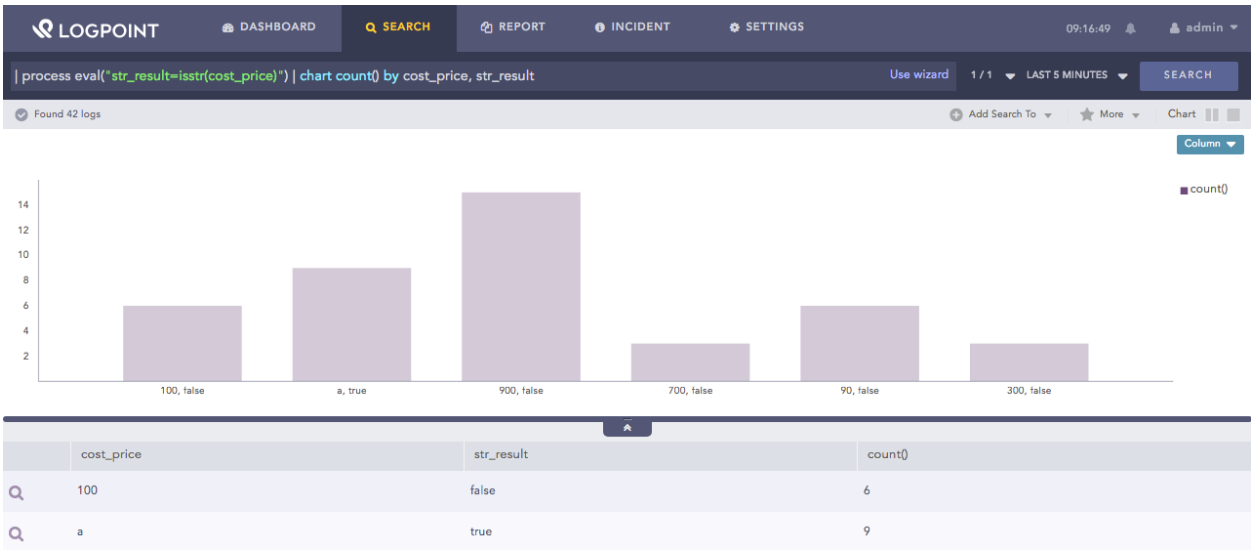


Fig. 6: Using isstr function

Here, the query returns **true** in the **str_result** identifier if the value in the **cost_price** field is a string. If the value is not a string, the function returns **false**.

The *chart count()* command displays the count of the combination of **cost_price** and **str_result** values as a chart and in a tabular form.

10.7 typeof

Accepts one argument *X* and returns the field type of the value of *X*, such as integer, double, string and boolean.

Syntax:

```
| process eval("identifier=typeof(X)")
```

Example:

```
| process eval("event_type=typeof(event_id)") | fields event_id, event_type
```

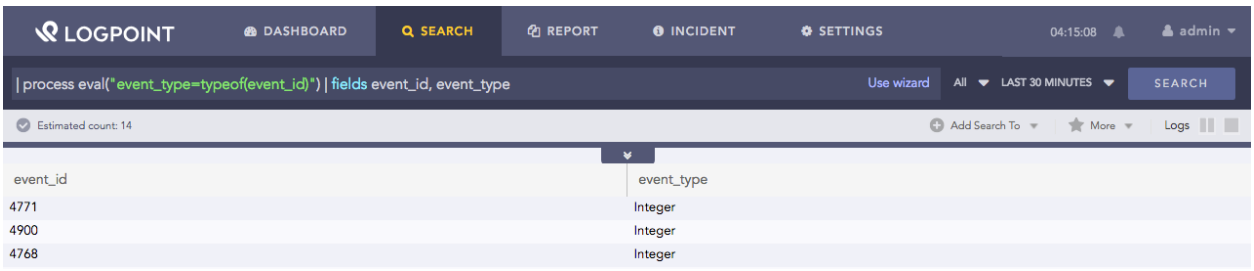


Fig. 7: Using typeof function

Here, the query returns the field type of the **event_id** value in the **event_type** identifier.

The *fields* command displays the value of **event_id** and **event_type** in a tabular form.

10.8 issubstr

Accepts two arguments *X* and *Y* as input and returns *true* if *X* is a substring of *Y*. If *X* is not a substring, the function returns *false*.

Syntax:

```
| process eval("identifier=issubstr(X, Y)")
```

Example:

```
| process eval("exists=issubstr('mal.exe','hi.exmal.exe,ok.dm') ")
```

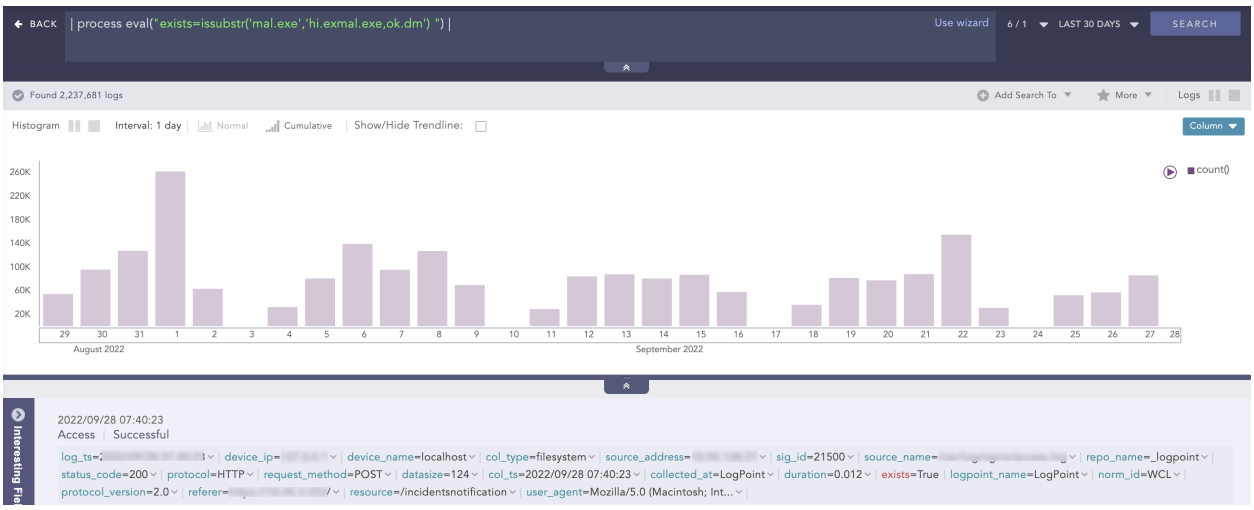


Fig. 8: Using issubstr function

Here, the query returns **true** in the **exists** identifier if **mal.exe** is substring of **hi.exmal.exe,ok.dm**. If **mal.exe** is not a substring, the function returns **false**.

CHAPTER

ELEVEN

LOGICAL EXPRESSIONS

Compares two boolean values and returns either *True* or *False*.

Generic syntax:

```
| process eval("identifier = first_operand logical_operator second_operand")
```

11.1 AND (&&)

Compares two boolean values. It returns *true* if both the values are true. If it isn't, it returns *false*.

Example:

```
| process eval("is_profit=Selling_price>cost_price") | process eval("is_more_discount=discount>
↪=51")
| process eval("is_more_profit=is_profit && is_more_discount")
| chart count() by is_profit, is_more_discount, is_more_profit
```

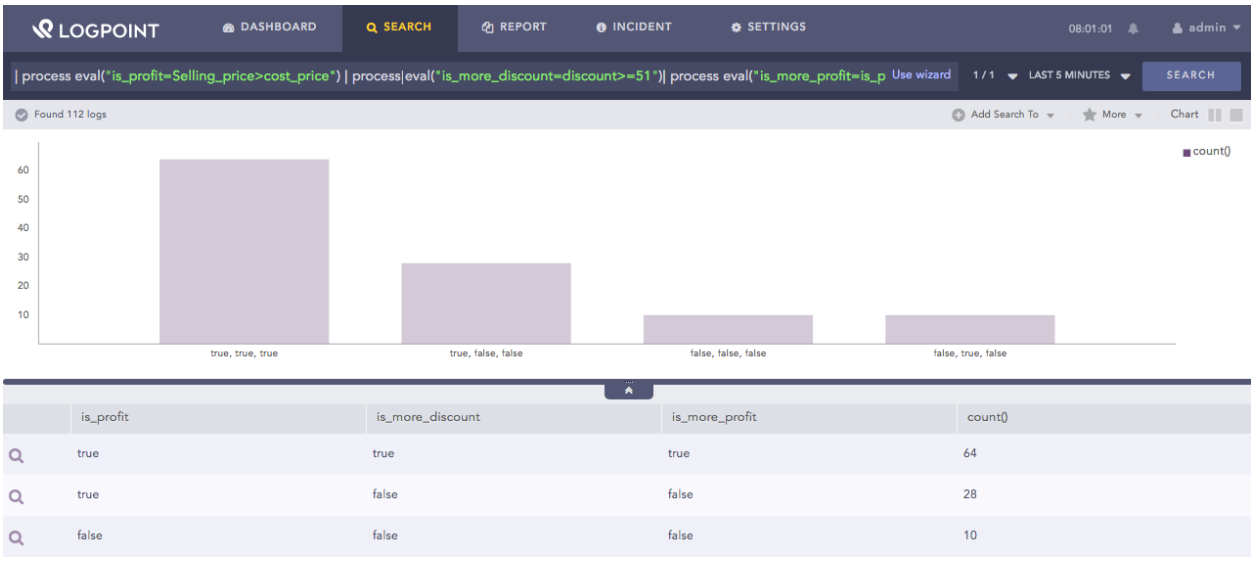


Fig. 1: Using AND function

Here, the query first returns either **true** or **false** in the **is_profit** identifier by comparing whether the value of the **Selling_price** field is greater than the value of the **cost_price** field. It then returns **true** or **false** in the **is_more_discount** identifier by comparing if the value of the **discount** field is greater than or equal to 51. Then it compares the values of **is_profit** and **is_more_discount** fields. It returns **true** in the **is_more_profit** identifier if both the values are true. If it isn't, it returns **false**.

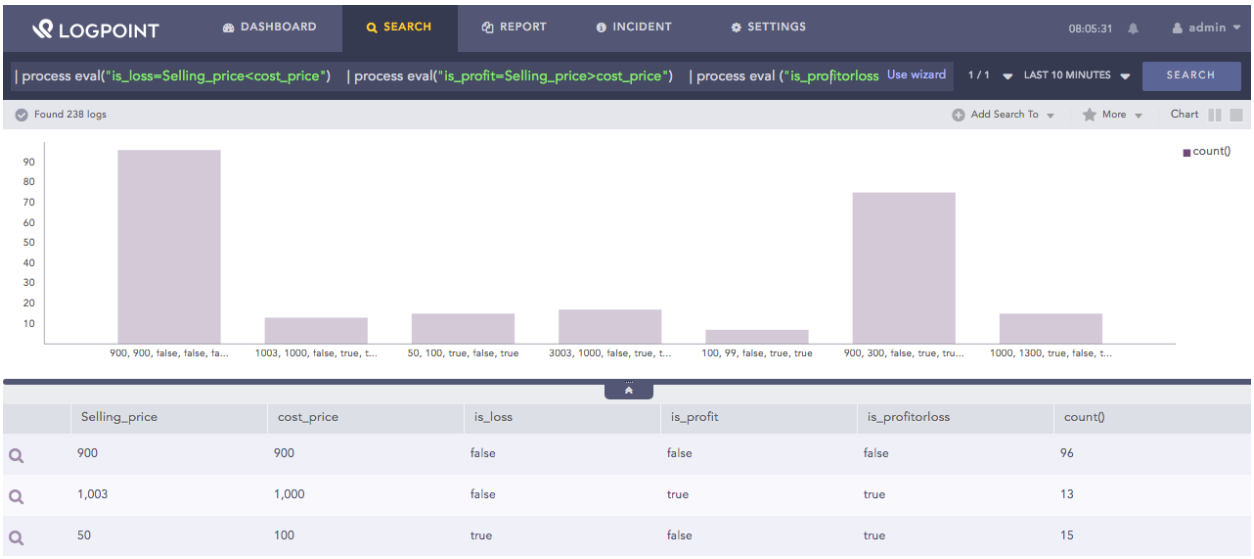
The *chart count()* command displays the count of the combination of **is_profit**, **is_more_discount**, and **is_more_profit** values as a chart and in a tabular form.

11.2 OR (||)

Compares two boolean values. It returns *true* if either value is true. If none of the values are true, it returns *false*.

Example:

```
| process eval("is_loss=Selling_price<cost_price")
| process eval("is_profit=Selling_price>cost_price")
| process eval ("is_profitorloss = is_loss || is_profit ")
| chart count() by Selling_price, cost_price, is_loss, is_profit, is_profitorloss
```



	Selling_price	cost_price	is_loss	is_profit	is_profitorloss	count()
	900	900	false	false	false	96
	1,003	1,000	false	true	true	13
	50	100	true	false	true	15

Fig. 2: Using OR function

Here, the query first returns **true** or **false** in the **is_loss** identifier by comparing if the value of the **Selling_price** field is less than the value of the **cost_price** field. It then returns **true** or **false** in the **is_profit** identifier by comparing if the value of the **Selling_price** field is greater than the value of the **cost_price** field. Then, it compares the values of **is_loss** and **is_profit** fields. It returns **true** in the **is_profitorloss** identifier if either value is **true**. If it isn't, it returns **false**.

The *chart count()* command displays the count of the combination of **Selling_price**, **cost_price**, **is_loss**, **is_profit** and **is_profitorloss** values as a chart and in a tabular form.

CHAPTER

TWELVE

MATHEMATICAL FUNCTIONS

Evaluates mathematical data.

12.1 abs

Accepts a numerical value *X* as input and returns the absolute value of the number as the output.

Syntax:

```
| process eval("identifier=abs(X)")
```

Example:

```
| process eval ("Profit= Selling_price - cost_price")
| process eval ("abs_value=abs(Profit)")
| fields Selling_price, cost_price, Profit, abs_value
```

Fig. 1: Using abs function

Here, the query first calculates the **Profit** field value. It then computes the absolute value of the **Profit** and returns its value in the **abs_value** identifier.

The *fields* command displays the value of **Selling_price**, **cost_price**, **Profit** and **abs_value** in a tabular form.

12.2 floor

Accepts a numerical value *X* and returns the greatest integer less than or equal to *X*.

Syntax:

```
process eval("identifier=floor(X)")
```

Example:

```
process eval("price_per_unit=Selling_price/unit_sold")
process eval("final_price = floor(price_per_unit)")
fields Selling_price, unit_sold, price_per_unit, final_price
```

Selling_price	unit_sold	price_per_unit	final_price
1202	4	300.5	300
1400	14	100	100
1207	2	603.5	603
1508	20	75.4	75
1205	6	200.83	200

Fig. 2: Floor function

Here, the query first calculates the **price_per_unit** field value. It then computes the greatest integer less than or equal to the value of **Profit** and returns its value in the **final_price** identifier.

The *fields* command displays the value of **Selling_price**, **cost_price**, **Profit** and **abs_value** in a tabular form.

12.3 ceiling

Accepts a numerical value *X* and rounds the number up to the smallest following integer value.

Syntax:

```
process eval("identifier=ceiling(X)")
```

Example:

```
process eval("ceiling_duration=ceiling(duration)")
```

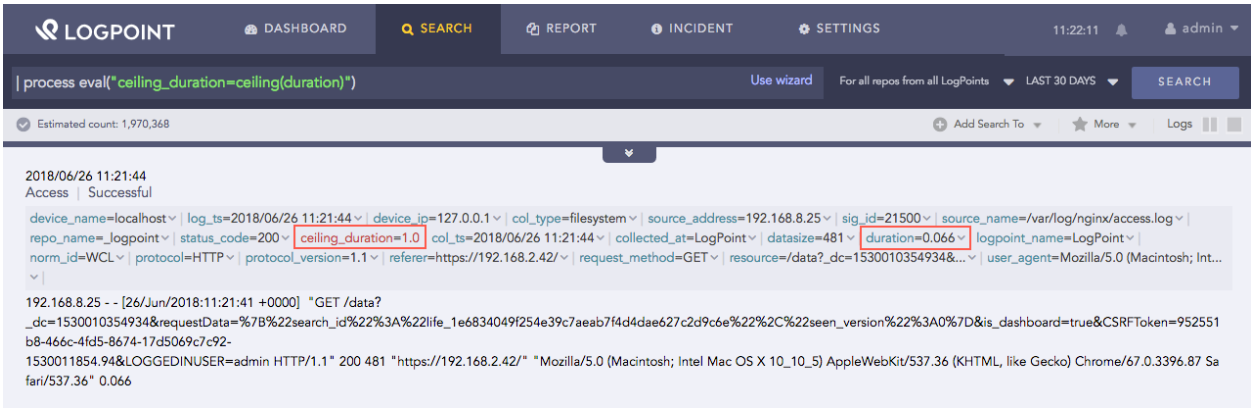


Fig. 3: Using ceiling function

Here, the query accepts the **duration** field value and returns the smallest following integer value in the **ceiling_duration** identifier.

12.4 exp

Accepts a numerical value *X* and evaluates the exponentiation with *e* base and *X* as the exponent, (e^X). You can also use *expe* instead of *exp*.

Syntax:

```
| process eval("identifier=exp(X)")
or
| process eval("identifier=expe(X)")
```

Example:

```
| process eval("result=exp(discount)") | fields discount, result
or
| process eval("result=expe(unit_sold)") | fields unit_sold, result
```

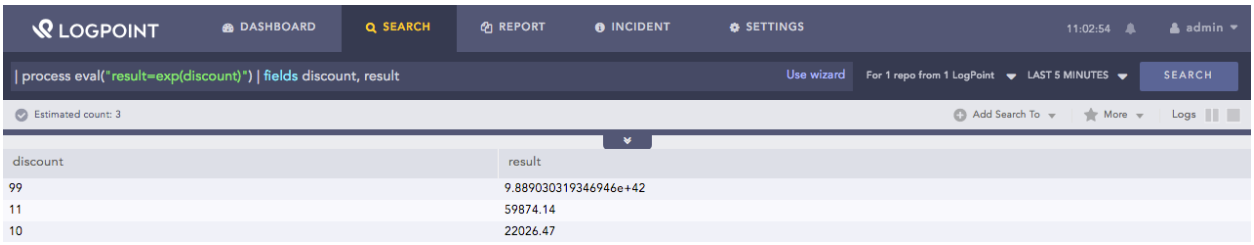


Fig. 4: Using exp function

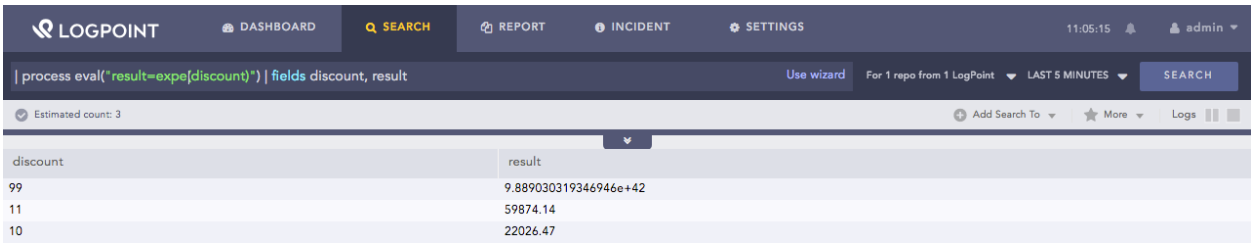


Fig. 5: Using exp function

Here, the query evaluates the exponentiation to the *e* base of the **discount** field and returns it in the **result** identifier.

The *fields* command displays the value of **discount** and **result** in a tabular form.

12.5 exp2

Accepts a numerical value *X* and evaluates the exponentiation with *2* base and *X* as the exponent, (2^X).

Syntax:

```
| process eval("identifier=exp2(X)")
```

Example:

```
| process eval("result=exp2(unit_sold)") | fields unit_sold, result
```

Fig. 6: Using exp2 function

Here, the query evaluates the exponentiation to the *2* base of the **unit_sold** field and returns it in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.6 exp10

Accepts a numerical value *X* and evaluates the exponentiation with *10* base and *X* as the exponent, (10^X).

Syntax:

```
| process eval("identifier=exp10(X)")
```

Example:

```
| process eval("result=exp10(unit_sold)") | fields unit_sold, result
```

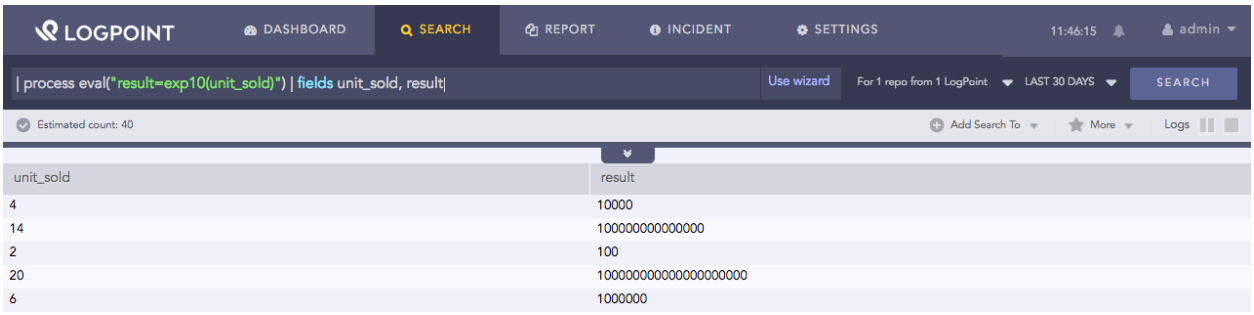


Fig. 7: Using exp10 function

Here, the query evaluates the exponentiation to the *10* base of the **unit_sold** field and returns it in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.7 log

Accepts a numerical value *X* and evaluates the logarithm of *X* with base *e*, (log_e(X)). You can also use the function loge instead of the function log.

Syntax:

```
| process eval("identifier=log(X)")
or
| process eval("identifier=loge(X)")
```

Example:

```
| process eval("result=log(unit_sold)") | fields unit_sold, result
or
| process eval("result=loge(unit_sold)") | fields unit_sold, result
```

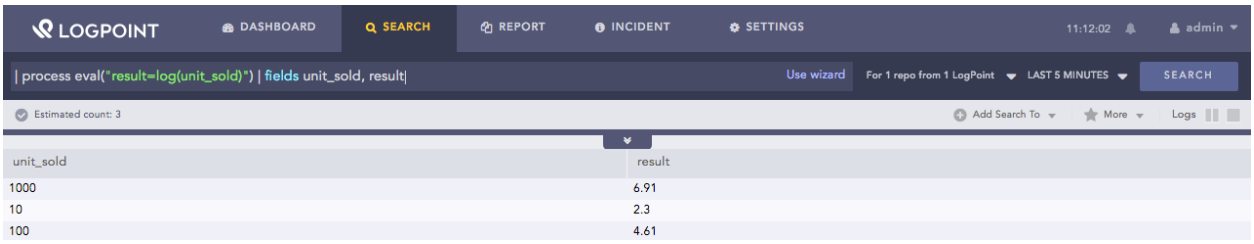


Fig. 8: Using log function

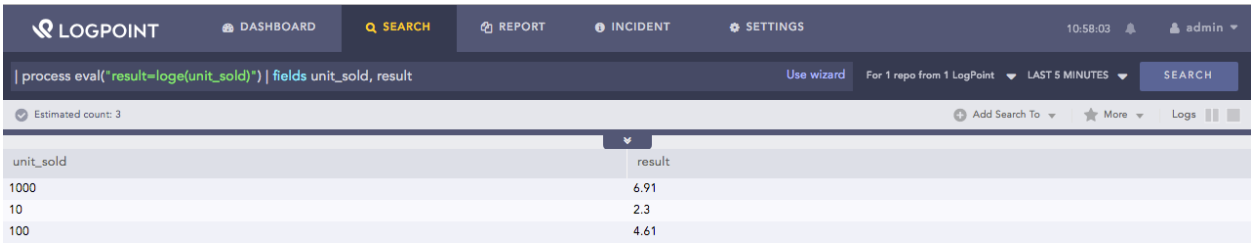


Fig. 9: Using log function

Here, the query evaluates the logarithm to the *e* base of the **unit_sold** field and returns it in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.8 log2

Accepts a numerical value *X* and evaluates the logarithm of *X* with base *2*, (log_2(X)).

Syntax:

```
| process eval("identifier=log2(X)")
```

Example:

```
| process eval("result=log2(unit_sold)") | fields unit_sold, result
```

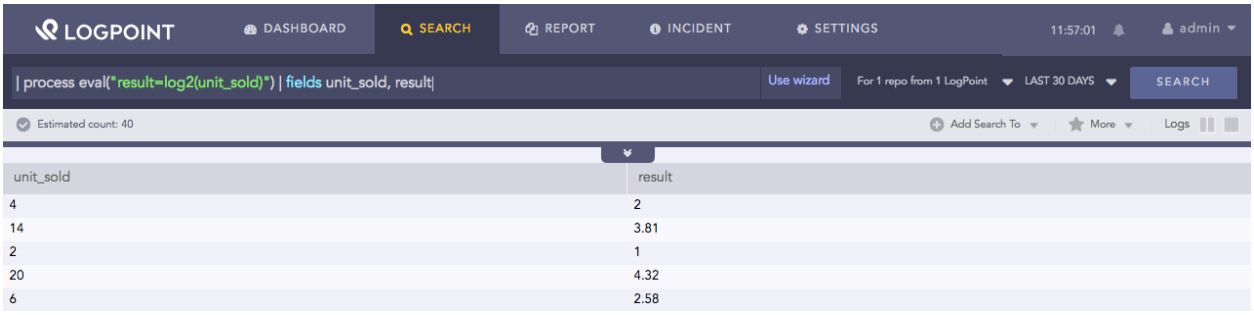


Fig. 10: Using log2 function

Here, the query evaluates the logarithm to the *2* base of the **unit_sold** field and returns it in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.9 log10

Accepts a numerical value *X* and evaluates the logarithm of *X* with base *10*, (log_{mathrm{10}} (X)).

Syntax:

```
| process eval("identifier=log10(X)")
```

Example:

```
| process eval("result=log10(unit_sold)") | fields unit_sold, result
```

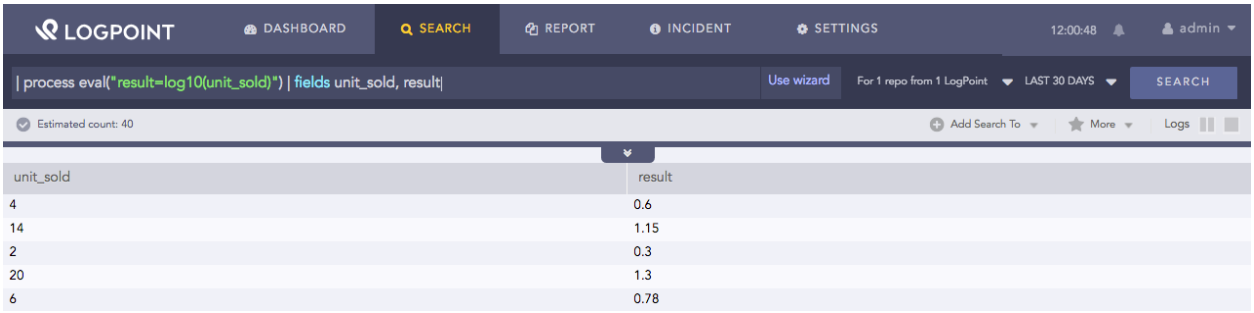


Fig. 11: Using log10 function

Here, the query evaluates the logarithm to the *10* base of the **unit_sold** field and returns it in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.10 pi

Returns the first 12 digits of the value of pi. Unlike other functions, this function does not take any argument.

Syntax:

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

Example:

```
| process eval ("area_circle=pi() * (radius^2)")
| chart count () by radius, area_circle
```

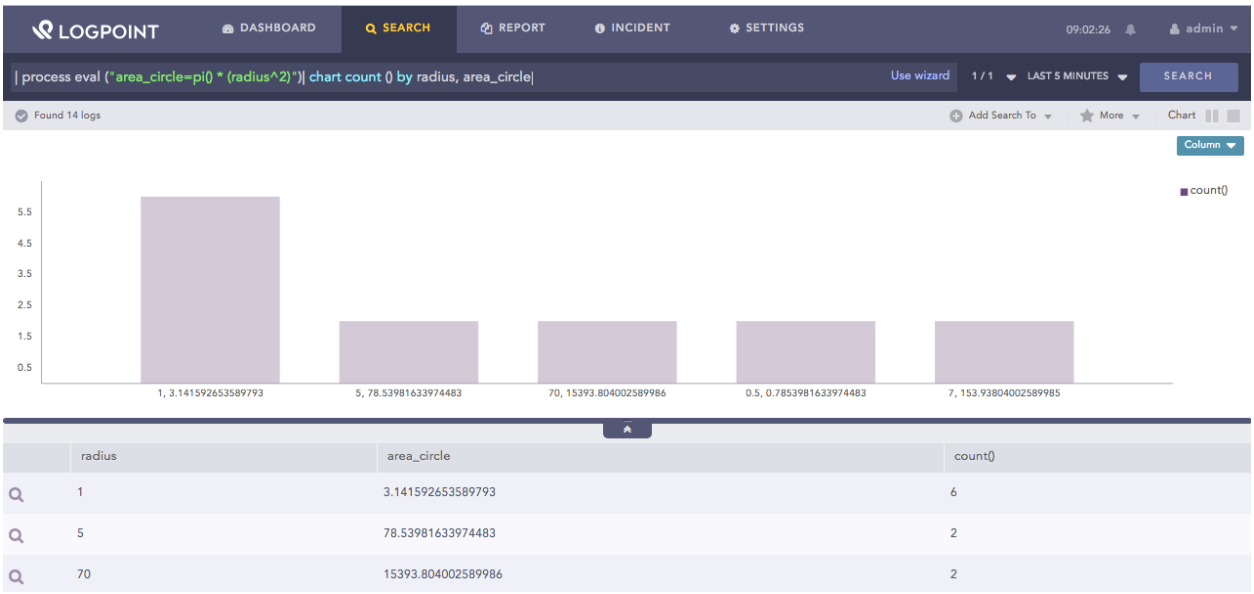


Fig. 12: Using pi function

Here, the query calculates the area of the circle where **pi()** gives the value of the mathematical constant (**π**). The query returns area in the **area_circle** identifier.

The *chart count()* command displays the count of the combination of **radius** and **area_circle** values as a chart and in a tabular form.

12.11 sqrt

Accepts a numeric *X* value and returns the square root of the numeric value.

Syntax:

```
| process eval("identifier=sqrt(X)")
```

Example:

```
| process eval("result=sqrt(unit_sold)") | fields unit_sold, result
```

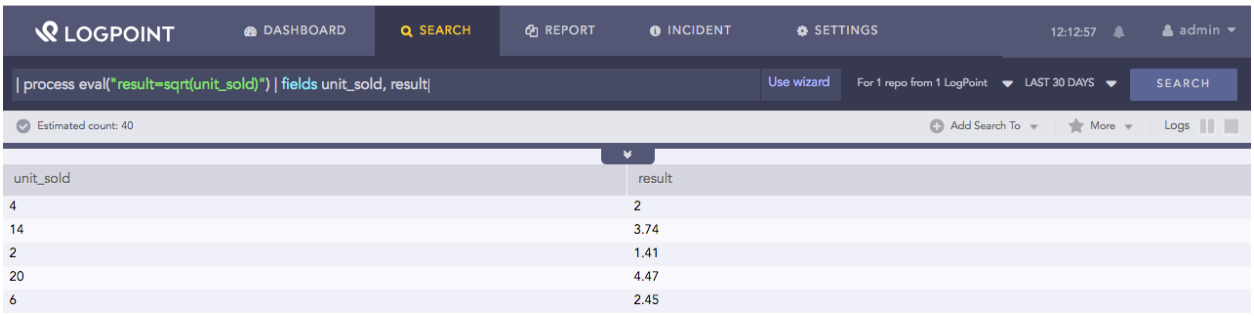


Fig. 13: Using sqrt function

Here, the query returns the square root of the **unit_sold** field value in the **unit_sold** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.12 random

Returns a random number ranging between 0 and 1. It does not take any argument. You can use this function in case you want a random number for any eval expression.

Syntax:

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

Example:

```
| process eval("x=random()")
```

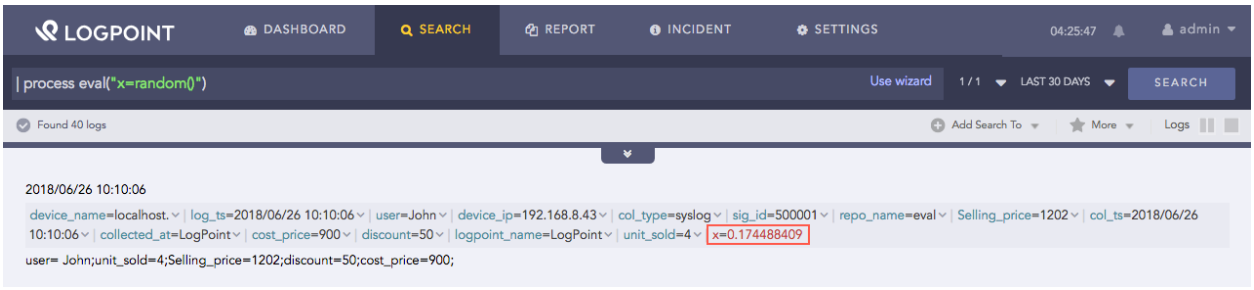


Fig. 14: Using random function

Here, the query returns a random number between 0 and 1 in the **x** identifier.

12.13 exact

Accepts a numeric calculation *X* and returns a result with a significant amount of precision.

Syntax:

```
| process eval("identifier=exact(X)")
```

Example:

```
| process eval("result=exact(3.4*unit_sold)") | fields unit_sold, result
```

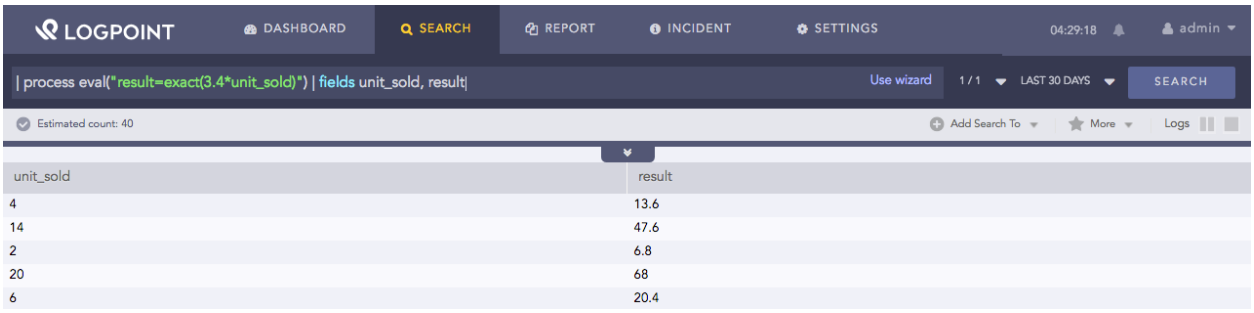


Fig. 15: Using exact function

Here, the query returns the precise value of the arithmetic expression **3.4*unit_sold** in the **result** identifier.

The *fields* command displays the value of **unit_sold** and **result** in a tabular form.

12.14 round

Accepts up to two numeric arguments: *X* and *Y*, and rounds the value specified in *X* by the amount of decimal specified in *Y*. Here *Y* is optional, and in case *Y* is not defined, it rounds the value of *X* to the nearest integer by default.

Syntax:

```
| process eval("identifier=round(X,Y)")
```

Example 1:

```
| process eval("x=round(12.233)")

Result: x=12
```

Example 2:

```
| process eval("result=round(12.234,2)")
```

Fig. 16: Using round function

Here, the query rounds up **12.234** to the hundredths (second decimal place) and returns its value in the **result** identifier.

12.15 sigfig

Accepts one numeric field *X* and rounds that number to the appropriate number of significant figures. It ignores the decimal numbers if provided and takes only the numbers before the decimal point. It does not round the number that is in the 10th and 100th place.

Syntax:

```
| process eval("identifier=sigfig(X)")
```

If *X* is of 1000th place, the function rounds it to the nearest 10.

Example 1:

```
| process eval("result=sigfig(1111)")
```

Fig. 17: Using sigfig function

Here, the query rounds up **1111** to the nearest 10 and returns its value in the **result** identifier.

If *X* is of 10000th place, the function rounds it to the nearest 100.

Example 2:

```
process eval("x=sigfig(11111)")
```

Here, the query rounds up **11111** to the nearest 100 and returns its value in the **result** identifier.

CHAPTER

THIRTEEN

MULTIVALUE FUNCTIONS

Evaluates multivalue arguments and returns multivalue fields.

13.1 split

Accepts two arguments: *X* and *Y*. It splits the *X* field values by the delimiter, such as comma, semicolon and blank space specified in the *Y* field and returns a multivalue field containing a list of split values.

Syntax:

```
| process eval("identifier=split(X,Y)")
```

Example:

```
| process eval("split_message=split(message, ' ')")
| fields message, split_message
```

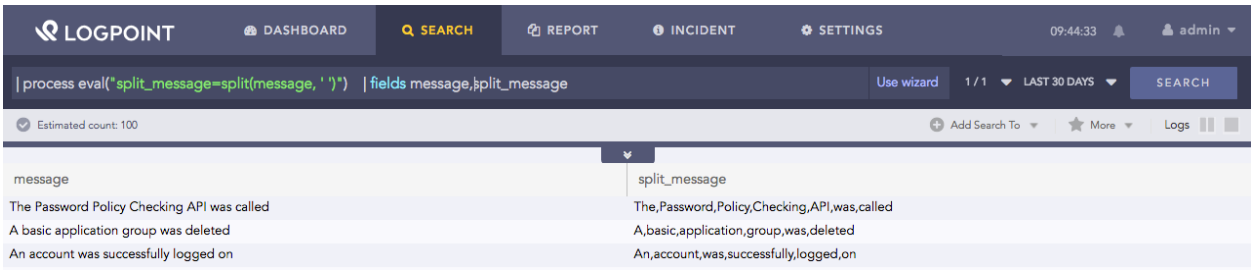


Fig. 1: Using split function

Here, the query splits the **message** field's value when the **split** function detects a space in the value and returns split values in the **split_message** identifier.

The *fields* command displays the value of **fields** and **split_message** in a tabular form.

When using special characters such as backslash '\' as a delimiter, they should be escaped.

Example:

```
| process eval("split_message=split(fieldname,'\\')")
| fields message, split_message
```

13.2 commands

Accepts a search string *X* or a field that contains a search string, and returns a multivalue field containing a list of the commands used in *X*.

Syntax:

```
| process eval("identifier=commands(X)")
```

Example:

```
| process eval("commands_list=commands('chart count() | fields name | rename message as alert
↪')")
| fields commands_list
```

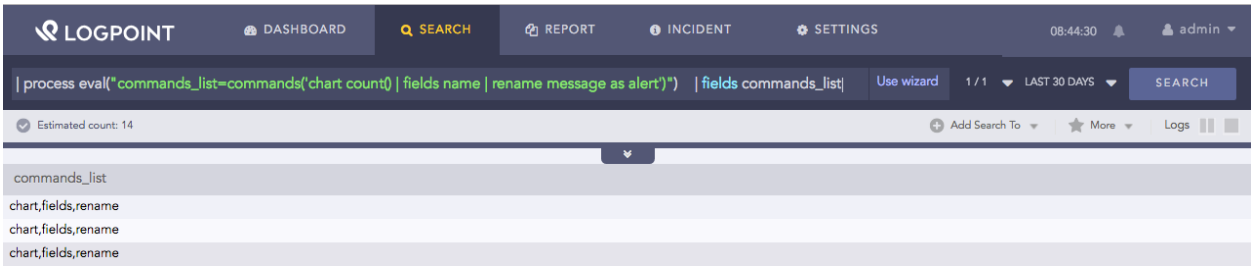


Fig. 2: Using commands function

Here, the query first returns the list of commands in the **chart count() | fields name | rename message as alert** search string. Then, the query using the *commands* returns them in the **commands_list** identifier.

The *fields* command displays the value of **commans_list** in a tabular form.

13.3 mvappend

Accepts two or more arguments and appends the values of the arguments. It returns a multivalue result containing a list of all the appended values. The arguments can be strings, multivalue fields or single value fields.

Syntax:

```
| process eval("identifier=mvappend(X, ...)")
```

Example:

```
| process eval("ip_address=mvappend(source_address, destination_address)")
| fields source_address, destination_address, ip_address
```

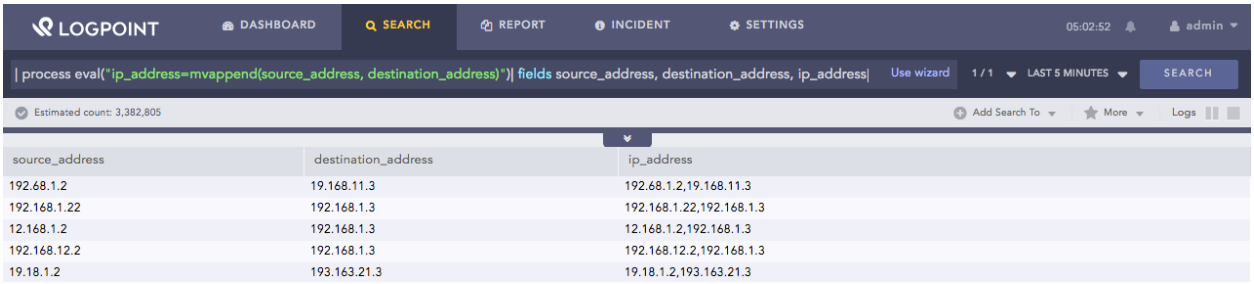


Fig. 3: Using mvappend function

Here, the query returns the appended value of the **source_addess** and **destination_address** fields in the **ip_address** identifier.

The *fields* command displays the value of **source_address**, **destination_address** and **ip_address** in a tabular form.

13.4 mvcount

Accepts either a multivalue field or a single value field *X* and returns the count of that field's values.

Syntax:

```
| process eval("identifier=mvcount(X)")
```

Example:

```
| process eval("message_character_count=mvcount(split(message, ' '))")
| fields message, message_character_count
```

Fig. 4: Using mvcount function

Here, the query compares the values obtained from **split(message, ' ')**. If the values obtained from the **split(message, ' ')** and pattern match, it returns the matched value in the **filter_account_message** identifier. If they don't match, it returns **0**.

The *fields* command displays the value of **message** and **message_character_count** in a tabular form.

13.5 mvdedup

It removes duplicate values of a multivalue field *X* and returns a multivalue output in a list.

Syntax:

```
| process eval("identifier=mvdedup(X)")
```

Example:

```
| process eval("discount_ml=if(discount < 50) {return 'less discount less'}
 else {return 'more discount more'}")
| process eval("result=mvdedup(split(discount_ml, ' '))")
| fields discount, discount_ml, result
```

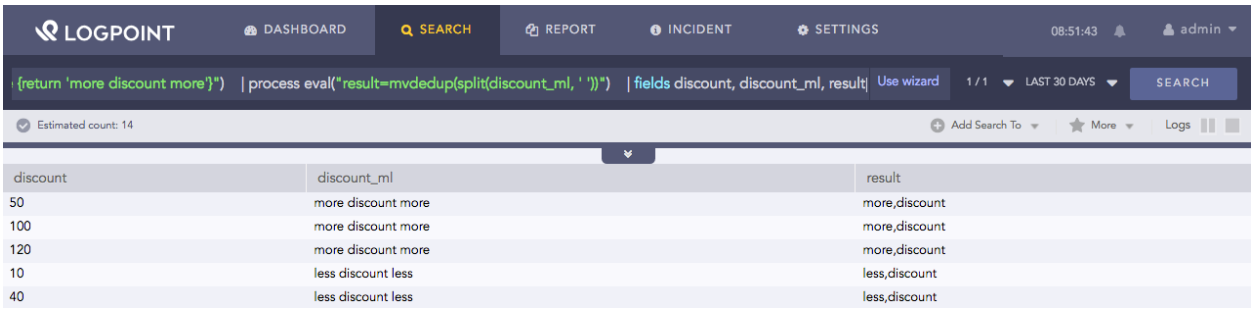


Fig. 5: Using mvdedup function

Here, the query first returns **less discount less** in the **discount_ml** identifier if discount

is less than 50, if it is more it returns **more discount more**. The *split* function splits the value of **discount_ml** when it encounters a space. Then, the *mvdeup* function removes the duplicate values in the value obtained from the **split(discount_ml, ' ')** and returns it in the **result** identifier.

The *fields* command displays the value of **discount**, **discount_ml** and **result** in a tabular form.

13.6 mvfilter

Accepts a multivalue **X** field and a **pattern** as input. It filters the field values using the **pattern** and returns a multivalue or a single value field containing the list of values that match the given **pattern**.

Syntax:

```
| process eval("identifier=mvfilter(X, pattern)")
```

Example:

```
| process eval("filter_account_message=mvfilter(split(message, ' '), 'acco.*')")
| fields message, filter_account_message
```

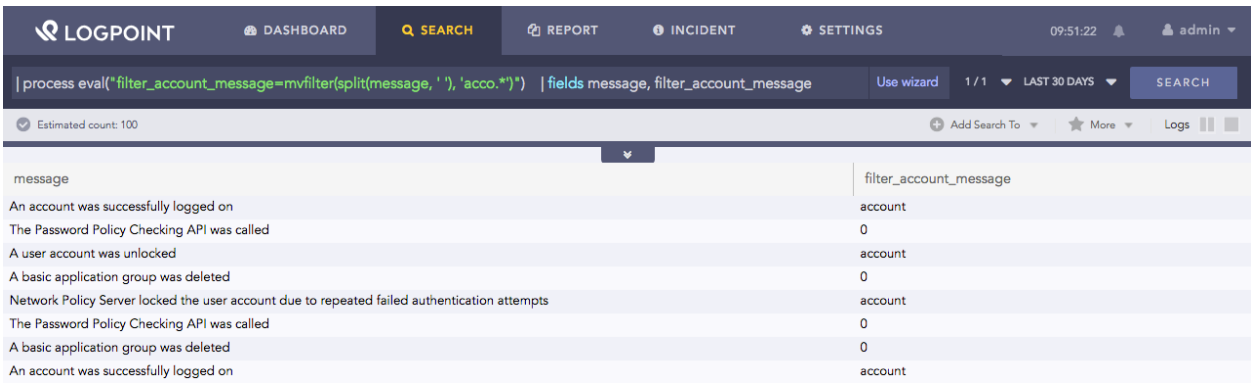


Fig. 6: Using mvfilter function

Here, the query compares the values obtained from **split(message, ' ')** to the pattern **acco.**. Any sequence of letters can follow the string **acco**. If the values obtained from **split(message, ' ')** and pattern matches, it returns the matched value in the *filter_account_message* identifier, if it doesn't match it returns **0**. To learn more about the **split(message, ' ')** value, go to *split*.

The *fields* command displays the value of **fields message** and **filter_account_message** in a tabular form.

13.7 mvfind

Accepts two arguments: a multivalue *X* field and a regex (regular expression) pattern *Y*. It tries to match the regular expression against any substring of *X* value. If there is a match, the function returns the index (beginning from zero) of the first value that matches the regex pattern. If there isn't a match, it returns null.

Syntax:

```
| process eval("identifier=mvfind(X, Y)")
```

Example:

```
| process eval("indexof_account=mvfind(split(message, ' '), 'acco.*')")
| fields message, indexof_account
```

Here, the query searches for the first match in the values obtained from **split(message, ' ')** to the pattern **acco.** Any sequence of letters can follow the **acco** string. If the function finds a match in the values obtained from **split(message, ' ')** to the pattern, it returns the index (position) of the first match (index starts from zero) in the **indexof_account** identifier.

The *fields* command displays the value of **fields message** and **indexof_account* in a tabular form.

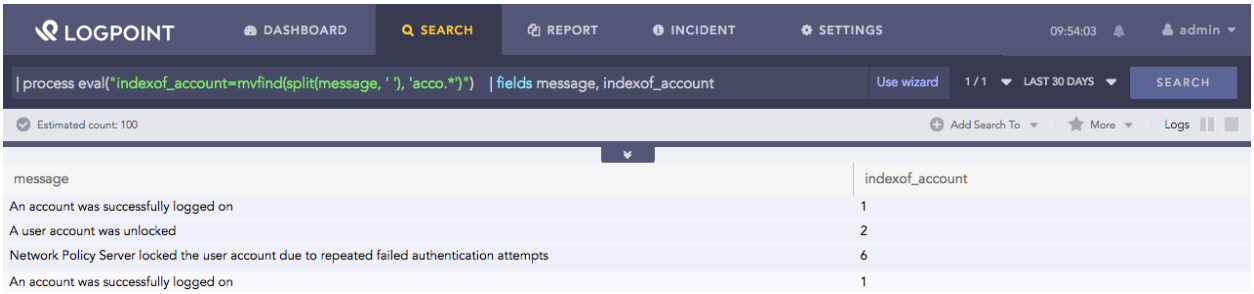


Fig. 7: Using mvfind function

13.8 mvindex

Accepts up to three arguments: a multivalue field *X*, a number *start_index* and a number *end_index*. It evaluates the values of *X* and returns the value that starts at the index specified by *start_index* and ends at the index specified by *end_index*.

- The *X* field and the *start_index* are required. The *end_index* is inclusive and optional while providing positive values for both the start and end index.
- If the *end_index* is not specified, the function returns only the value at *start_index*.

- The *start_index* and *end_index* can be negative. Both the *start_index* and *end_index* is required while providing negative values for either the *start_index* or *end_index*.
- If the indices are out of range or invalid, the result is null.

Syntax:

```
| process eval("identifier=mvindex(X, start_index, end_index)")
```

Example:

```
| process eval("index_message=mvindex(split(message, ' '), 2)")
| fields message, index_message
```

Here, the query returns the third value, where the index starts at 0, from those obtained values from **split(message, ' ')** in the **index_message** identifier.

The *fields* command displays the value of **fields message** and **index_message** in a tabular form.

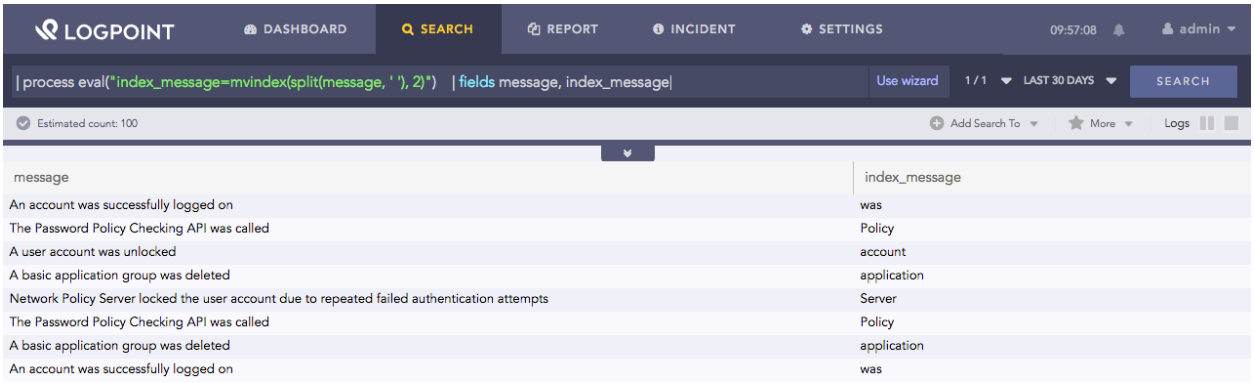


Fig. 8: Using mvindex function

13.9 mvjoin

Accepts two arguments: a multivalue *X* field and a string delimiter (such as comma, semicolon and blank space) *Y*. The function concatenates the individual values within *X* using the value of *Y* as a separator.

Syntax:

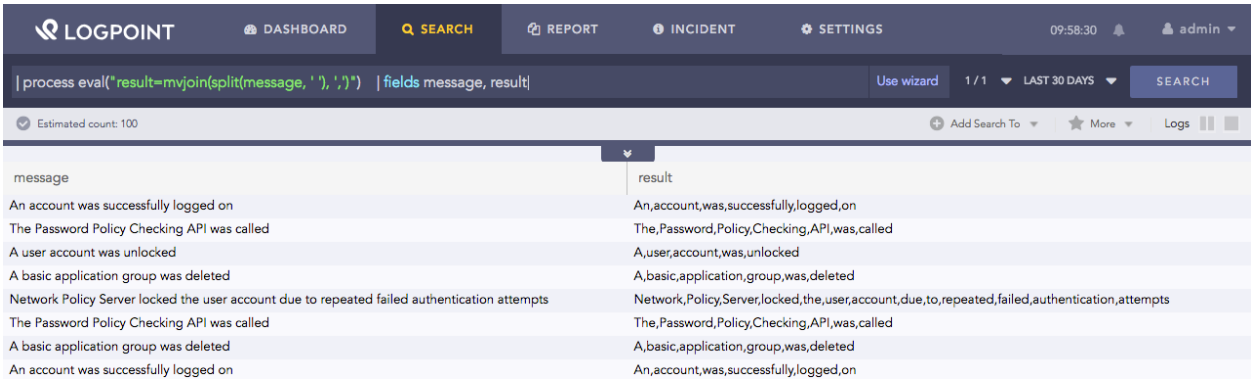
```
| process eval("identifier=mvjoin(X, Y)")
```

Example:

```
process eval("result=mvjoin(split(message, ' '), ',')")
fields message, result
```

Here, the query accepts the values from **split(message, ' ')** and combines them with a comma. It returns the joined values in the **result** identifier. Go to *split* to know on the value from **split(message, ' ')**.

The *fields* command displays the value of the **message** and **result** in a tabular form.



message	result
An account was successfully logged on	An,account,was,successfully,logged,on
The Password Policy Checking API was called	The,Password,Policy,Checking,API,was,called
A user account was unlocked	A,user,account,was,unlocked
A basic application group was deleted	A,basic,application,group,was,deleted
Network Policy Server locked the user account due to repeated failed authentication attempts	Network,Policy,Server,locked,the,user,account,due,to,repeated,failed,authentication,attempts
The Password Policy Checking API was called	The,Password,Policy,Checking,API,was,called
A basic application group was deleted	A,basic,application,group,was,deleted
An account was successfully logged on	An,account,was,successfully,logged,on

Fig. 9: Using mvjoin function

13.10 mvrange

It takes up to three arguments: a starting number *X*, an ending number *Y* and an optional step increment *Z*. It returns the range of *X* and *Y*, where *Y* is excluded from the result.

- If *Z* is not provided, the default increment step is +1.
- If *Z* is a timespan like 1d (1 day), then *X* and *Y* are treated as UNIX time.

Syntax:

```
process eval("identifier=mvrange(X, Y, Z)")
```

Example 1:

```
process eval("range=mvrange(1,5)")
```

Fig. 10: Using mvrange function

Here, the query returns the value from 1 to 5 (excluding 5) with +1 increment in the **range** identifier. The result is 1, 2, 3, 4.

Example 2:

```
| process eval("range=mvrange(1.1,5)")
```

Here, the query returns the value from 1.1 to 5 with +1 increment in the **mvrange** identifier. The result is 1.1, 2.1, 3.1, 4.1.

Example 3:

```
| process eval("range=mvrange(1.5,6,1.5)")
```

Here, the query returns the value from 1.5 to 6 (excluding 6) with +1.5 increment in the **mvrange** identifier. The result is 1.5, 3, 4.5.

Example 4:

```
| process eval("range=mvrange(1134,343434,'1d')")
```

Here, the query returns the value from 1134 to 343434 with +86400 increment in the **mvrange** identifier. The result is 1134, 87534, 173934, 260334.

Here, 1d = 86400 seconds

Example 5:

```
| process eval("range=mvrange(1233.124224,2434455.1232323,'1w')")
```

Here, the query returns the value from 1233.124224 to 2434455.1232323 with +604800 increment in the **mvrange** identifier. The result is 1233.124224, 606033.124224, 1210833.1242240001, 1815633.1242240001, 2420433.124224.

Here, 1w = 604800 seconds

13.11 mvsort

Accepts a multivalue *X* field and returns a multivalue field containing the list of values of *X* sorted in lexicographical or alphabetical order.

- In lexicographical order, numbers are ordered by digits and come before letters. If the first digits match, the second digits get compared. The comparison is like a string. For example, 10 comes before 2, but 111 comes after 10 (and after 1000) because 0 is less than 1. A lexical sort compares the characters in each string as characters, not integral values.
- All uppercase letters comes before lowercase.

Syntax:

```
| process eval("identifier=mvsort(X)")
```

Example 1:

```
| process eval("sort=mvsort(split(message, ' '))") | fields message, sort
```

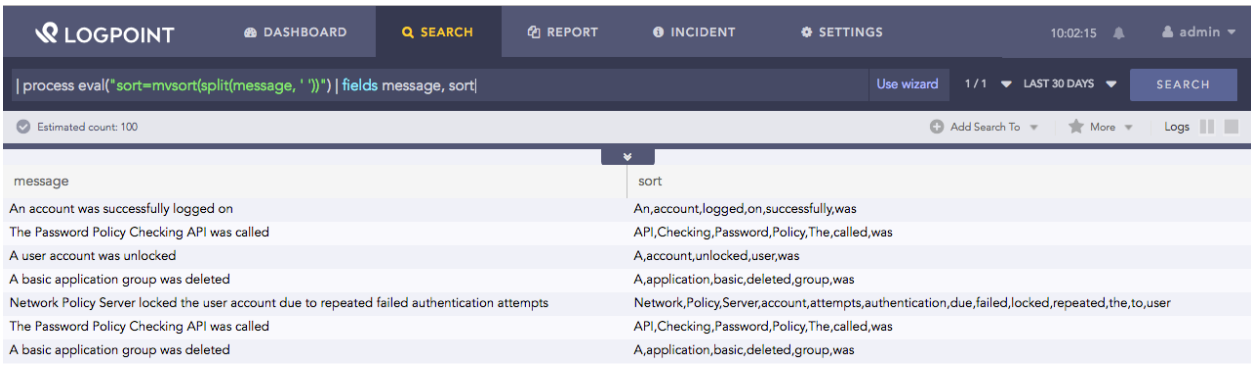


Fig. 11: Using mvsort function

Here, the query accepts the values from **split(message, ' ')** and sorts them according to lexicographical rules. It returns the sorted value in the **sort=mvsort** identifier.

The *fields* command displays the value of the **message** and **sort** in a tabular form.

Example 2:

If testData = {"test1", 1, 1.2}

```
| process eval("sort=mvsort(testData)")
```

Here, the query sorts the data in the *testData* field according to lexicographical rules. It returns the sorted value in the **sort** identifier. The result is [1, 1.2, "test1"].

13.12 mvzip

Accepts up to three arguments: two multivalue *X* and *Y* fields and an optional delimiter *Z*. It combines the first values of X and Y, the second values of X and Y, and so on. *Z* separates the values of *X* and *Y*. The comma is a default delimiter.

Syntax:

```
process eval("identifier=mvzip(X,Y,Z)")
```

Example 1:

If users = [“john”, “jack”, “kim”], machines = [“lp1”, “lp2”, “lp3”]

```
process eval("x=mvzip(users,machines)")
```

Here, the query accepts the **users** and **machines** field values. It combines the first, second, and third values of the **user** field, respectively, with that of the **machines** field. A comma separates the combined values. It returns the combined value in the **X** identifier.

Result: [“john,lp1”, “jack,lp2”, “kim,lp3”]

CHAPTER

FOURTEEN

RELATIONAL EXPRESSIONS

Returns a boolean value, True or False, based on whether the relation specified by the relational operator is satisfied or not.

Generic syntax:

```
| process eval("identifier = first_operand relational_operator second_operand")
```

14.1 Less than (<)

Compares two operands. It returns *true* if the first operand is less than the second operand, or *false* if the first operand is greater than the second.

Example:

```
| process eval("is_loss=Selling_price<cost_price")
| chart count() by Selling_price, cost_price, is_loss
```

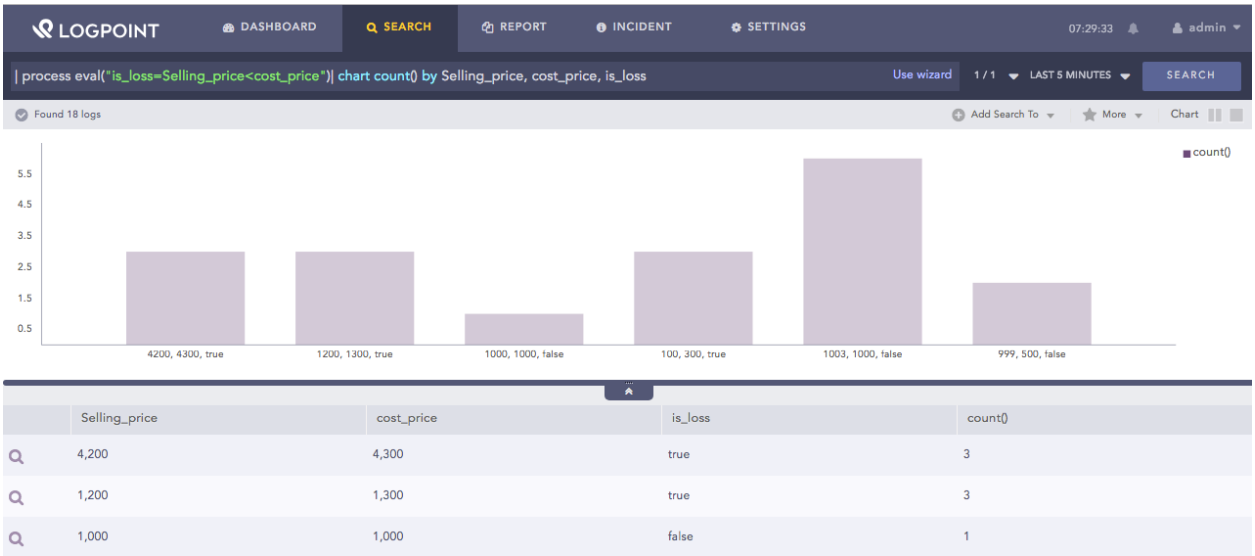


Fig. 1: Using Less than function

Here, the query compares the **cost_price** and **Selling_price** field values. It returns **true** in the **is_loss** identifier if the **cost_price** is less than the **Selling_price**, if it isn't it returns **false**.

The *chart count()* displays the count of the combination of **cost_price** and **Selling_price** values as a chart and in a tabular form.

14.2 Greater than (>)

Compares two operands. It returns *true* if the first operand is greater than the second operand or returns *false* if the first operand is less than the second.

Example:

```
| process eval("is_profit=Selling_price>cost_price")
| chart count() by Selling_price, cost_price, is_profit
```

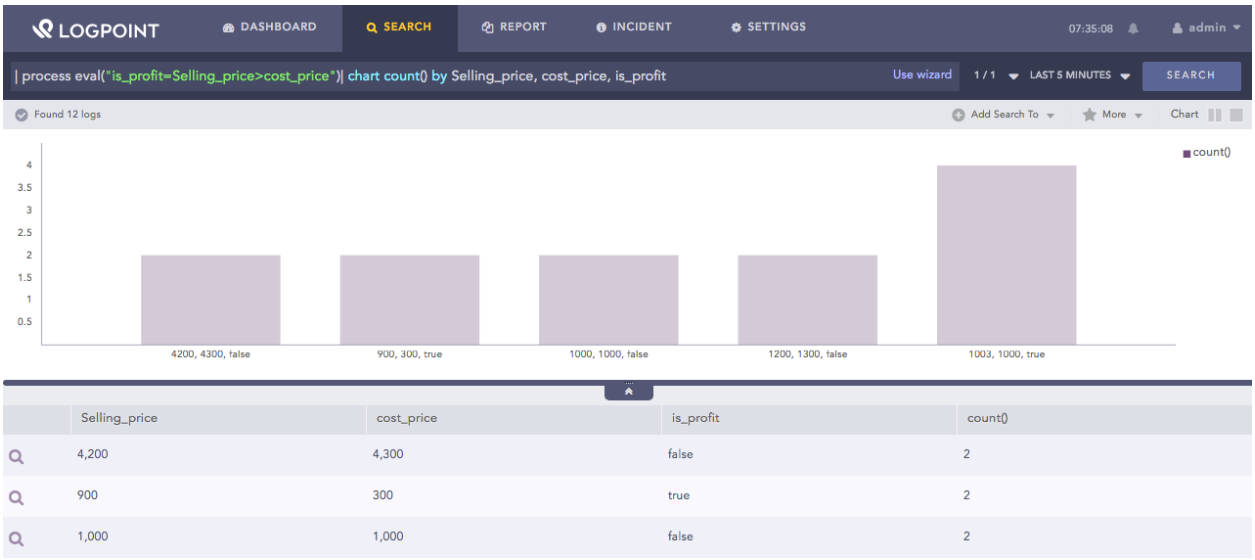


Fig. 2: Using Greater than function

Here, the query compares the **Selling_price** and **cost_price** fields value. It returns **true** in the **is_profit** identifier if the **Selling_price** is greater than the **cost_price**, if it isn't it returns **false**.

The *chart count()* displays the count of the combination of **Selling_price**, **cost_price**, and **is_profit** values as a chart and in a tabular form.

14.3 Less than or equal to (<=)

Compares two operands. It returns *True* if the first operand is **less than or equal to** the second operand, if it isn't it returns *False*.

Example:

```
process eval("is_less_discount=discount<=50")
chart count() by discount, is_less_discount
```

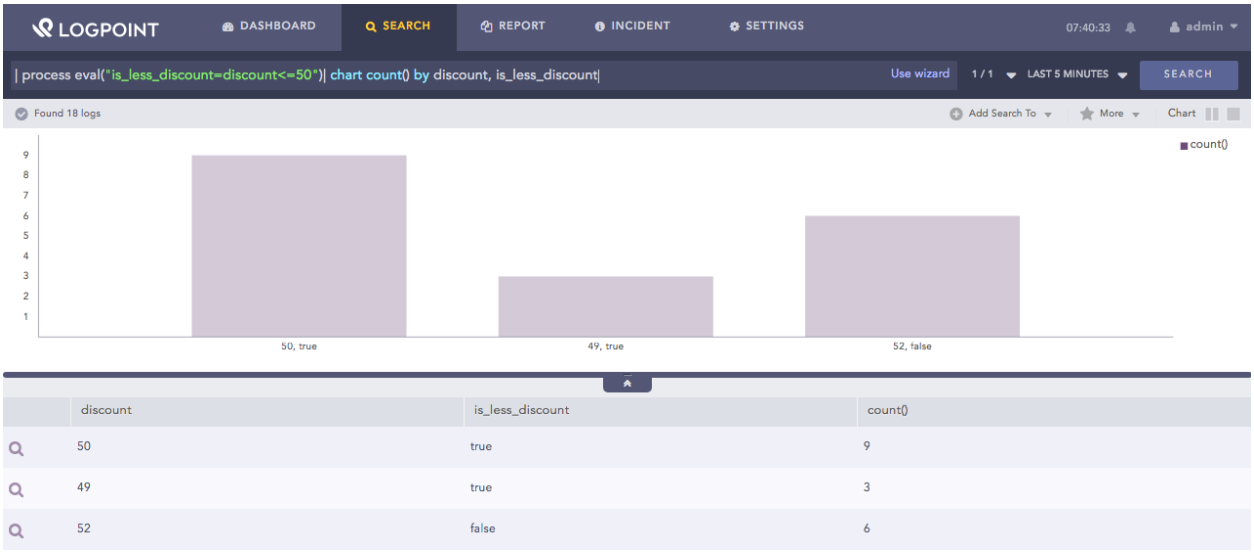


Fig. 3: Using Less than or equals function

Here, the query compares the **discount** field value with 50. It returns **true** in the **is_less_discount** identifier if the **discount** is less than or equal to 50, if it isn't it returns **false**.

The *chart count()* displays the count of the combination of **discount** and **is_less_discount** values as a chart and in a tabular form.

14.4 Greater than or equal to (>=)

Compares two operands. It returns *true* if the first operand is **greater than or equal to** the second operand, if it isn't it returns *false*.

Example:

```
process eval("is_more_discount=discount>=51")
chart count() by discount, is_more_discount
```

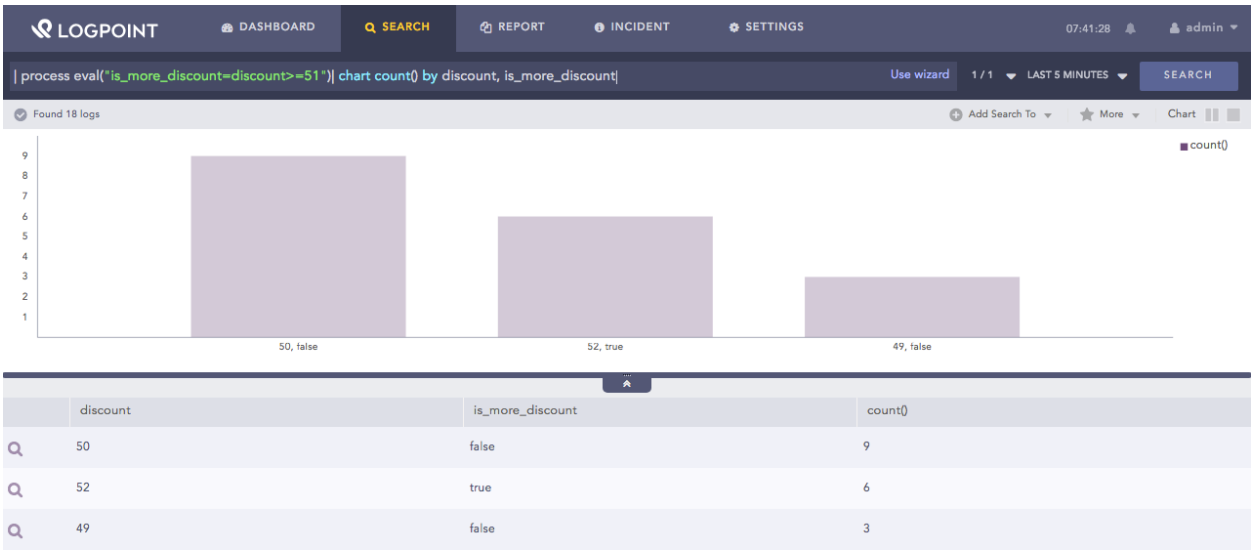


Fig. 4: Using Greater than or equals function

Here, the query compares the **discount** field value with 51. It returns **true** in the **is_more_discount** identifier if the **discount** is greater than or equal to 51, if it isn't it returns **false**.

The *chart count()* displays the count of the combination of **discount** and **is_more_discount** values as a chart and in a tabular form.

14.5 Not equal to (!=)

Compares two operands. It returns *true* if the first operand is **not equal to** the second operand, if it isn't it returns *false*.

Example:

```
| process eval("is_profit_or_loss=Selling_price!=cost_price")
| chart count() by Selling_price, cost_price, is_profit_or_loss
```

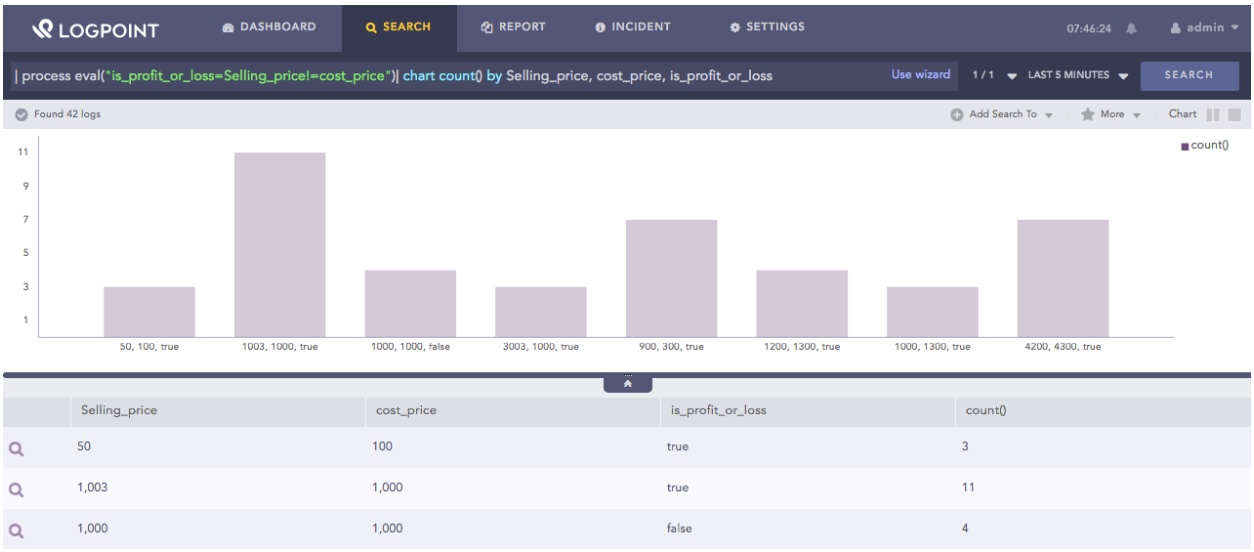


Fig. 5: Using Not equals function

Here, the query compares the **Selling_price** field value with the **cost_price** field. It returns **true** in the **is_profit_or_loss** identifier if the **Selling_price** is not equal to the **cost_price**, if it is it returns **false**.

The *chart count()* displays the count of the combination of **Selling_price**, **cost_price** and **is_profit_or_loss** values as a chart and in a tabular form.

14.6 Equal to (==)

Compares two operands. It returns *true* if the first operand is **equal to** the second operand, if it isn't it returns *false*.

Example:

```
| process eval("no_profit_or_loss=Selling_price==cost_price")
| chart count() by Selling_price, cost_price, no_profit_or_loss
```

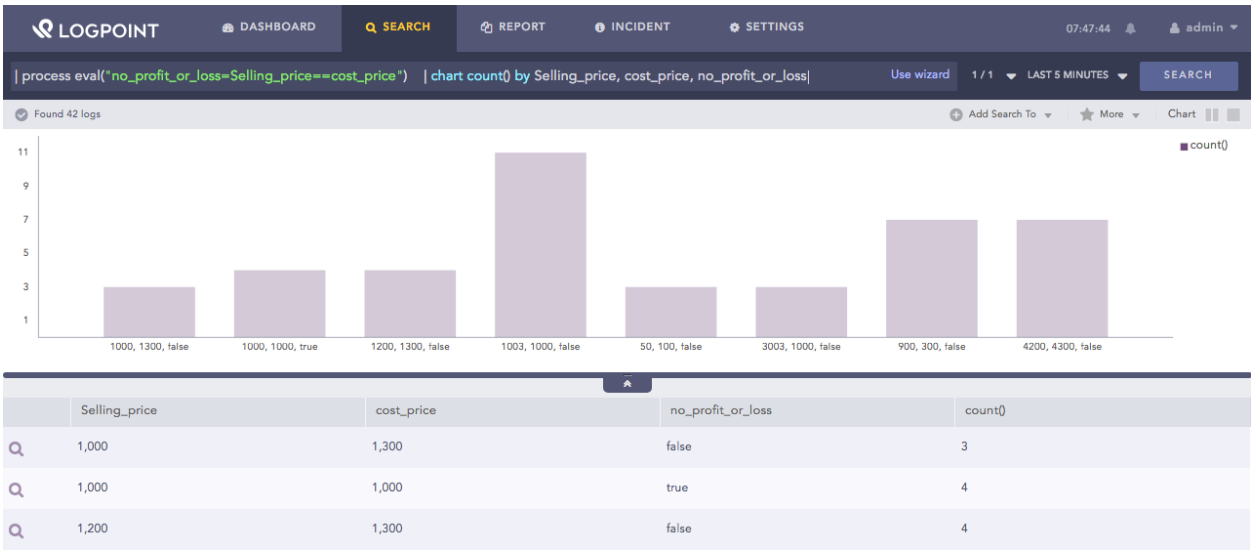


Fig. 6: Using Equal to function

Here, the query compares the **Selling_price** field value with the **cost_price** field. It returns **true** in the **no_profit_or_loss** identifier if the **Selling_price** is equal to the **cost_price**, if it isn't it returns **false**.

The **chart count()* displays the count of the combination of **Selling_price**, **cost_price**, and **no_profit_or_loss** values as a chart and in a tabular form.

CHAPTER

FIFTEEN

STATISTICAL FUNCTIONS

Evaluates the maximum and minimum value of numeric or string argument.

15.1 max

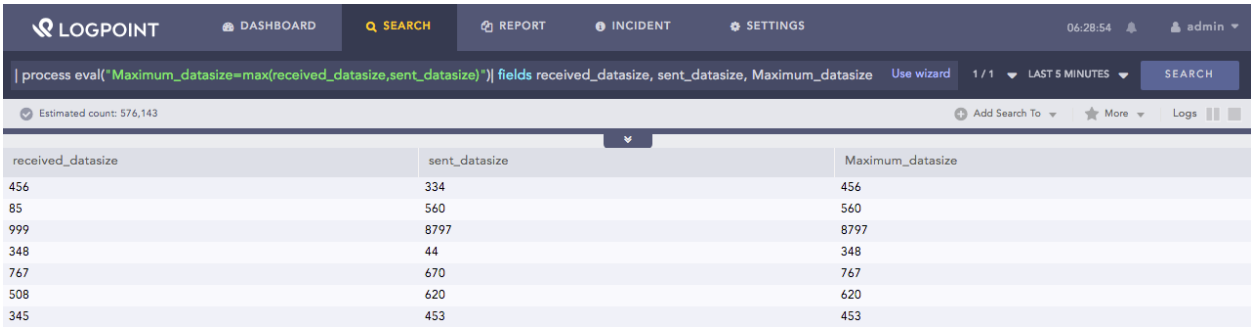
Accepts an arbitrary number of arguments as input and returns the maximum of any numeric or string argument. The values of arguments get converted into ASCII (American Standard Code for Information Interchange), so a string is greater than a number.

Syntax:

```
| process eval("identifier=max(X, ...)")
```

Example:

```
| process eval("Maximum_datasize=max(received_datasize,sent_datasize)")
| fields received_datasize, sent_datasize, Maximum_datasize
```



received_datasize	sent_datasize	Maximum_datasize
456	334	456
85	560	560
999	8797	8797
348	44	348
767	670	767
508	620	620
345	453	453

Fig. 1: Using max function

Here, the query returns the highest value of the **received_datasize** and **sent_datsize** fields in the **Maximum_datasize** identifier.

The *fields* command displays the value of **received_datasize**, **sent_datasize** and **Maximun_datasize** in a tabular form.

15.2 min

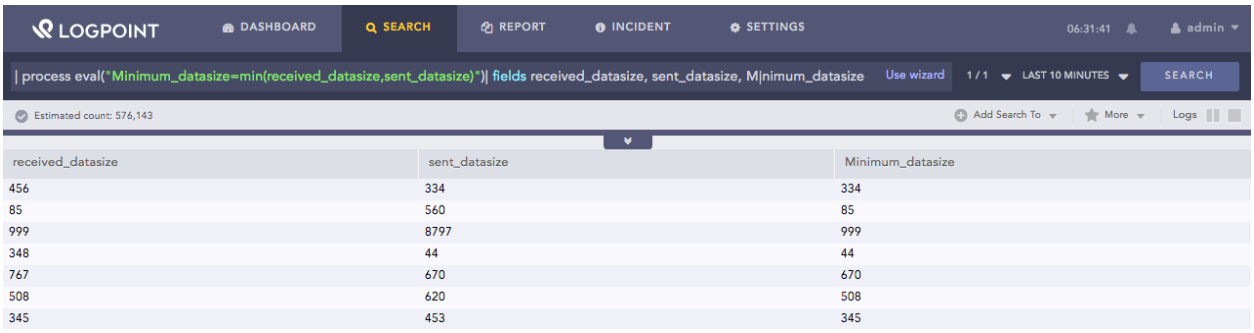
Accepts an arbitrary number of arguments as input and returns the minimum of any numeric or string argument. The values of arguments get converted into ASCII, so a string is greater than a number.

Syntax:

```
| process eval("identifier=min(X, ...)")
```

Example:

```
| process eval("Minimum_datasize=min(received_datasize,sent_datasize)")
| fields received_datasize, sent_datasize, Minimum_datasize
```



received_datasize	sent_datasize	Minimum_datasize
456	334	334
85	560	85
999	8797	999
348	44	44
767	670	670
508	620	508
345	453	345

Fig. 2: Using min function

Here, the query returns the lowest value of the **received_datasize** and **sent_datsize** fields in the **Minimum_datasize** identifier.

The *fields* command displays the value of **received_datasize**, **sent_datasize** and **Minimum_datasize** in a tabular form.

CHAPTER

SIXTEEN

STRING FUNCTIONS

Evaluates string values and fields.

16.1 len

Accepts a string value *X* as input. It evaluates the string's character length and returns the count of a character's number in the string.

Syntax:

```
| process eval("identifier=len(X)")
```

Example:

```
| process eval("message_length=len(message)")
| fields message, message_length
```

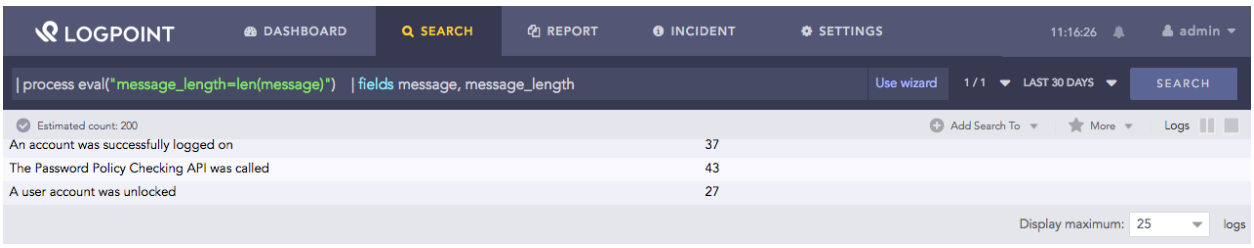


Fig. 1: Using len function

Here, the query counts the **message** field's character length and returns the result in the **message_length** identifier.

The *fields* command displays the value of the **message** and **message_length** in a tabular form.

16.2 issubstr

Accepts two arguments: a string value *X* and a source string *Y*. It returns **true** if *X* is a substring of *Y*. The substring can be at any position of the source string.

Syntax:

```
| process eval("identifier=issubstr(X,Y)")
```

Example 1:

```
| process eval("result=issubstr('WSS','AWSService') ")
```

Fig. 2: Using issubstr function

Here, the query returns **true** value in **result** field as **WSS** is sub string of **AWSService**.

Example 2:

```
| process eval("exists=issubstr('mal.exe','hi.exmal.exe,ok.dm') ")
```

Fig. 3: Using issubstr function

Here, the query returns **true** value in **exists** field as **mal.exe** is sub string of **hi.exmal.exe,ok.dm**.

16.3 substr

Accepts up to three arguments, a string value *X*, a start index and an end index. It evaluates the substring of *X* and returns the substring that starts at the index specified by *start_index* and ends at the index specified by *end_index*. Here the *end_index* is exclusive.

Syntax:

```
| process eval("identifier=substr(X, start_index, end_index)")
```

Example:

```
| process eval("substring=substr(col_type, 0, 4)")
```

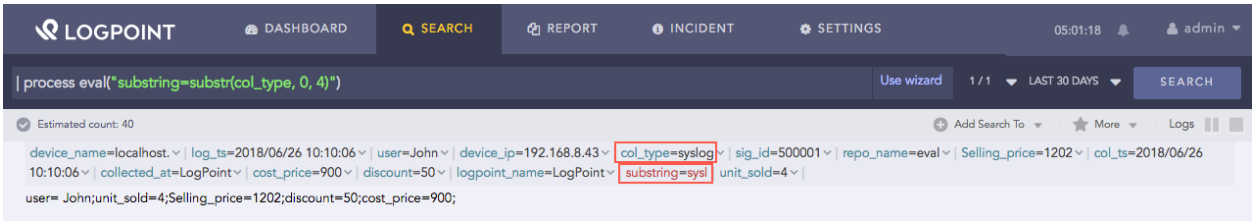


Fig. 4: Using substr function

Here, the query checks the **col_type** event's substring starting at **0** index and ending at **4** index and returns the result in **substring** identifier.

16.4 lower

Accepts only one string argument *X* as input. It converts the string to lowercase and returns the converted string value.

Syntax:

```
| process eval("identifier=lower(X)")
```

Example:

```
| process eval("username=lower(user)") | fields user, username
```

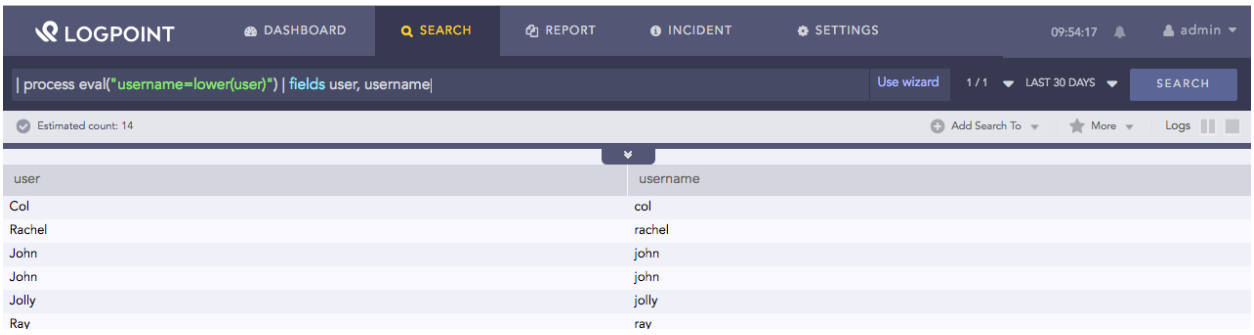


Fig. 5: Using lower function

Here, the query converts the **user** field value to lowercase and returns the result in the **username** identifier.

The *fields* command displays the value of **user** and **username** in a tabular form.

16.5 upper

Accepts only one string argument *X* as input and converts the string to uppercase and returns the converted string value.

Syntax:

```
| process eval("identifier=upper(string_value)")
```

Example:

```
| process eval("username=upper(user)") | fields user, username
```

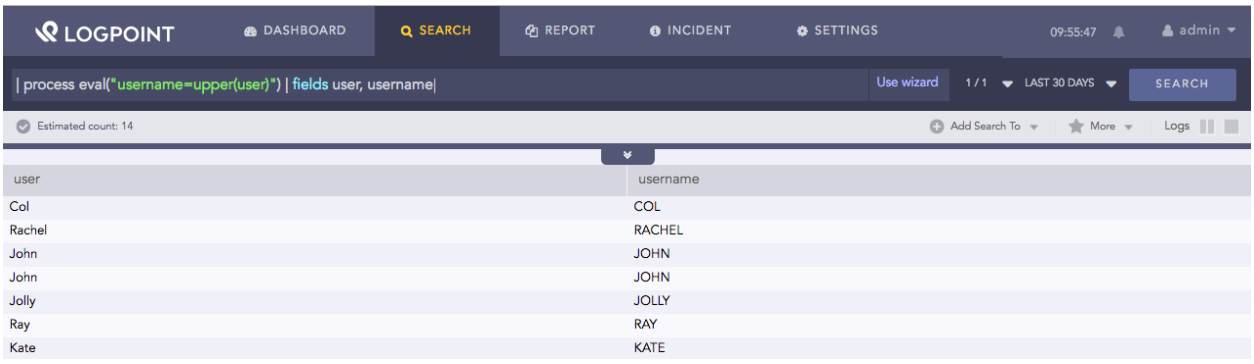


Fig. 6: Using upper function

Here, the query converts the **user** field value to uppercase and returns the result in the **username** identifier.

The *fields* command displays the value of **user** and **username** in a tabular form.

16.6 trim

Accepts only one string argument *X*. It trims the spaces to the left and right in the string and returns a trimmed value. Trailing spaces are the white spaces located at the end of a line, without any other characters following it, for example blank spaces and tabs.

Syntax:

```
| process eval("identifier=trim(X)")
```

Example:

```
| process eval("username=trim(' Bob ')")
```

Fig. 7: Using trim function

Here, the query removes the spaces to the left and right from **Bob** and returns the trimmed value in the **username** identifier.

16.7 ltrim

Accepts up to two string arguments *X* and *Y* as input. It trims the string *Y* from the left side of the field *X* and returns a trimmed value. If *Y* is not defined, it trims the spaces from the left side.

Syntax:

```
| process eval("identifier=ltrim(X, Y)")
```

Example:

```
| process eval("result=ltrim(device_name, 'local')")
```

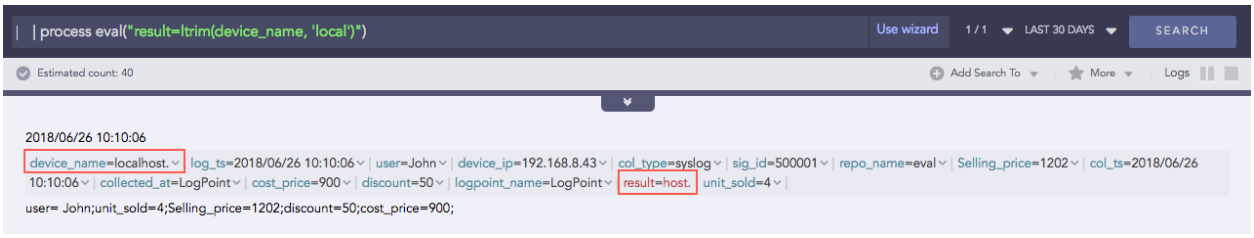


Fig. 8: Using ltrim function

Here, the query removes the string **local** from the left side in the value of the **device_name** field and returns the trimmed value in the **result** identifier.

16.8 rtrim

It takes up to two string arguments: *X* and *Y*. It trims *Y* from the right side of the *X* field and returns a trimmed value. If *Y* is not defined, it trims the trailing spaces from the right side.

Syntax:

```
| process eval("identifier=rtrim(X, Y)")
```

Example:

```
| process eval("result=rtrim(device_name, 'host')")
```

Fig. 9: Using rtrim function

Here, the query removes the **host** string from the right side of the **device_name** field value and returns the trimmed value in the **result** identifier.

16.9 replace

Accepts three arguments as input: a string *X*, a regex string *Y* and a string *Z*. It substitutes the string *Z* in the string *X* for every occurrence of the regex string *Y* and

returns a string value.

Syntax:

```
| process eval("identifier=replace(X, Y, Z)")
```

Example:

```
| process eval("result=replace('123', '[0-9]', 'X')")
```

Fig. 10: Using replace function

Here, the query substitutes **X** in the **123** string for the every occurence of the **[0-9]** regex string and returns the replaced value in the **result** identifier.

16.10 spath

Accepts two arguments: *X* and *Y*. It returns a value extracted from the structured data type in *X*, based on the location path in *Y*.

Syntax:

```
| process eval("identifier=spath(X, Y)")
```

X: The structured data type in XML or JSON format.

Y: The XML or JSON formatted location path.

Example 1:

```
| process eval("usern=spath('<name>john</name>', 'name')")
```

Fig. 11: Using spath function

Here, the query extracts the value from the **name** location and returns it in the **usern** identifer.

Example 2:

```
| process eval("usern=spath('{name:\john\}', 'name')")
```

Fig. 12: Using spath function

Here, the query extracts the value from the **name:** location and returns it in the **usern** identifer.

Note: For JSON format data,

- Keys must be without quotes. LogPoint currently does not support nested quotes.
- If the value of any key is a string, replace quote with backslash as shown in *Example 2* above.
- For example, the JSON data is in a key-value pair. Where, keys and values must be within double quotes {"name":"John"}. However, while using the **spath** function, the JSON data is written as {name:\john\}.

16.11 urldecode

Accepts an escaped URL character *X*, for example **http://www.logpoint.com/download?r=header** and returns the decoded or unescaped URL string.

Syntax:

```
| process eval("identifier=urldecode(X)")
```

Example:

```
| process eval("decoded_url=urldecode('http%3A%2F%2Fwww.logpoint.com%2Fdownload%3Fr
↪%3Dheader')")
```

Fig. 13: Using urldecode function

Here, the query decodes the escaped url **http%3A%2F%2Fwww.logpoint.com%2Fdownload%3Fr%** and returns the decoded url in the **decoded_url** identifier.

16.12 uuid

It generates a random Universal Unique Identifier (UUID) for a log.

Syntax:

```
| process eval("X=uuid()")
```

Example 1:

```
| process eval("id=uuid()")
```

Fig. 14: Using uuid function

Here, the query generates a random uuid for the log and returns the uuid in the **id** field.

Example 2:

```
| process eval("random_id=uuid()") |chart count() by uuid
```

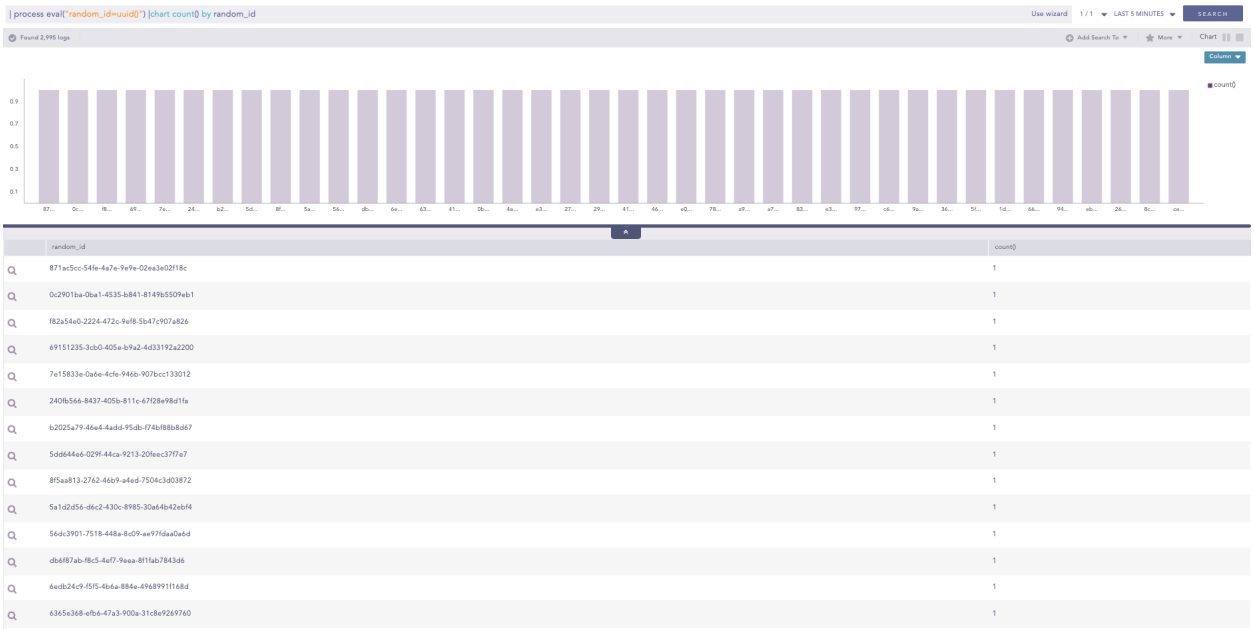


Fig. 15: Using uuid function

Here, the query generates a random uuid for each log and returns the count of uuid in the **random_id** field.

16.13 mimedecode

It decodes the Multipurpose Internet Mail Extensions (MIME) encoded values. It accepts an argument: an encoded string, for example, **=?utf-8?B?MTIzIFRlc3Rp?=** or a field with a valid MIME encoded string with metadata.

Syntax:

```
| process eval("X=mimedecode(Y)")
```

Example 1:

```
| process eval("result=mimedecode('=?utf-8?B?MTIzIFRlc3Rp?=')")
```

Fig. 16: Using mimedecode function

Here, the query decodes the encoded string **'=?utf-8?B?MTIzIFRlc3Rp?='** and returns the decoded value in the **result** field.

Example 2:

```
| process eval("result=mimedecode('Subject: =?iso-8859-1?Q?=A1Hola,_se=F1or!?=')")
```

Fig. 17: Using mimedecode function

Here, the query decodes the **Subject** field and returns the decoded value in the **result** field.

Use Case 1:

mimedecode() eval command chaining.

Query

```
| process eval("result1=mimedecode('=?UTF-8?B?U3ViamVjdDogPT9pc28tODg1OS0xP1E/
↪PUExSG9sYSxfc2U9RjFvciE/PQ==?=')")
| process eval("result2=mimedecode(result1)")
```

Encoded String

```
U3ViamVjdDogPT9pc28tODg1OS0xP1E/PUExSG9sYSxfc2U9RjFvciE/PQ==
```

Charset/encoding

```
iso-8859-1 -> UTF-8
```

Type

```
Query Printable -> Base64
```

Result 1

```
Subject: =?iso-8859-1?Q?=A1Hola,_se=F1or!?=
```

Result 2

```
Subject: ¡Hola,_señor!
```

Use Case 2:

mimedecode() working with other eval commands.

Query

```
| process eval("result1=mimedecode('=?UTF-8?B?U3ViamVjdDogPT9pc28tODg1OS0xP1E/
↪PUExSG9sYSxfc2U9RjFvciE/PQ==?=')")
| process eval("result2=mimedecode(result1)")
| process eval("result3=mimedecode(mimedecode('=?UTF-8?B?
↪U3ViamVjdDogPT9pc28tODg1OS0xP1E/PUExSG9sYSxfc2U9RjFvciE/PQ==?='))")
| process eval("result=result2==result3")
| fields result1, result2, result3, result
```

Encoded String

```
U3ViamVjdDogPT9pc28tODg1OS0xP1E/PUExSG9sYSxfc2U9RjFvciE/PQ==
```

Charset/encoding

```
iso-8859-1 -> UTF-8
```

Type

```
Query Printable -> Base64
```

Result 1

```
Subject: =?iso-8859-1?Q?=A1Hola,_se=F1or!?=
```

Result2

Subject: ¡Hola,_señor!

Result3

Subject: ¡Hola,_señor!

Result

true

CHAPTER

SEVENTEEN

TRIGONOMETRIC FUNCTIONS

Evaluates trigonometric and hyperbolic values.

17.1 sin

Accepts one argument *X* as input and returns the sine of *X*.

Syntax:

```
| process eval("identifier=sin(X)")
```

Example:

```
| process eval("sin_value=sin(length)") | fields length, sin_value
```

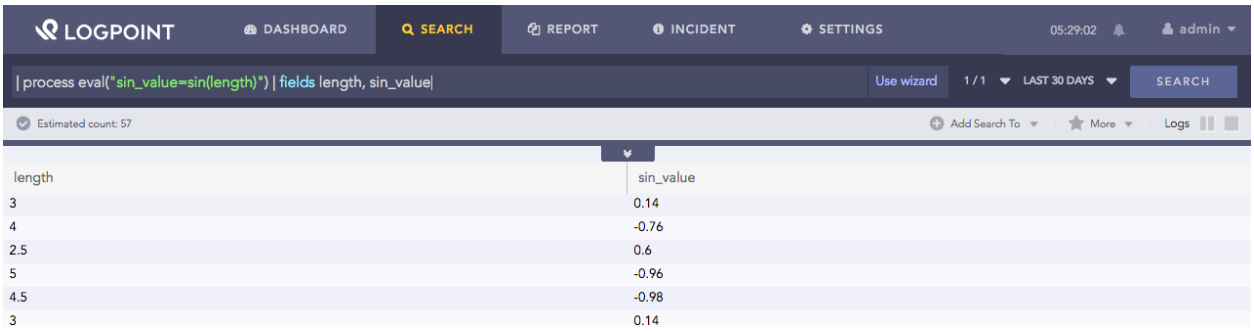


Fig. 1: Using sin function

Here, the query returns the sine of the **length** field in the **sin_value** identifier.

The *fields* command displays the value of **length** and **sin_value** fields in a tabular form.

17.2 sinh

Accepts one argument *X* as input and returns the hyperbolic sine of *X*.

Syntax:

```
| process eval("identifier=sinh(X)")
```

Example:

```
| process eval("sinh_value=sinh(length)") | fields length, sinh_value
```

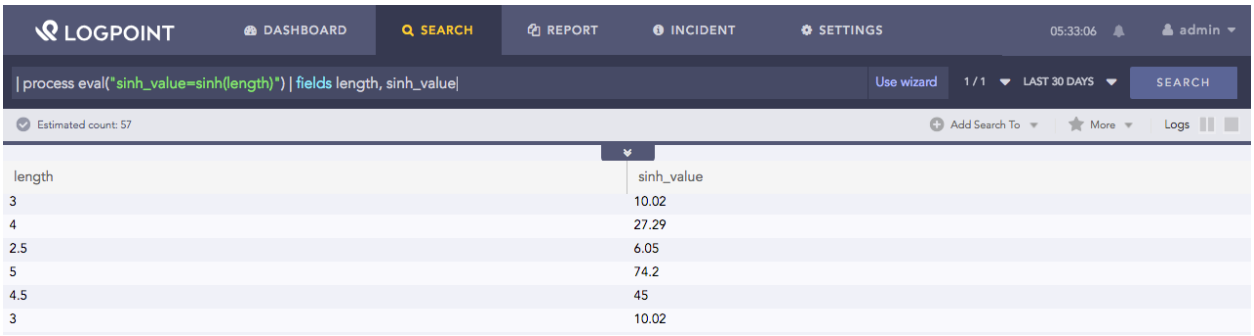


Fig. 2: Using sinh function

Here, the query returns the hyperbolic sine of the **length** field in the **sinh_value** identifier.

17.3 asin

Accepts one argument *X* as input and returns the inverse sine of *X*. The value of *X* must be in the range from -1 to 1 inclusive.

Syntax:

```
| process eval("identifier=asin(X)")
```

Example:

```
| process eval("asin_value=asin(1)")
```

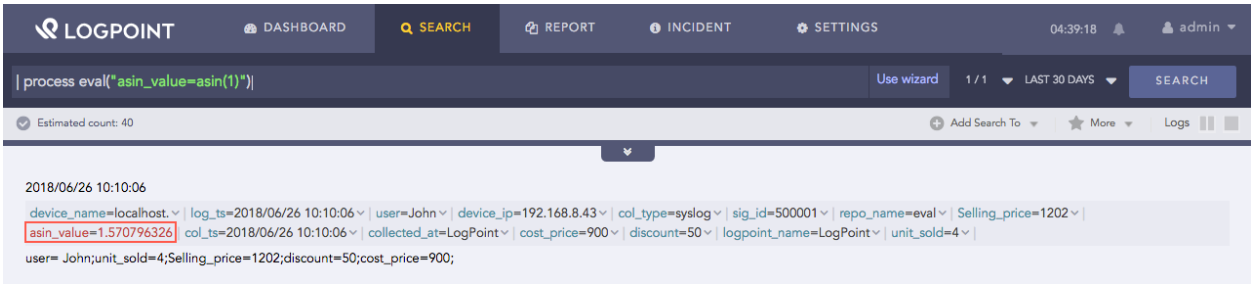


Fig. 3: Using asin function

Here, the query returns the inverse sine of **1** in the **asin_value** identifier.

17.4 asinh

Accepts one argument *X* as input and returns the inverse hyperbolic sine of *X*.

Syntax:

```
| process eval("identifier=asinh(X)")
```

Example:

```
| process eval("asinh_value=asinh(1)")
```

Fig. 4: Using asinh function

Here, the query returns the inverse hyperbolic sine of **1** in the **asinh_value** identifier.

17.5 cos

Accepts one argument *X* as input and returns the cosine of *X*.

Syntax:

```
| process eval("identifier=cos(X)")
```

Example:

```
| process eval("cos_value=cos(length)") | fields length, cos_value
```

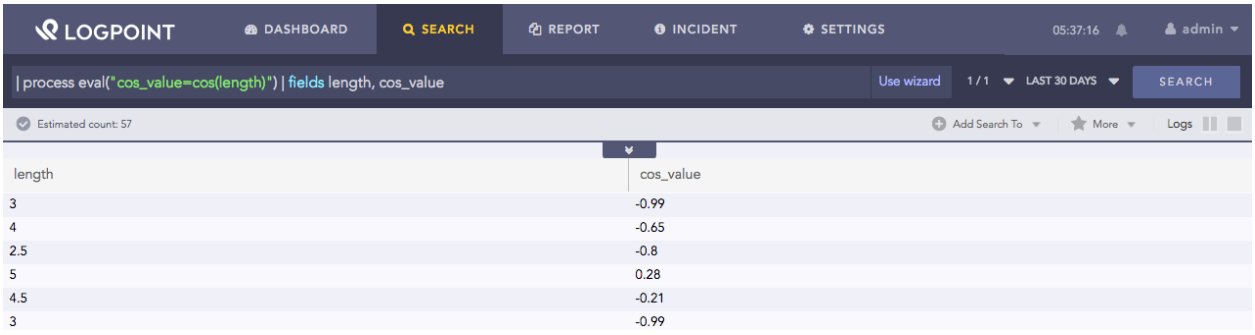


Fig. 5: Using cos function

Here, the query returns the cosine of the **length** field in the **cos_value** identifier.

The *fields* command displays the value of **length** and **cos_value** fields in a tabular form.

17.6 cosh

Accepts one argument *X* as input and returns the hyperbolic cosine of *X*.

Syntax:

```
| process eval("identifier=cosh(X)")
```

Example:

```
| process eval("cosh_value=cosh(length)") | fields length, cosh_value
```

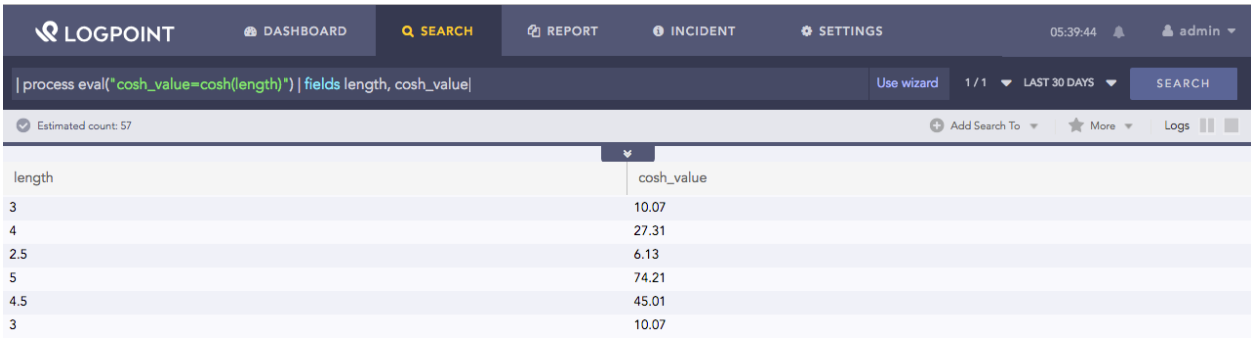


Fig. 6: Using cosh function

Here, the query returns the hyperbolic cosine of the **length** field in the **cosh_value** identifier.

The *fields* command displays the value of **length** and **cosh_value** fields in a tabular form.

17.7 acos

Accepts one argument *X* as input and returns the inverse cosine of *X*. *X* must be in the range from -1 to 1 inclusive.

Syntax:

```
| process eval("identifier=acos(X)")
```

Example:

```
| process eval("acos_value=acos(0)")
```

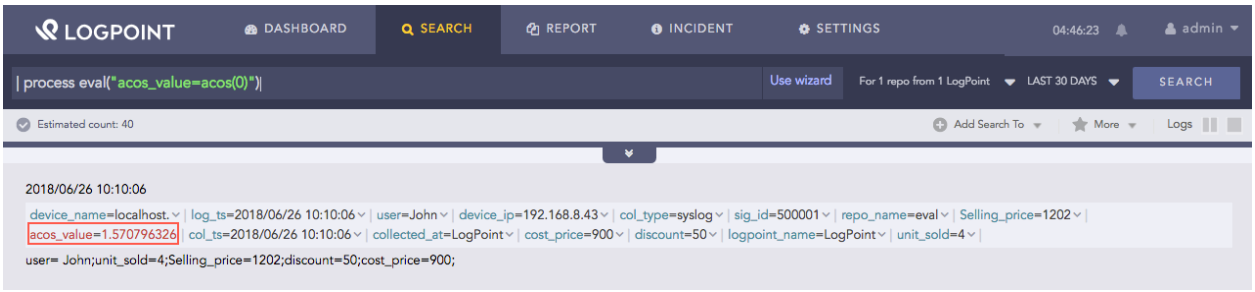


Fig. 7: Using acos function

Here, the query returns the inverse cosine of **0** in the **acos_value** identifier.

17.8 acosh

Accepts one argument *X* as input and returns the inverse hyperbolic cosine of *X*.

Syntax:

```
| process eval("identifier=acosh(X)")
```

Example:

```
| process eval("acosh_value=acosh(1)")
```

Fig. 8: Using acosh function

Here, the query returns the inverse hyperbolic cosine of **1** in the **acosh_value** identifier.

17.9 tan

Accepts one argument *X* as input and returns the tangent of *X*.

Syntax:

```
| process eval("identifier=tan(X)")
```

Example:

```
| process eval("tan_value=tan(length)")
| fields length, tan_value
```

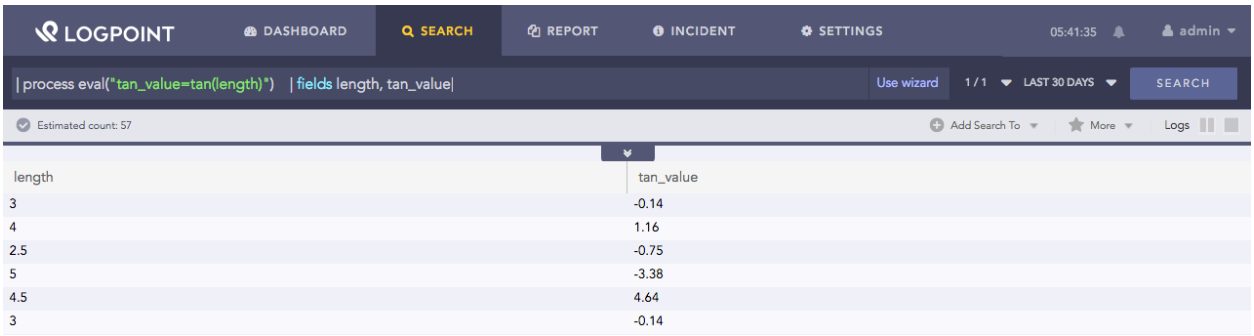


Fig. 9: Using tan function

Here, the query returns the tangent of the **length** field in the **vtan_value** identifier.

The *fields* command displays the value of **length** and **tan_value** fields in a tabular form.

17.10 tanh

Accepts one argument *X* as input and returns the hyperbolic tangent of *X*.

Syntax:

```
| process eval("identifier=tanh(X)")
```

Example:

```
| process eval("tanh_value=tanh(length)")
| fields length, tanh_value
```

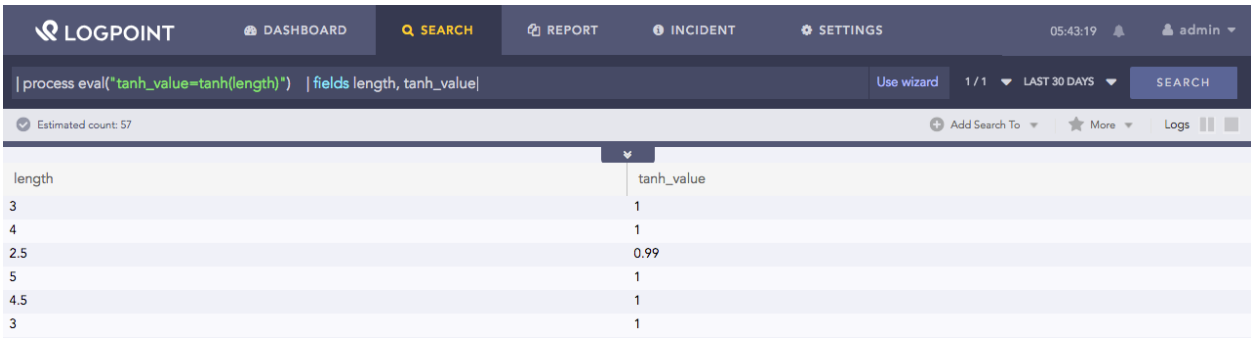


Fig. 10: Using tanh function

Here, the query returns the hyperbolic tangent of the **length** field in the **tanh_value** identifier.

The *fields* command displays the value of **length** and **tanh_value** fields in a tabular form.

17.11 atan

Accepts one argument *X* as input and returns the inverse tangent of *X*.

Syntax:

```
| process eval("identifier=atan(X)")
```

Example:

```
| process eval("atan_value=atan(1)")
```

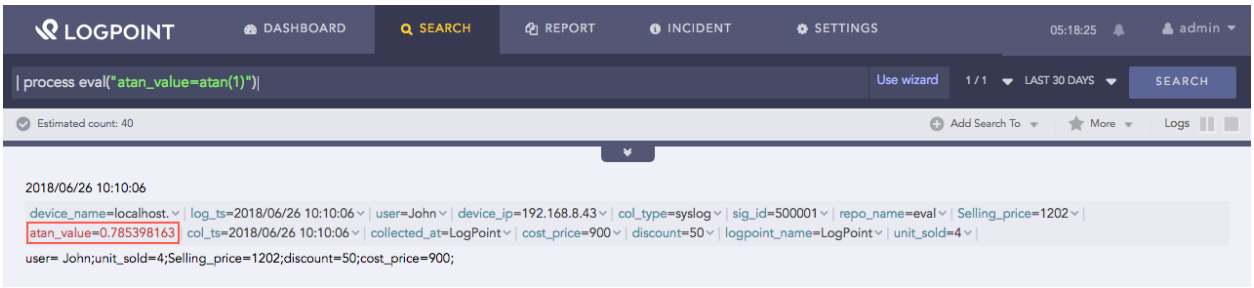


Fig. 11: Using atan function

Here, the query returns the inverse tangent of **1** in the **atan_value** identifier.

17.12 atanh

Accepts one argument *X* as input and returns the inverse hyperbolic tangent of *X*.

Syntax:

```
| process eval("identifier=atan(X)")
```

Example:

```
| process eval("atanh_value=atanh(1)")
```

Fig. 12: Using atanh function

Here, the query returns the inverse hyperbolic tangent of **1** in the **atanh_value** identifier.

17.13 hypot

Accepts two arguments *X* and *Y* as input and returns the hypotenuse of a right-angled triangle with *X* length and *Y* base. It follows the equation of the Pythagorean theorem, (hypotenuse = sqrt{length^2 + base^2}).

Syntax:

```
| process eval("identifier=hypot(X,Y)")
```

Example:

```
| process eval("hyp=hypot(3,4)")
```

Fig. 13: Using hypot function

Here, the query calculates the hypotenuse value of the triangle with **3** length and **4** base and returns its value in the **tan_value** identifier.