

Integrations

Evaluation Process Plugin

V3.0.0

CONTENTS

1	Evaluation Process Plugin	1
2	Arithmetic expressions	3
2.1	Addition (+)	3
2.2	Subtraction (-)	4
2.3	Multiplication (*)	4
2.4	Division (/)	5
2.5	Modulus (%)	5
2.6	Power (^)	6
3	Relational expressions	7
3.1	Less than (<)	7
3.2	Greater than (>)	8
3.3	Less than or equal to (<=)	9
3.4	Greater than or equal to (>=)	10
3.5	Not equal to (!=)	11
3.6	Equal to (==)	12
4	Logical expressions	14
4.1	AND (&&)	14
4.2	OR ()	15
5	Conditional and Comparison functions	17
5.1	If Statement	17
5.2	If-else Statement	18
5.3	If-elseif-else Statement	18
5.4	Case Statement	19
5.5	cidrmatch	20
5.6	coalesce	21
5.7	false	22
5.8	in	22
5.9	match	23
5.10	like	24

5.11	null	25
5.12	nullif	26
5.13	searchmatch	27
5.14	true	28
6	Multivalue functions	29
6.1	split	29
6.2	commands	30
6.3	mvappend	30
6.4	mvcount	31
6.5	mvdedup	32
6.6	mvfilter	32
6.7	mvfind	33
6.8	mvindex	34
6.9	mvjoin	35
6.10	mvrange	36
6.11	mvsort	37
6.12	mvzip	38
7	Statistical functions	40
7.1	max	40
7.2	min	41
8	Conversion functions	42
8.1	printf	42
8.2	tonumber	43
8.3	tostring	44
9	Expression with Parentheses	46
10	String functions	47
10.1	len	47
10.2	substr	48
10.3	lower	48
10.4	upper	49
10.5	trim	50
10.6	ltrim	50
10.7	rtrim	51
10.8	replace	51
10.9	spath	52
10.10	urldecode	53
11	Cryptographic functions	54
11.1	md5	54
11.2	sha1	55
11.3	sha256	55

11.4 sha512	56
12 Mathematical functions	57
12.1 abs	57
12.2 floor	58
12.3 ceiling	58
12.4 exp	59
12.5 exp2	60
12.6 exp10	61
12.7 log	61
12.8 log2	62
12.9 log10	63
12.10pi	63
12.11sqrt	64
12.12random	65
12.13exact	66
12.14round	66
12.15sigfig	67
13 Trigonometric functions	69
13.1 sin	69
13.2 sinh	69
13.3 asin	70
13.4 asinh	71
13.5 cos	71
13.6 cosh	72
13.7 acos	72
13.8 acosh	73
13.9 tan	74
13.10tanh	74
13.11atan	75
13.12atanh	75
13.13hypot	76
14 Date/Time functions	77
14.1 now	77
14.2 relative_time	77
14.3 strftime	78
14.4 strptime	79
14.5 time	80
14.6 Date/Time patterns	80
15 Informational functions	82
15.1 isbool	82
15.2 isint	83

15.3	isnotnull	84
15.4	isnull	85
15.5	isnum	86
15.6	isstr	87
15.7	typeof	88
16	Uninstalling the Application	89
16.1	Uninstalling the Evaluation Process Plugin Application in LogPoint	89

EVALUATION PROCESS PLUGIN

The **Evaluation Process Plugin** enables the use of the **eval process command**. This command evaluates mathematical, boolean, and string expressions during a LogPoint search and places the result of the evaluation in an **identifier** as a new field.

Syntax:

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

- **Expression:** An expression is a combination of numbers, variables, operators, functions, brackets and punctuation marks that are grouped to represent a value.
- **Identifier:** An identifier contains the result of the evaluation of expressions.

Note:

- Make sure the name of the identifier is not the same as an existing field name. If the same name is used Logpoint discards the value of the identifier.
- While using a string value in an eval expression, always place the string within single quotes ('').
- An eval expression also uses an existing key from an event.
- An invalid expression or syntax mismatch does not generate any exception or error.

Example:

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

The above example calculates the value of *Revenue* by multiplying the values of *unit_sold* and *Selling_price*.

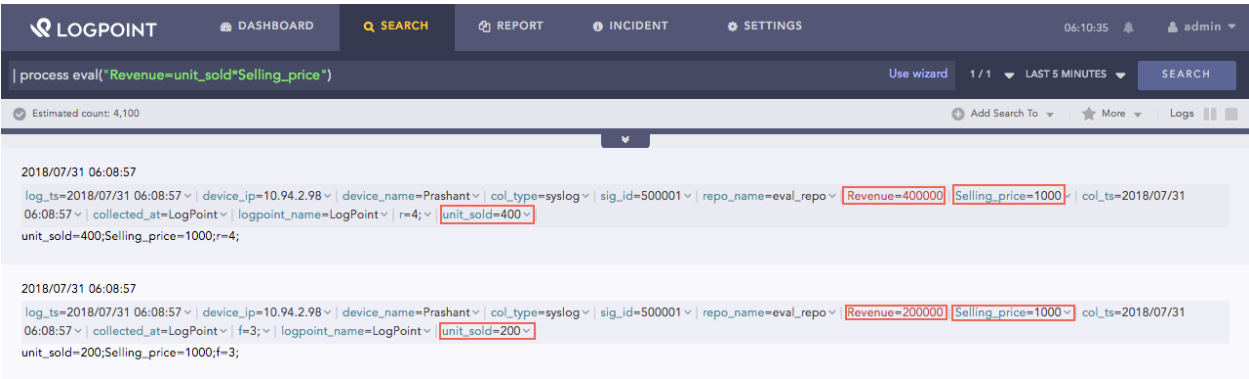


Fig. 1: Eval Expression

ARITHMETIC EXPRESSIONS

An **Arithmetic expression** 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")
```

2.1 Addition (+)

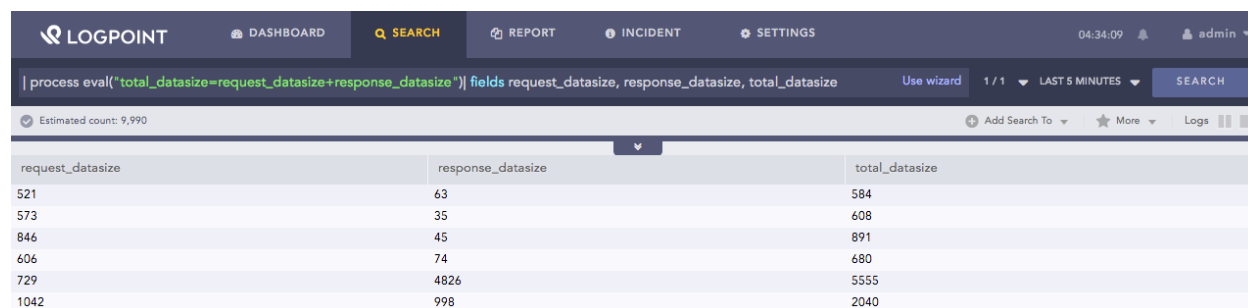
This function accepts numerical values as inputs for **addition** and generates the output in the destination field (identifier).

Example:

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

The above example calculates the value of the *total_datasize* identifier by adding the values of the *request_datasize* and *response_datasize* fields.

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



The screenshot shows the Logpoint interface with a search bar containing the command: `| process eval("total_datasize=request_datasize+response_datasize") | fields request_datasize, response_datasize, total_datasize`. Below the search bar, a table displays the results of the command. The table has three columns: *request_datasize*, *response_datasize*, and *total_datasize*. The data rows are as follows:

request_datasize	response_datasize	total_datasize
521	63	584
573	35	608
846	45	891
606	74	680
729	4826	5555
1042	998	2040

Fig. 1: Addition function

Note: The **Addition** function also accepts string values as inputs and returns the concatenation of values as the output.

2.2 Subtraction (-)

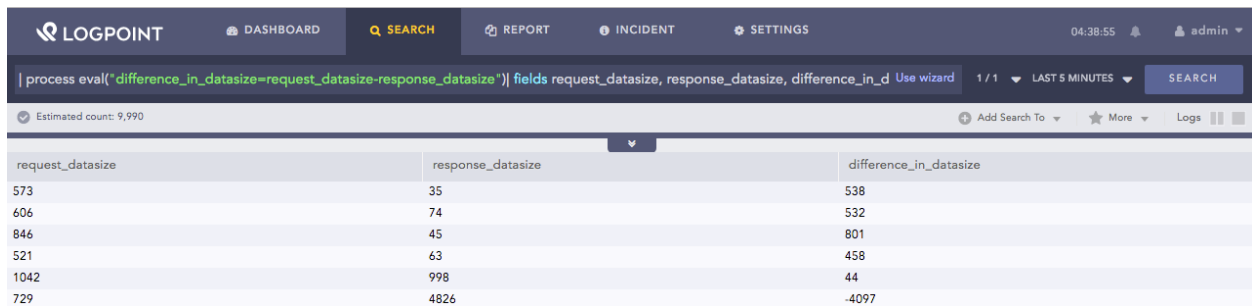
This function accepts numerical values as inputs for **subtraction** and generates the output in the destination field (identifier).

Example:

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

The above example 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 the three fields in a tabular form.



The screenshot shows the Logpoint dashboard with a search bar containing the query: `| process eval("difference_in_datasize=request_datasize-response_datasize") | fields request_datasize, response_datasize, difference_in_d Use wizard`. Below the search bar, a table displays the results of the subtraction function. The table has three columns: *request_datasize*, *response_datasize*, and *difference_in_datasize*. The data rows are as follows:

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: Subtraction function

2.3 Multiplication (*)

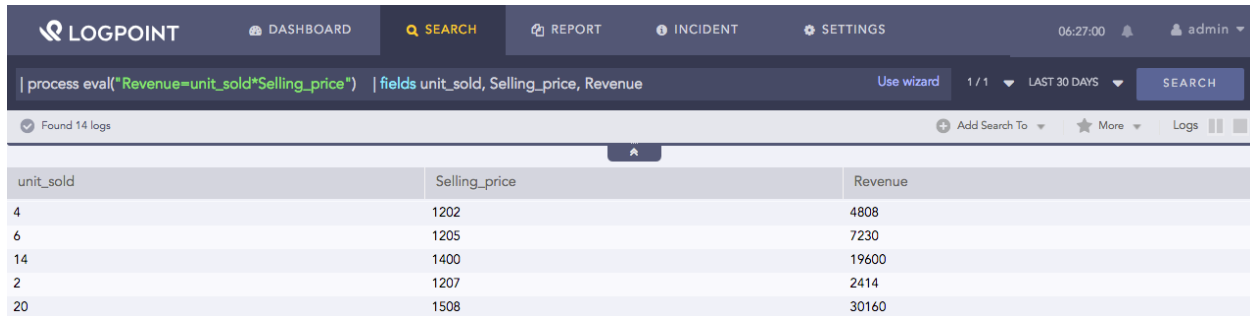
This function accepts numerical values as inputs for **multiplication** and generates the output in the destination field (identifier).

Example:

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

The above example 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.



The screenshot shows the Logpoint interface with a search bar containing the query `| process eval("Revenue=unit_sold*Selling_price") | fields unit_sold, Selling_price, Revenue`. Below the search bar, a table displays the results of the multiplication function.

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

Fig. 3: Multiplication function

2.4 Division (/)

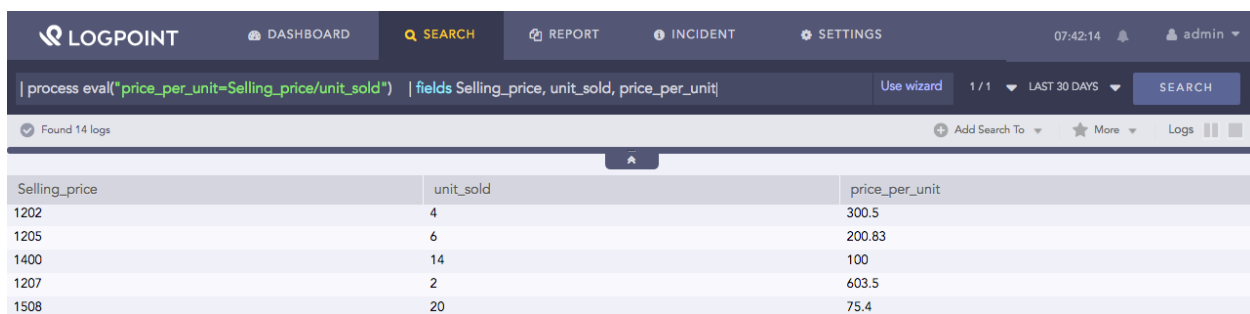
This function accepts numerical values as inputs for the **division** and generates the output in the destination field (identifier).

Example:

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

The above example 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.



The screenshot shows the Logpoint interface with a search bar containing the query `| process eval("price_per_unit=Selling_price/unit_sold") | fields Selling_price, unit_sold, price_per_unit`. Below the search bar, a table displays the results of the division function.

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: Division function

2.5 Modulus (%)

This function accepts numerical values as inputs 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
```

The above example 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.



Selling_price	cost_price	remainder
1202	900	302
1205	1000	205
1400	1400	0
1207	1300	1207
1508	1000	508

Fig. 5: Modulus function

2.6 Power (^)

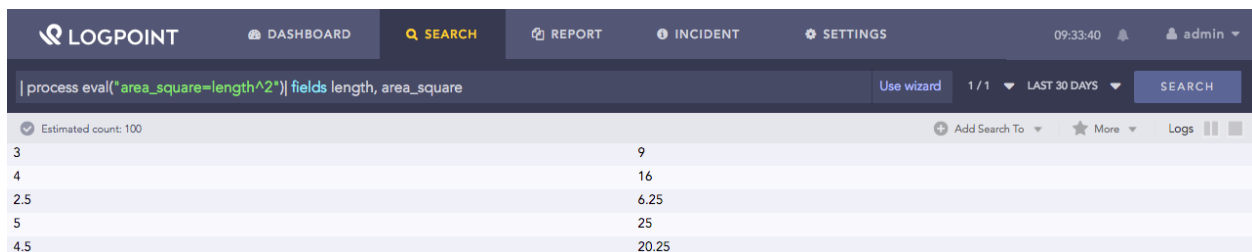
This function accepts numerical values as inputs for the **power** operation and generates the output in the destination field (identifier).

Example:

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

The above example 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.



length	area_square
3	9
4	16
2.5	6.25
5	25
4.5	20.25

Fig. 6: Power function

RELATIONAL EXPRESSIONS

The **Relational expression** is an expression that returns a boolean value, i.e., *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")
```

3.1 Less than (<)

This function compares two operands. It returns *True* if the first operand is **less than** the second operand, else returns *False*.

Example:

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

The above example compares the values of the *cost_price* and *Selling_price* fields. It returns *true* in the *is_loss* identifier if the *cost_price* is less than the *Selling_price*, else returns *false*.

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

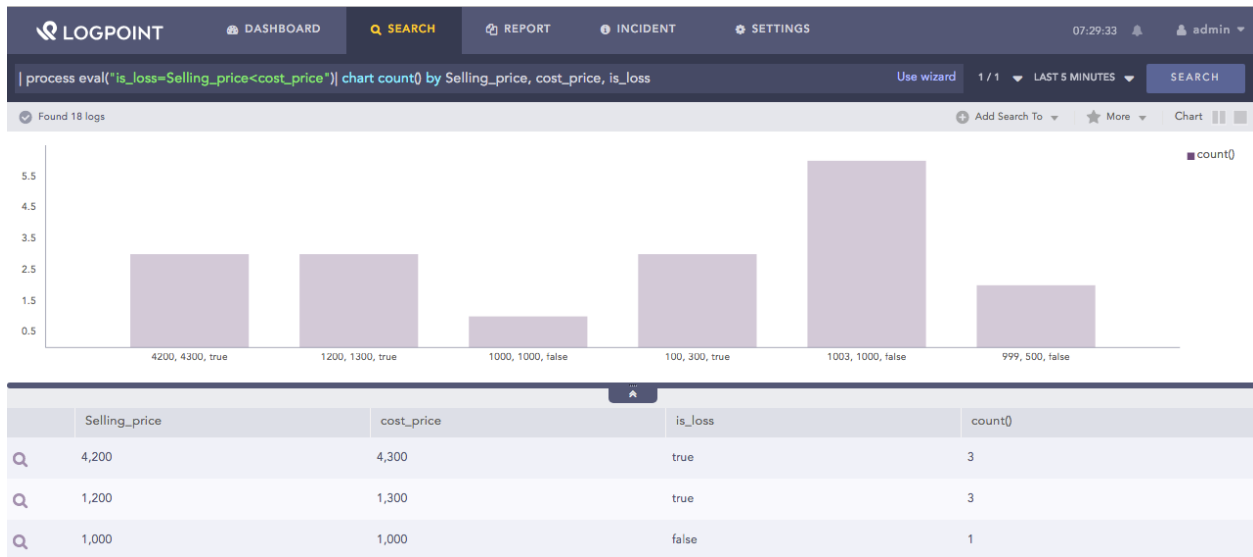


Fig. 1: Less than function

3.2 Greater than (>)

This function compares two operands. It returns *True* if the first operand is **greater than** the second operand, else returns *False*.

Example:

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

The above example compares the values of 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*, else 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.

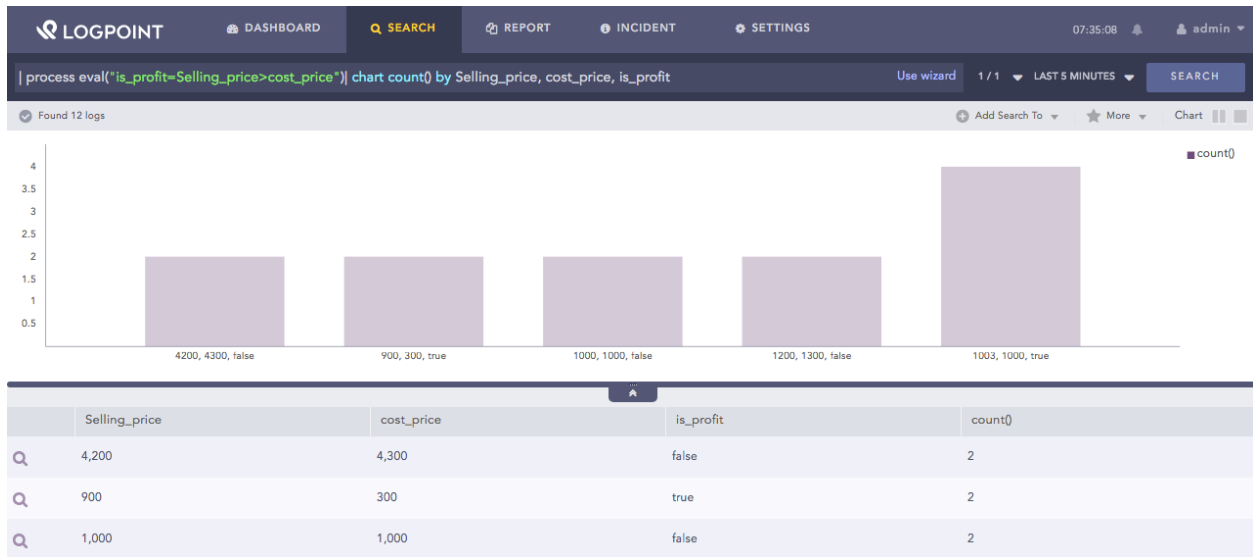


Fig. 2: Greater than function

3.3 Less than or equal to (<=)

This function compares two operands. It returns *True* if the first operand is **less than or equal to** the second operand, else returns *False*.

Example:

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

The above example compares the value of the *discount* field with 50. It returns *true* in the *is_less_discount* identifier if the *discount* is less than or equal to 50, else returns *false*.

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

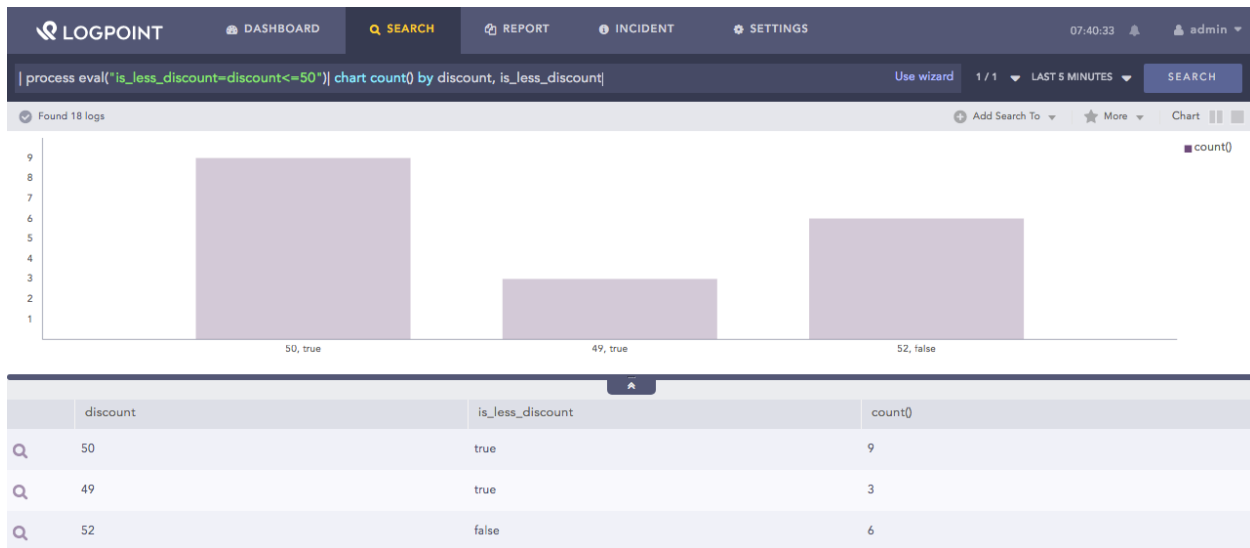


Fig. 3: Less than or equals function

3.4 Greater than or equal to (>=)

This function compares two operands. It returns *True* if the first operand is **greater than or equal to** the second operand, else returns *False*.

Example:

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

The above example compares the value of the *discount* field with 51. It returns *true* in the *is_more_discount* identifier if the *discount* is greater than or equal to 51, else returns *false*.

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

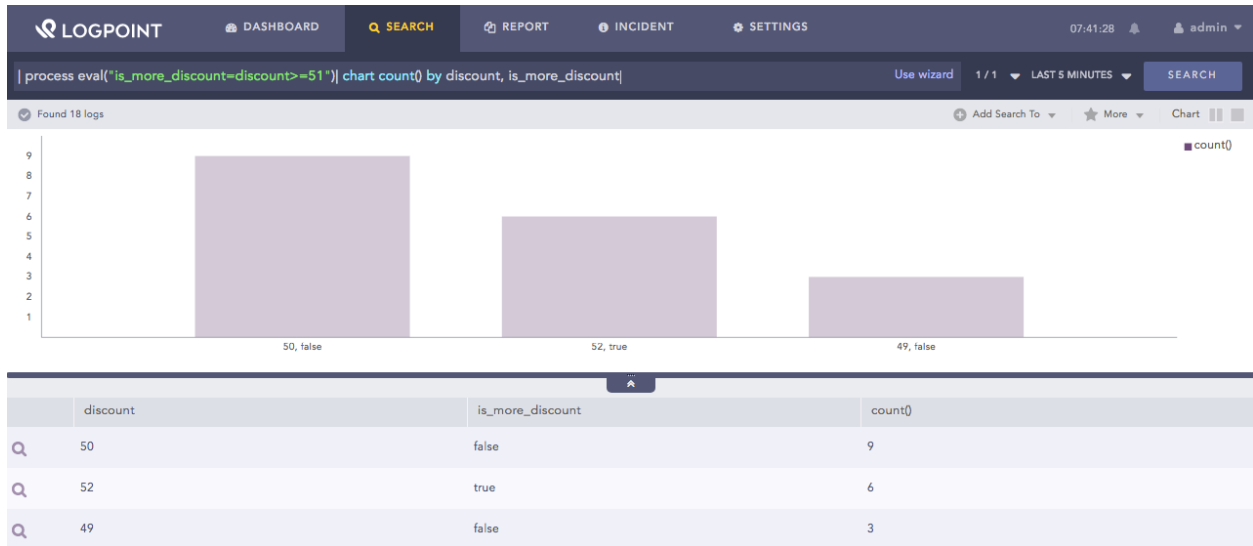


Fig. 4: Greater than or equals function

3.5 Not equal to (!=)

This function compares two operands. It returns *True* if the first operand is **not equal** to the second operand, else returns *False*.

Example:

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

The above example compares the value of the *Selling_price* field 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*, else returns *false*.

The *chart count()* command 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.

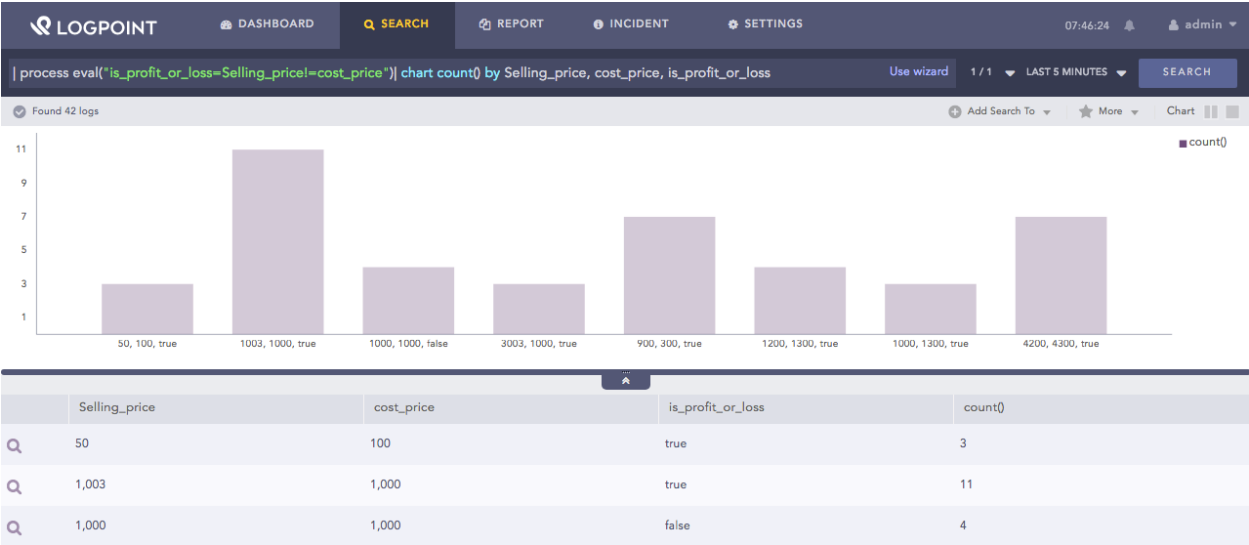


Fig. 5: Not equals function

3.6 Equal to (==)

This function compares two operands. It returns *True* if the first operand is **equal to** the second operand, else returns *False*.

Example:

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

The above example compares the value of the `Selling_price` field 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`, else returns *false*.

The `chart count()` command 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.

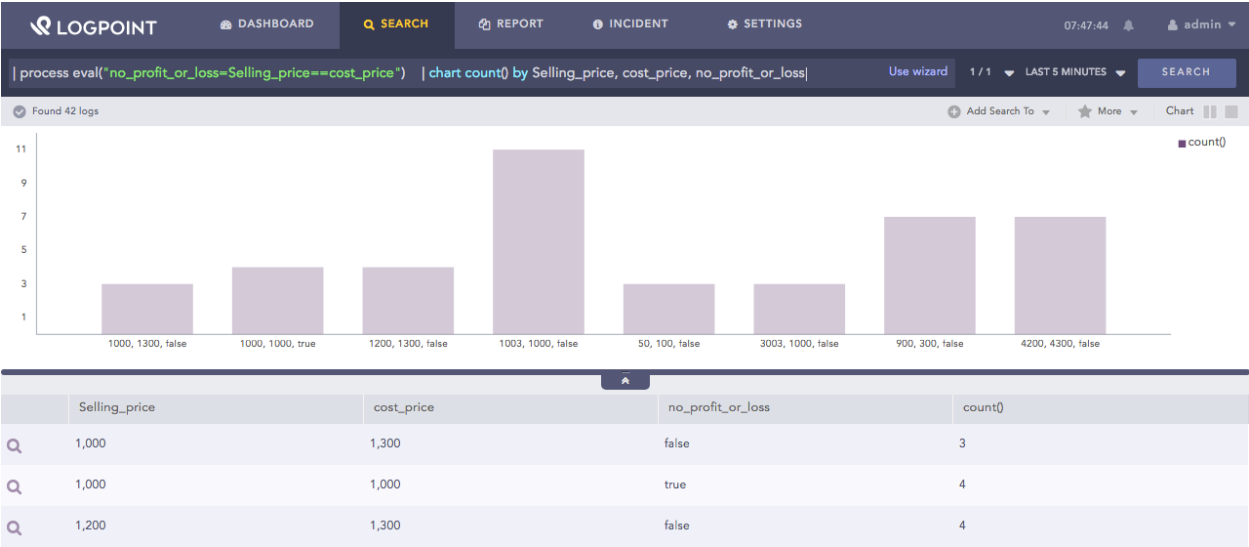


Fig. 6: Equals function

LOGICAL EXPRESSIONS

The **Logical expression** compares two boolean values and returns either *True* or *False*.

Generic syntax:

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

4.1 AND (&&)

This function compares two boolean values. It returns *True* if both both the values are true. If they are not 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
```

The above example 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, else 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.

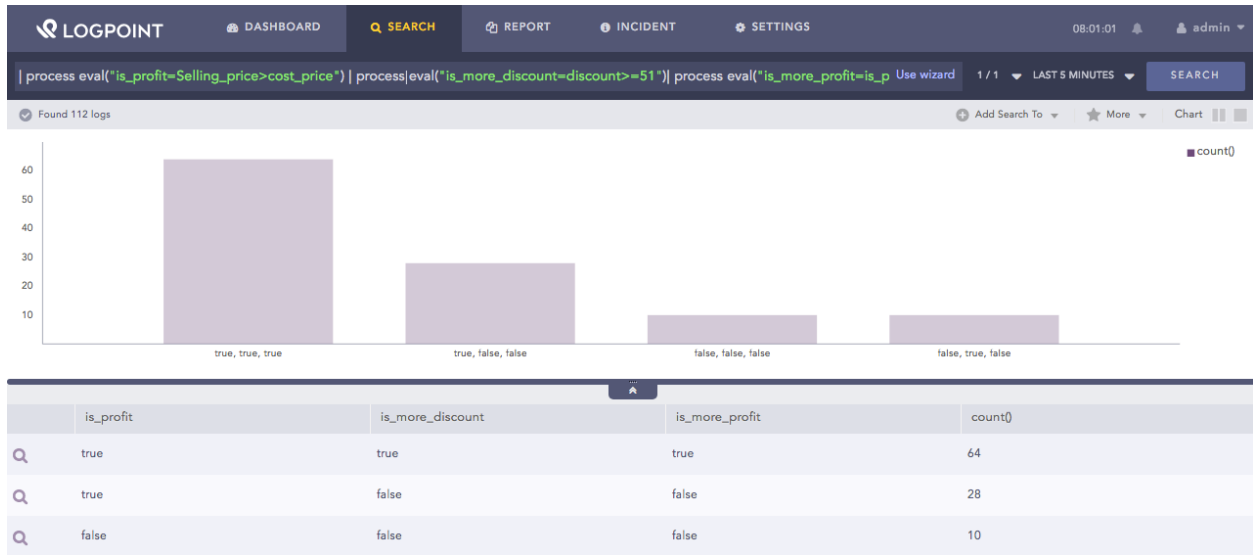


Fig. 1: AND function

4.2 OR (||)

This function 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
```

The above example 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*, else 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.

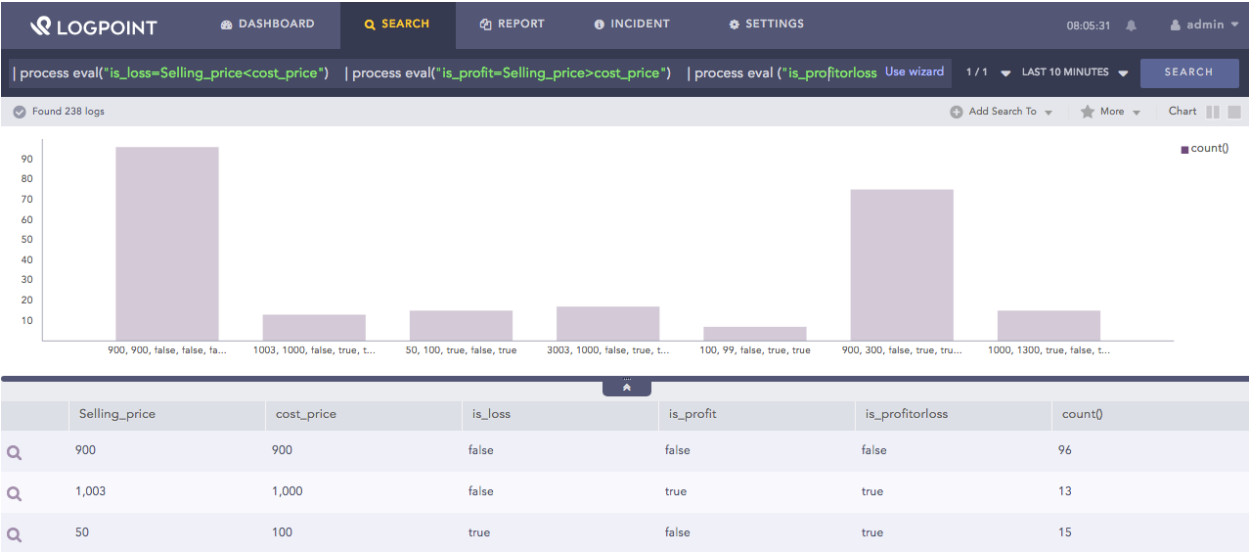


Fig. 2: OR function

CONDITIONAL AND COMPARISON FUNCTIONS

The **Conditional and Comparison functions** evaluate conditions and compare values.

5.1 If Statement

This function accepts a *condition* and a string value *X*. It compares the given *condition*; if the *condition* is true, it returns the value provided in *X*.

Syntax:

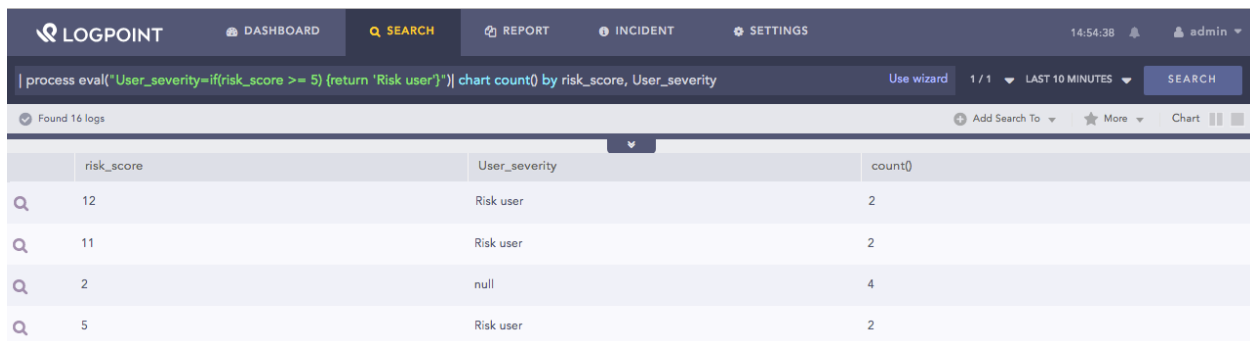
```
| process eval("identifier=if(condition) {return X}")
```

Example:

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

The above example returns *Risk user* in the *User_severity* identifier if the value of the *risk_score* field is greater than or equal to 5.

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.



	risk_score	User_severity	count()
Q	12	Risk user	2
Q	11	Risk user	2
Q	2	null	4
Q	5	Risk user	2

Fig. 1: If statement function

5.2 If-else Statement

This function accepts a condition and two strings *X* and *Y*. It compares the given *condition*; if the *condition* is true, it returns *X*, else returns *Y*.

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
```

The above example 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, else returns *profit/loss*.

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

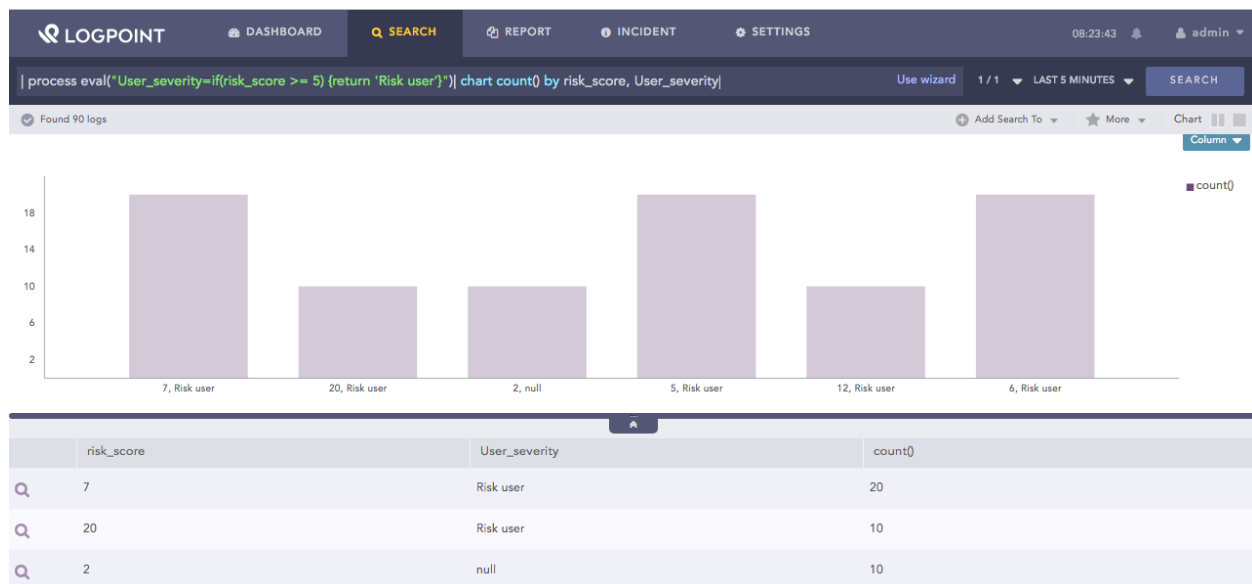


Fig. 2: If-else statement function

5.3 If-elseif-else Statement

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

- if the first *condition* is true, it returns the value provided in X,
- else compares the second *condition*; if the second *condition* is true, it returns the value provided in Y,
- else returns the value provided in Z.

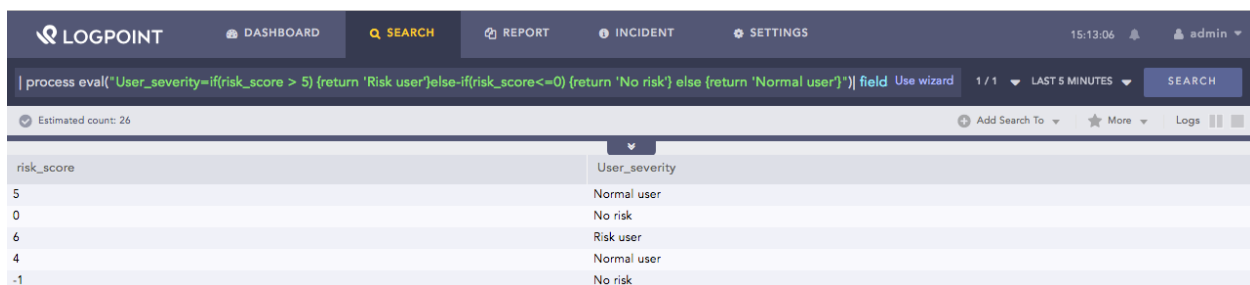
Syntax:

```
| 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
```

The above example checks if the value of the *risk_score* field is greater than 5. It returns *Risk user* in the *User_severity* identifier if the condition is true, else it compares the second condition, i.e., it checks if the value of the *risk_score* field 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.



risk_score	User_severity
5	Normal user
0	No risk
6	Risk user
4	Normal user
-1	No risk

Fig. 3: If-elseif-else statement function

5.4 Case Statement

This function 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,
- else checks if the *case_two* matches the value of the *data*; if the condition is true, it returns the value in Y,
- else returns the value in Z by default.

Syntax:

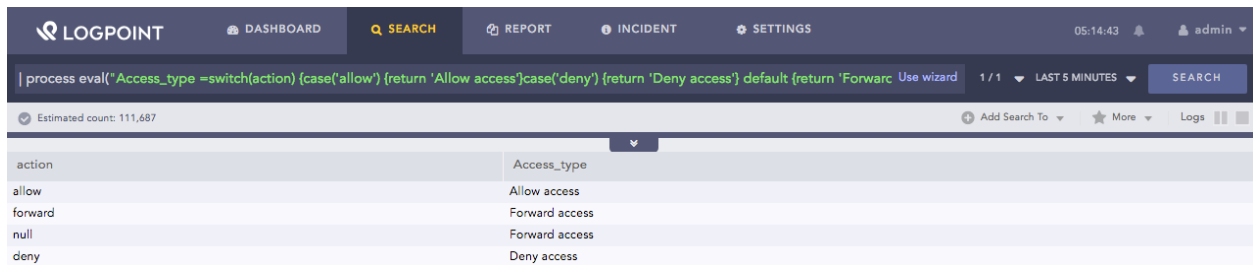
```
| process eval("identifier=switch(data) {case(case_one) {return X}
case(case_two) {return Y} default {return Z}}")
```

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
```

The above example returns *Allow access* in the *Access_type* identifier if the the value of the *action* field is *access*; Else if the value of *action* is *deny*; it returns *Deny access*, else it returns *Forward access* by default.

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



action	Access_type
allow	Allow access
forward	Forward access
null	Forward access
deny	Deny access

Fig. 4: Case statement function

5.5 cidrmatch

This function 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, else returns *False*.

Syntax:

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

Example:

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

The above example returns *true* in the *is_local_ip* identifier if the value of the field *device_ip* matches the CIDR notation 127.0.0.0/8, else returns *false*.

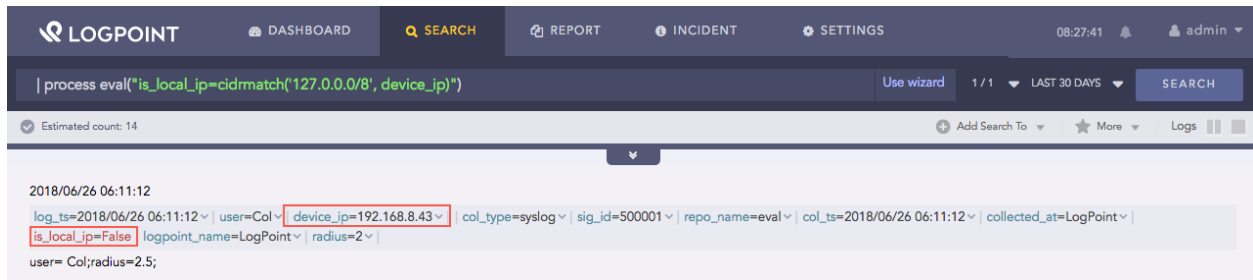


Fig. 5: Cidrmatch function

5.6 coalesce

This function accepts an arbitrary number of arguments as inputs 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
```

The above example returns the value of the *ip_address* field in the *ip_add* identifier if the its value is not null. If null, it checks the value of the *device_ip* field. If the *device_ip* field 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.

ip_address	device_ip	ip_add
134.38.23.45	10.94.2.98	134.38.23.45
null	10.94.2.98	10.94.2.98
152.66.265.36	10.94.2.98	152.66.265.36
null	10.94.2.98	10.94.2.98
192.178.2.3	10.94.2.98	192.178.2.3
192.178.2.3	10.94.2.98	192.178.2.3

Fig. 6: Coalesce function

5.7 false

This function returns *False*. The *false* function in combination with other functions represents a condition that is undoubtedly false, i.e., $1==0$. 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
```

The above example 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*, else 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.

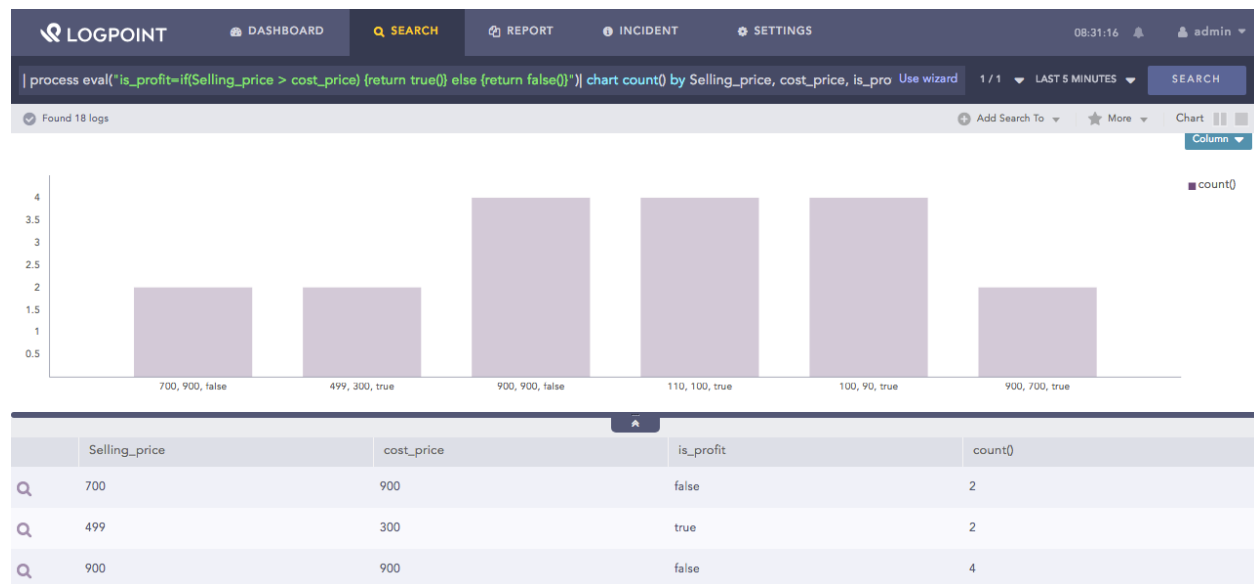


Fig. 7: False function

5.8 in

This function 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*, else 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
```

The above example returns *true* in the *isUserAdmin* identifier if the value in the *user* field matches with any one value in the list, i.e., *Administrator*, *administrator*, *Admin*, and *admin*, else 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.

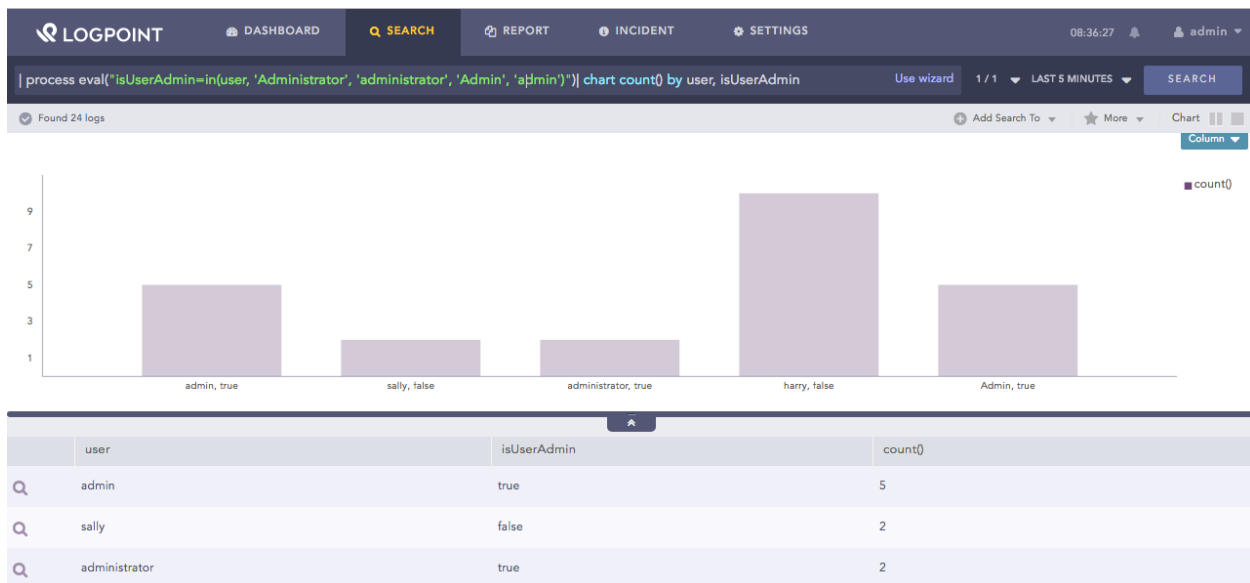


Fig. 8: In function

5.9 match

This function 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*.

This function 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
```

The above example 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 value of `col_type` field, else 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.

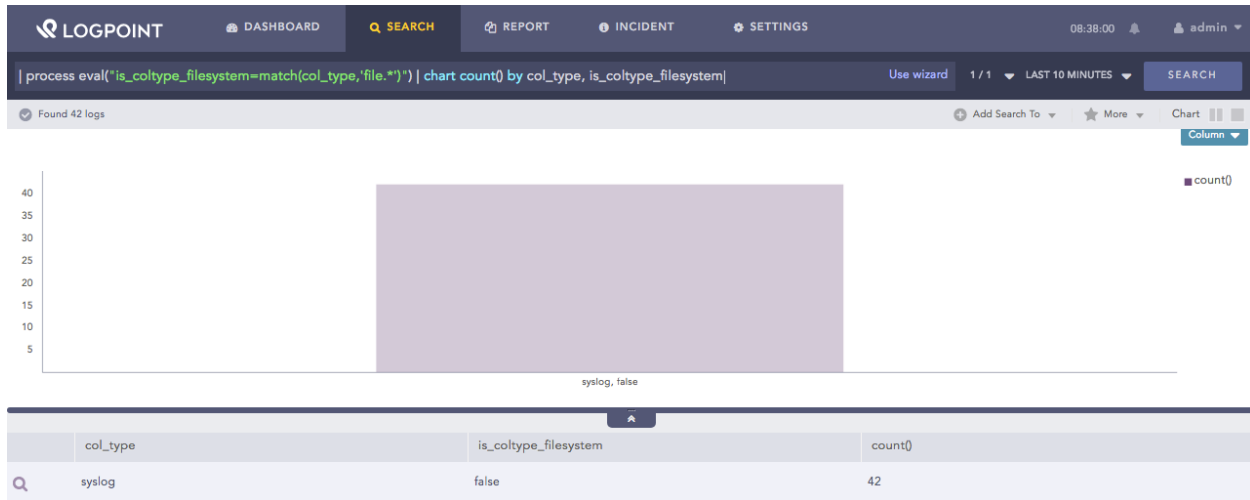


Fig. 9: Match function

5.10 like

This function accepts a text field *X* and a *pattern*. It returns *True* if the text in the field *X* matches the given *pattern*, else returns *False*. This function also returns *True* if the text in the pattern exactly matches the text in the field *X*.

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
```

The above example compares the `sys%` pattern with the value in the `col_type` field. It returns *true* in the `is_coltype_filesystem` identifier if the `sys%` pattern is an exact match

or is a substring of the value of `col_type` field, else 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.

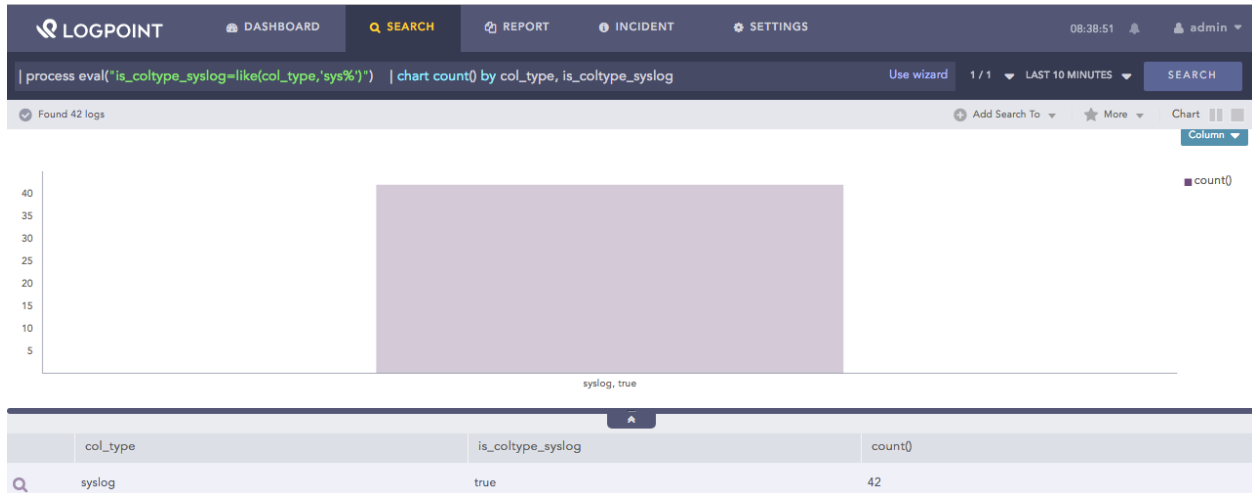


Fig. 10: Like function

5.11 null

This function returns null. You use the null function in combination with other functions. You use this function in case 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
```

The above example returns *null* in the `User_severity` identifier if the value of the `score` field is less or equal to 5, else 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.

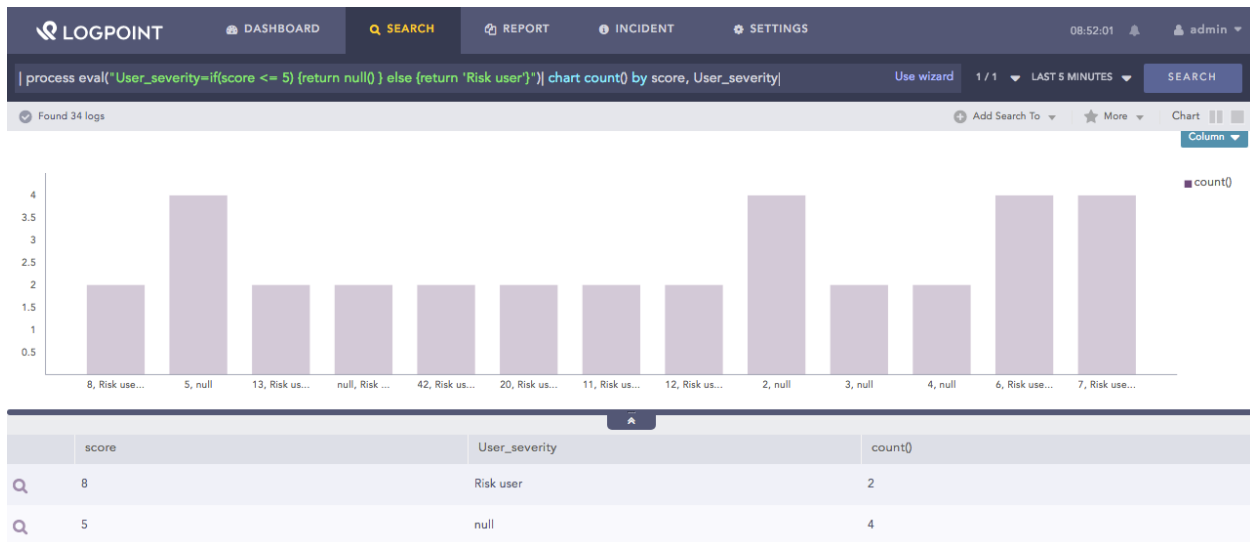


Fig. 11: Null function

5.12 nullif

This function compares two arguments, *X* and *Y*. If *X* = *Y*, it returns null, else 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
```

The above example returns *null* in the *access_type* identifier if the value of the *access* field is *DELETE*, else 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.

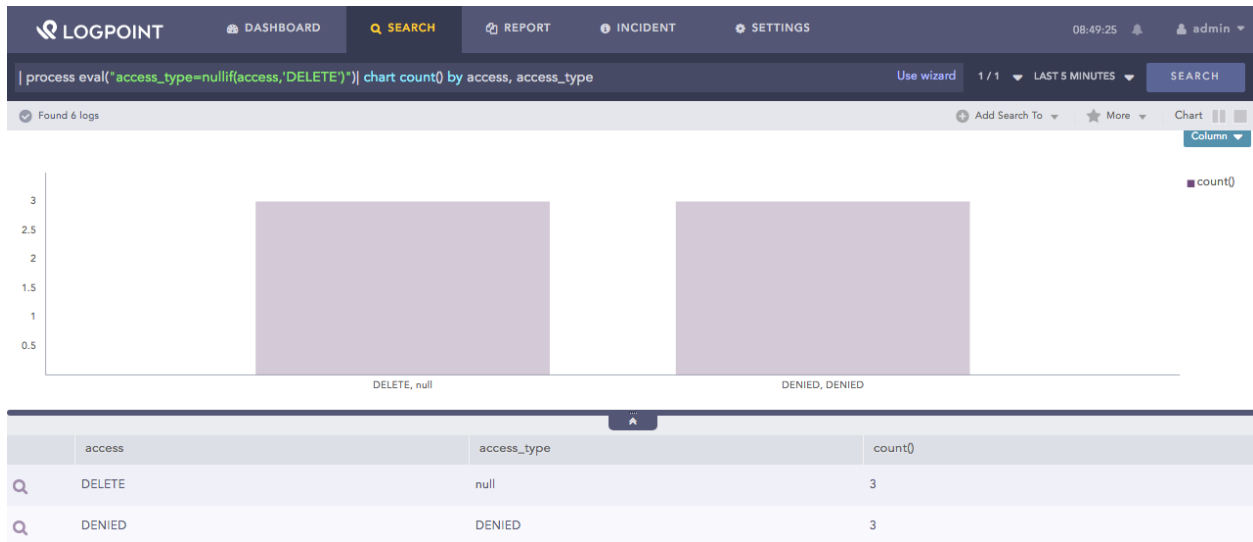


Fig. 12: Nullif function

5.13 searchmatch

This function accepts a string field *X* as input. It returns *True* if the value of *X* matches the event type, else returns *False*. You can use the pipe (|) symbol to separate multiple values of *X*.

Syntax:

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

Example:

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

The above example returns *null* in the *is_authentication_event* identifier if the value of the *access* field is *DELETE*, else returns *false*.

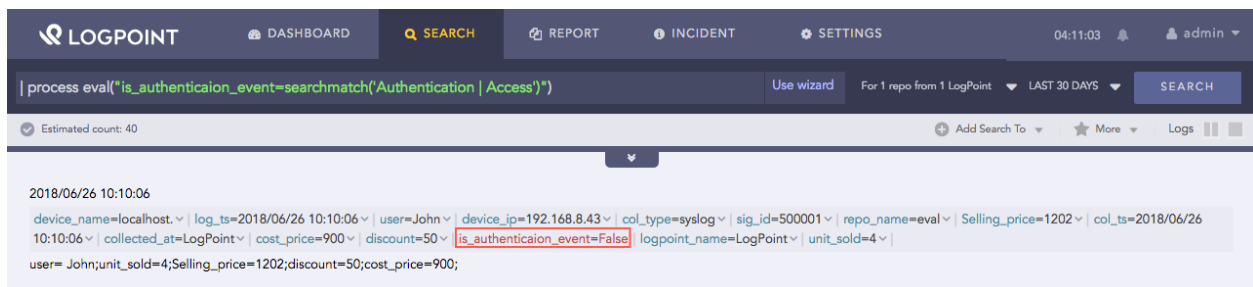


Fig. 13: Searchmatch function

5.14 true

This function returns *True*. It is often used in combination with other functions to represent a condition that is undoubtedly true, i.e., $1==1$. 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
```

The above example returns *true* in the *is_profit* identifier if the value of the *Selling_price* field is greater than the value of the *cost_price* field, else 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.

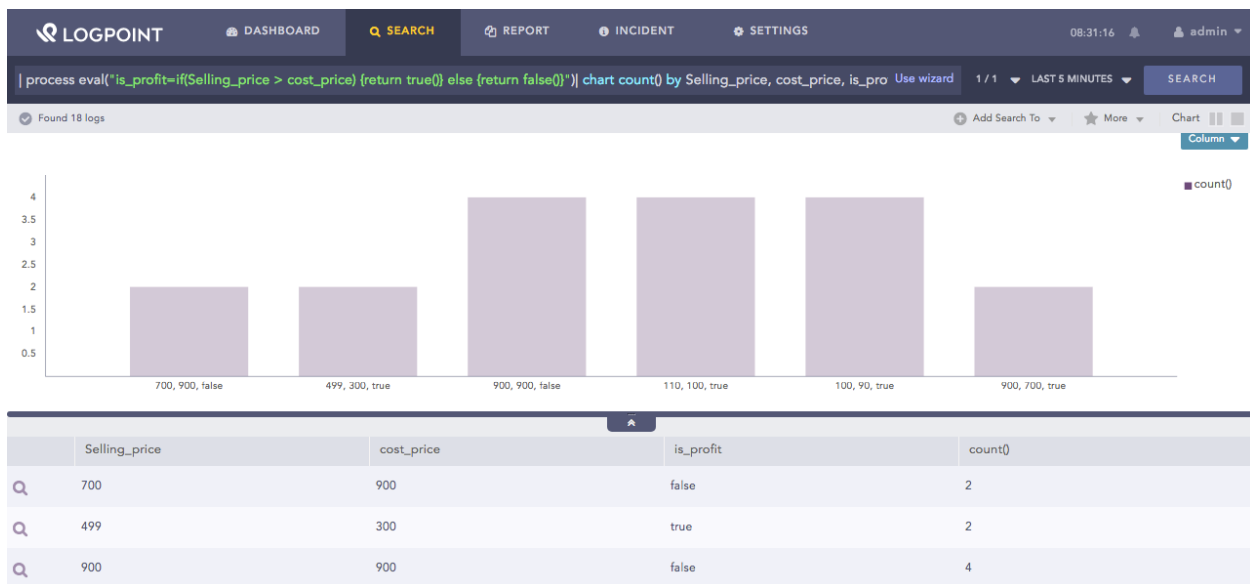


Fig. 14: True function

MULTIVALUE FUNCTIONS

The **Multivalue functions** evaluate multivalue arguments and return multivalue fields.

6.1 split

This function accepts two arguments, *X* and *Y* as inputs. It splits the values of the field *X* by the delimiter (such as comma, semicolon, blank space) specified in the field *Y* and returns a multivalue field containing a list of split values.

Syntax:

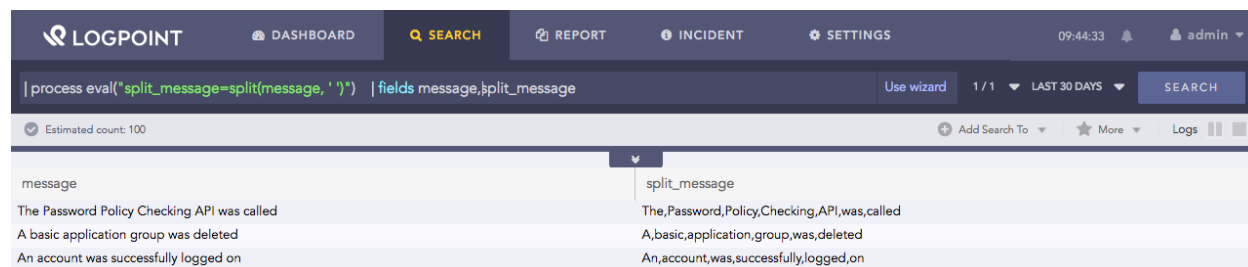
```
| process eval("identifier=split(X,Y)")
```

Example:

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

The above example splits the value of the *message* field when the *split* function encounters 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.



The screenshot shows the Logpoint dashboard interface. At the top, there is a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The SEARCH tab is active. Below the navigation bar, a search query is entered: `| process eval("split_message=split(message, ' ')") | fields message, split_message`. The results are displayed in a table with two columns: *message* and *split_message*. The table contains three rows of data, each representing a log entry that has been split by spaces.

message	split_message
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. 1: Split function

6.2 commands

This function accepts a search string *X* as input, 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
```

The above example returns the list of the commands present in the provided search string, i.e., *chart count() | fields name | rename message as alert*, and returns them in the identifier *commands_list*.

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

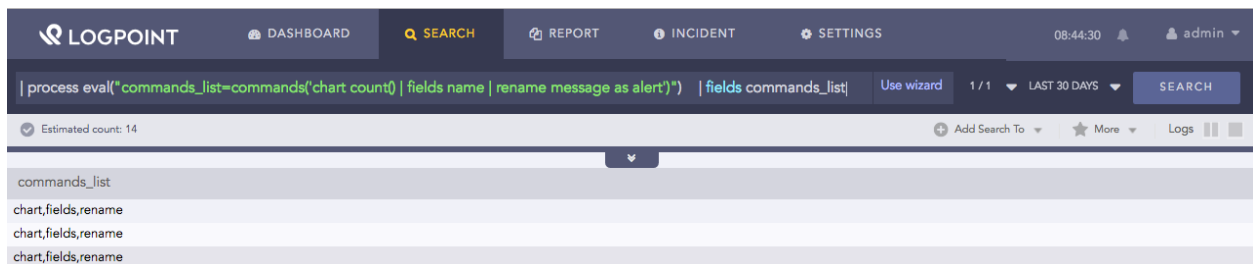


Fig. 2: Commands function

6.3 mvappend

This function accepts two or more arguments as inputs 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:

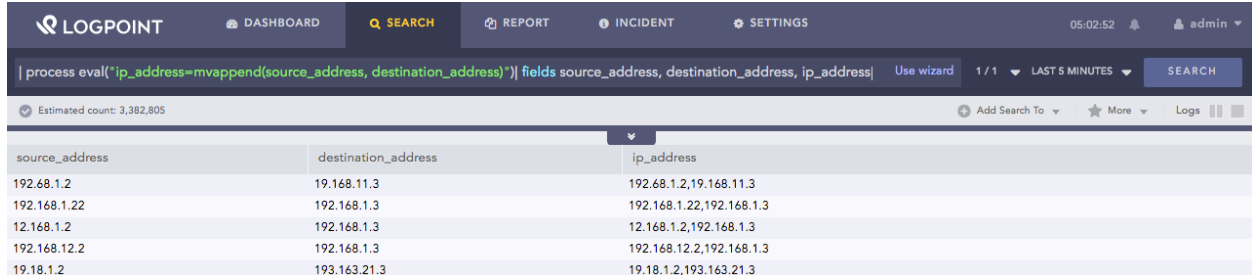
```
| process eval("identifier=mvappend(X, ...)")
```

Example:

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

The above example returns the appended value of the `source_address` 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.



The screenshot shows the Logpoint Search interface with a search query: `| process eval("ip_address=mvappend(source_address, destination_address)") | fields source_address, destination_address, ip_address`. The results table has three columns: `source_address`, `destination_address`, and `ip_address`. The estimated count is 3,382,805.

source_address	destination_address	ip_address
192.68.1.2	19.168.11.3	192.68.1.2,19.168.11.3
192.168.1.22	192.168.1.3	192.168.1.22,192.168.1.3
12.168.1.2	192.168.1.3	12.168.1.2,192.168.1.3
192.168.12.2	192.168.1.3	192.168.12.2,192.168.1.3
19.18.1.2	193.163.21.3	19.18.1.2,193.163.21.3

Fig. 3: Mvappend function

6.4 mvcount

This function accepts either a multivalue field or a single value field `X` as input and returns the count of the values of that field.

Syntax:

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

Example:

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

The above example compares the values obtained from `split(message, ' ')` to the pattern `acco..`. `acco..` means any sequence of letters that can follow the string `acco`. If the values obtained from `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.



The screenshot shows the Logpoint Search interface with a search query: `| process eval("message_character_count=mvcount(split(message, ' '))") | fields message, message_character_count`. The results table has two columns: `message` and `message_character_count`. The estimated count is 100.

message	message_character_count
The Password Policy Checking API was called	7
A basic application group was deleted	6
An account was successfully logged on	6

Fig. 4: Mvcount function

6.5 mvdedup

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

Syntax:

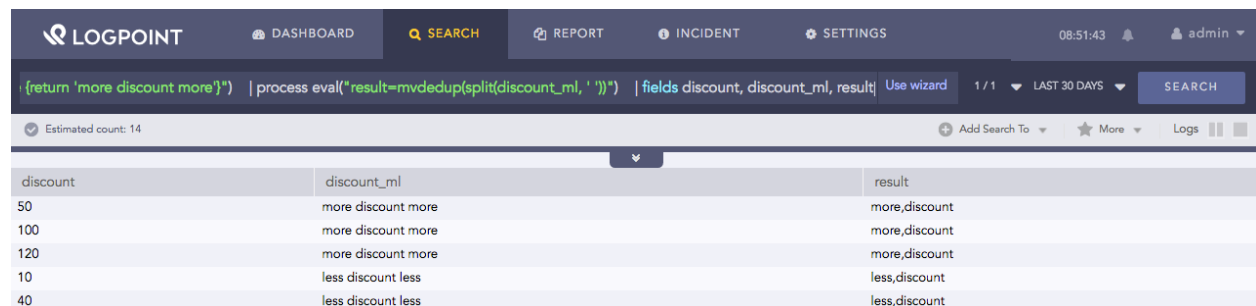
```
| process eval("identifier=mvdedup(X)")
```

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
```

The above example first returns *less discount less* in the *discount_ml* identifier if *discount* is less than 50, else returns *more discount more*. The *split* function splits the value of *discount_ml* when it encounters a space. Then, the *mvdedup* 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.



The screenshot shows the Logpoint dashboard interface. At the top, there's a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The SEARCH tab is active. Below the navigation bar, a search query is entered: `(return 'more discount more') | process eval("result=mvdedup(split(discount_ml, ' '))") | fields discount, discount_ml, result`. The results are displayed in a table with three columns: *discount*, *discount_ml*, and *result*. The table contains five rows of data.

discount	discount_ml	result
50	more discount more	more,discount
100	more discount more	more,discount
120	more discount more	more,discount
10	less discount less	less,discount
40	less discount less	less,discount

Fig. 5: Mvdedup function

6.6 mvfilter

This function accepts a multivalue field *X* and a *pattern* as inputs. It filters the values of the field using the *pattern*, and returns a multivalue or a single value field containing the list of values that match the given *pattern*.

Syntax:

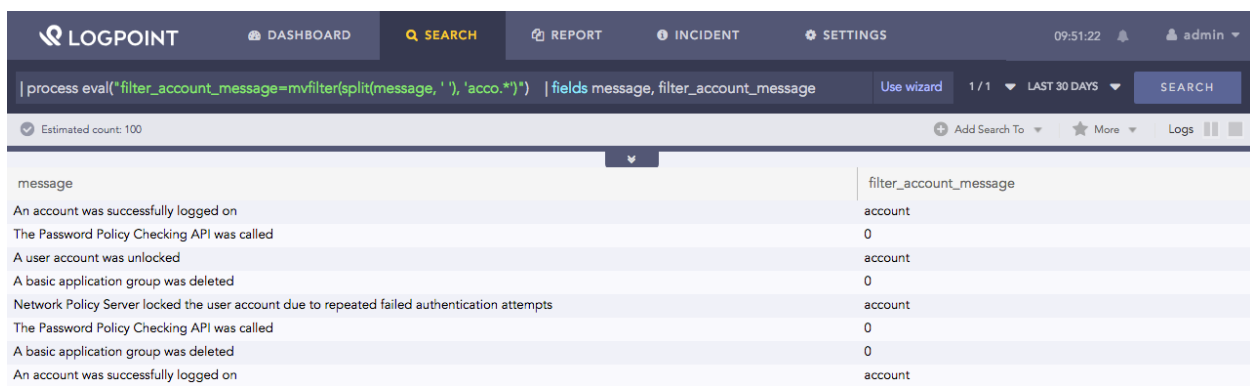
```
| process eval("identifier=mvfilter(X, pattern)")
```

Example:

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

The above example compares the values obtained from `split(message, ' ')` to the pattern `acco.*`. `acco.*` mean 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, else returns 0. Refer to the [split](#) section to know on the value from `split(message, ' ')`.

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



message	filter_account_message
An account was successfully logged on	account
The Password Policy Checking API was called	0
A user account was unlocked	account
A basic application group was deleted	0
Network Policy Server locked the user account due to repeated failed authentication attempts	account
The Password Policy Checking API was called	0
A basic application group was deleted	0
An account was successfully logged on	account

Fig. 6: Mvfilter function

6.7 mvfind

This function accepts two arguments, a multivalue field `X`, and a regex (regular expression) pattern `Y`. It tries to match the regular expression against any substring of the value in `X`. 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)")
```

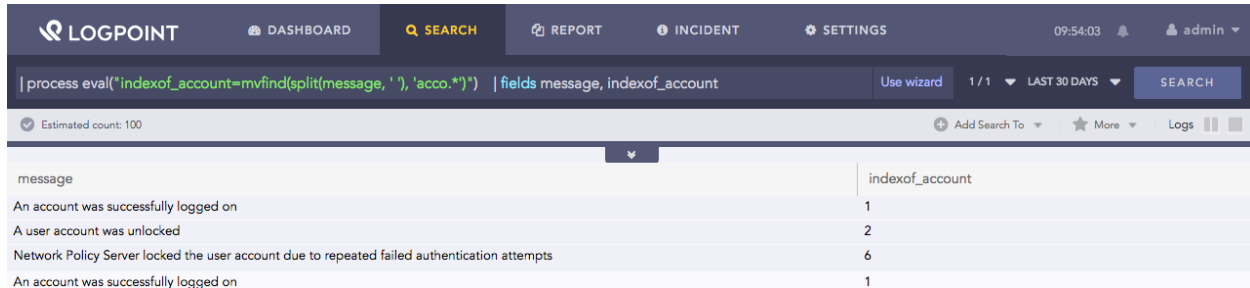
Example:

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

The above example search for the first match in the values obtained from `split(message, ' ')` to the pattern `acco.*`. `acco.*` mean any sequence of letters can follow the string `acco`.

If the function finds a match in the values obtained from `split(message, ' ')` to the pattern `acco.*`, 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.



message	indexof_account
An account was successfully logged on	1
A user account was unlocked	2
Network Policy Server locked the user account due to repeated failed authentication attempts	6
An account was successfully logged on	1

Fig. 7: Mvfind function

6.8 mvindex

This function accepts up to three arguments, a multivalued field `X`, a number `start_index` and a number `end_index`. It evaluates the values of the string `X` and returns the value that starts at the index specified by `start_index` and ends at the index specified by `end_index`.

- The field `X` 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:

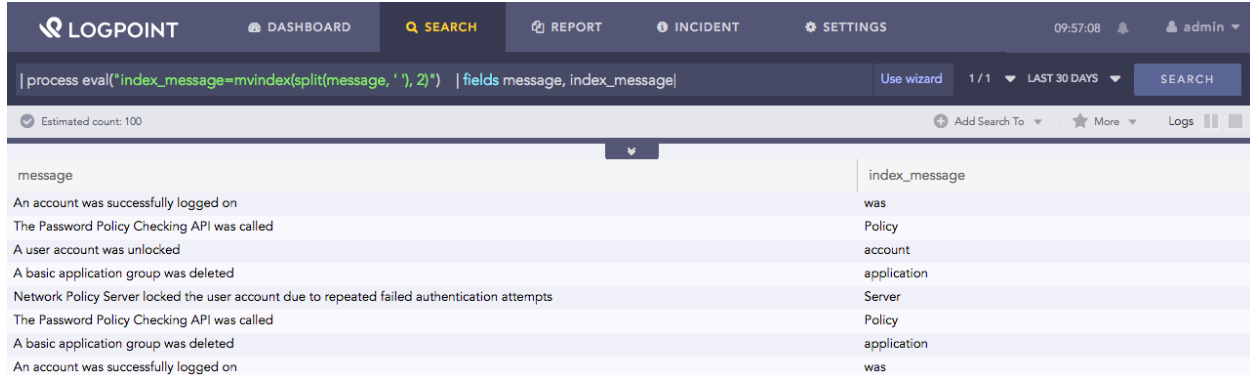
```
| process eval("identifier=mvindex(X, start_index, end_index)")
```

Example:

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

The above example returns the third value (index starts from 0) in the values obtained from `split(message, ' ')` in the `index_message` identifier.

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



The screenshot shows the Logpoint dashboard with a search bar containing the query: `| process eval("index_message=mvindex(split(message, ' '), 2)") | fields message, index_message|`. Below the search bar, a table displays the results of the query. The table has two columns: `message` and `index_message`. The `message` column contains log entries, and the `index_message` column contains the corresponding values extracted from the logs.

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

Fig. 8: Mvindex function

6.9 mvjoin

This function accepts two arguments, a multivalue field `X`, and a string delimiter (such as comma, semicolon, 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
```

The above example accepts the values from `split(message, ' ')` and joins the received values with a comma. It returns the joined values in the `result` identifier. Refer to the [split](#) section 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: Mvjoin function

6.10 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)")
```

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

log_ts	user	device_ip	col_type	sig_id	repo_name	col_ts	collected_at	range
2018/06/26 06:11:12	Col	192.168.8.43	syslog	500001	eval	2018/06/26 06:11:12	LogPoint	[1, 2, 3, 4]

Fig. 10: Mvrange function

Example 2:

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

The above example returns the value from 1.1 to 5 with +1 increment in the *mvrangle* identifier. The result is 1.1, 2.1, 3.1, 4.1.

Example 3:

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

The above example returns the value from 1.5 to 6 (excluding 6) with +1.5 increment in the *mvrangle* identifier. The result is 1.5, 3, 4.5.

Example 4:

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

The above example returns the value from 1134 to 343434 with +86400 increment in the *mvrangle* identifier. The result is 1134, 87534, 173934, 260334.

Here, 1d = 86400 seconds

Example 5:

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

The above example returns the value from 1233.124224 to 2434455.1232323 with +604800 increment in the *mvrangle* identifier. The result is 1233.124224, 606033.124224, 1210833.1242240001, 1815633.1242240001, 2420433.124224.

Here, 1w = 604800 seconds

6.11 mvsort

This function accepts a multivalue field *X* and returns a multivalue field containing the list of values of the field *X* sorted in lexicographical order (alphabetically).

- In lexicographical order, numbers are sorted by digits. So numbers are sorted 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 also after 1000) because 0 is less than 1. A lexical sort compares the characters in each string as characters, not integral values.
- All the uppercase letters are sorted before the lowercase letters.

Syntax:

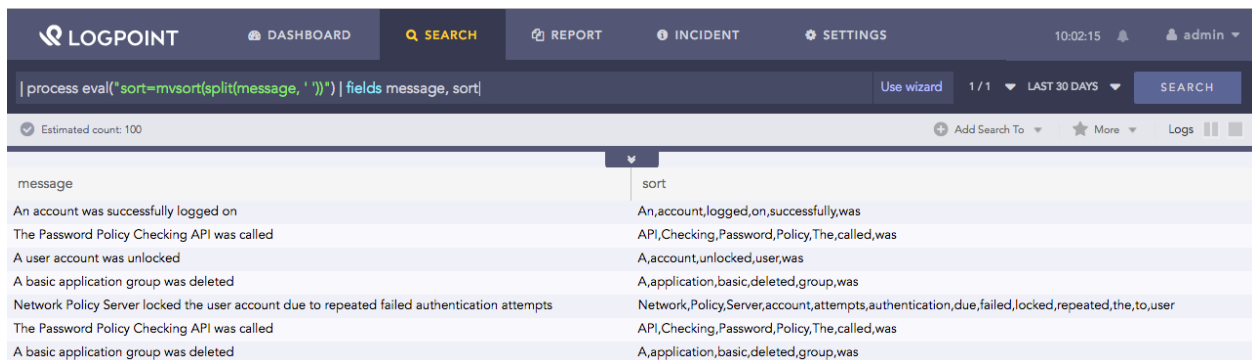
```
| process eval("identifier=mvsort(X)")
```

Example 1:

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

The above example accepts the values from `split(message, ' ')` and sorts them according to the rule described above. 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.



The screenshot shows the Logpoint dashboard with a search bar containing the query: `| process eval("sort=mvsort(split(message, ' '))" | fields message, sort)`. Below the search bar, there is a table with two columns: `message` and `sort`. The table contains several rows of log messages and their corresponding sorted values.

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

Fig. 11: Mvsort function

Example 2:

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

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

The above example sorts the data in the `testData` field according to the rule described above and returns the sorted value in the identifier `sort`. The result is `[1, 1.2, "test1"]`

6.12 mvzip

This function accepts up to three arguments. It takes two multivalue fields, `X` and `Y` and combines the values of these fields. It joins the first value of the `X` field with the first value of the `Y` field and the second value of `X` with the second value of `Y`. The third field `Z` specifies a delimiting character that joins the two values of the fields `X` and `Y`. The `Z` field is optional, and the default delimiter is a comma.

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)")
```

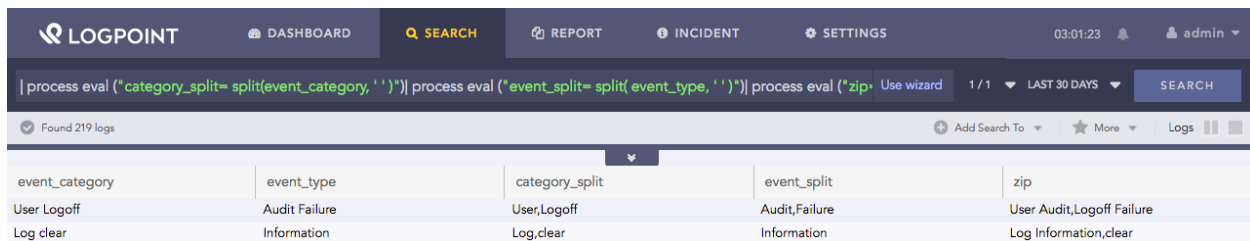
The above example accepts the values from `users` and `machines` fields. It joins the first value of the `user` field with the first value of the `machines` field and the second value of the `users` field with the second value of the `machines` field and continues in the same way. A comma separates the values in the joined fields. It returns the joined value in the `x` identifier. Result: `["john,lp1", "jack,lp2", "kim,lp3"]`

Example 2:

```
| process eval ("category_split= split(event_category, ' ')")
| process eval ("event_split= split(event_type, ' ')")
| process eval ("zip= mvzip(category_split,event_split, ' ')")
| fields event_category, event_type ,category_split, event_split, zip
```

The above example accepts the values of `category_split` and `event_split` fields. It joins the first value of the `category_split` field with the first value of the `event_split` field and the second value of the `category_split` field with the second value of the `event_split` field and continues in the same way. Space separates the values in the joined fields. It returns the joined value in the `zip` identifier. Refer to the [split](#) section to know on the value from `split(event_category, ' ')` and `split(event_type, ' ')`.

The `fields` command displays the value of `event_category`, `event_type`, `category_split`, and `zip` in a tabular form.



The screenshot shows the Logpoint dashboard interface. At the top, there's a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The SEARCH tab is active. Below the navigation bar, a search query is entered: `| process eval ("category_split= split(event_category, ' ')") | process eval ("event_split= split(event_type, ' ')") | process eval ("zip= mvzip(category_split,event_split, ' ')")`. The results show 219 logs found. A table displays the results with columns: event_category, event_type, category_split, event_split, and zip.

event_category	event_type	category_split	event_split	zip
User Logoff	Audit Failure	User,Logoff	Audit,Failure	User Audit,Logoff Failure
Log clear	Information	Log,clear	Information	Log Information,clear

Fig. 12: Mvzip function

STATISTICAL FUNCTIONS

The **Statistical functions** evaluate the maximum and minimum value of numeric or string argument.

7.1 max

This function accepts an arbitrary number of arguments as inputs 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
```

The above example 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.

The screenshot shows the Logpoint interface with a search bar containing the query: `| process eval("Maximum_datsize=max(received_datsize,sent_datsize)") | fields received_datsize, sent_datsize, Maximum_datsize`. The results table has three columns: `received_datsize`, `sent_datsize`, and `Maximum_datsize`. The data rows are as follows:

received_datsize	sent_datsize	Maximum_datsize
456	334	456
85	560	560
999	8797	8797
348	44	348
767	670	767
508	620	620
345	453	453

Fig. 1: Max function

7.2 min

This function accepts an arbitrary number of arguments as inputs and returns the minimum 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=min(X, ...)")
```

Example:

```
| process eval("Minimum_datsize=min(received_datsize,sent_datsize)")
| fields received_datsize, sent_datsize, Minimum_datsize
```

The above example returns the lowest value of the `received_datsize` and `sent_datsize` fields in the `Minimum_datsize` identifier.

The `fields` command displays the value of `received_datsize`, `sent_datsize`, and `Minimum_datsize` in a tabular form.

The screenshot shows the Logpoint interface with a search bar containing the query: `| process eval("Minimum_datsize=min(received_datsize,sent_datsize)") | fields received_datsize, sent_datsize, Minimum_datsize`. The results table has three columns: `received_datsize`, `sent_datsize`, and `Minimum_datsize`. The data rows are as follows:

received_datsize	sent_datsize	Minimum_datsize
456	334	334
85	560	85
999	8797	999
348	44	44
767	670	670
508	620	508
345	453	345

Fig. 2: Min function

CONVERSION FUNCTIONS

The **Conversion functions** convert numbers and strings to different formats.

8.1 printf

This function accepts a string *format* and *arguments* as inputs and returns a formatted string value based on these inputs.

Syntax:

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

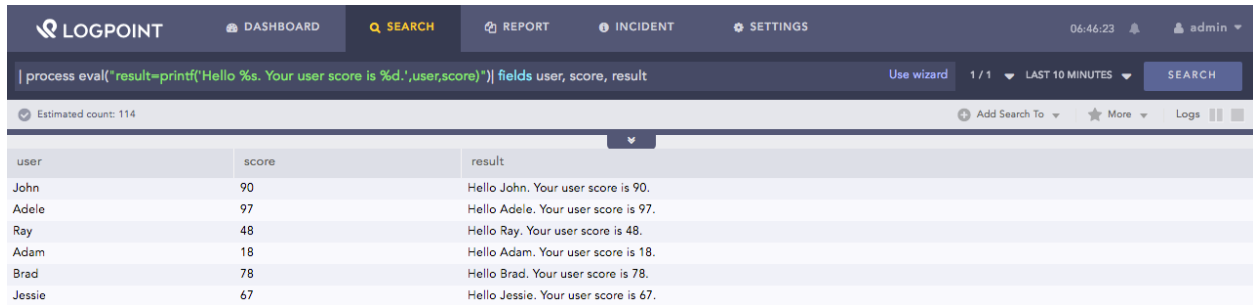
- **format**: The **format** is a character string that comprises one or more format conversion specifiers. The format must always be within single quotes (' ').
- **arguments**: The **arguments** can include one or more string, number, or field name.

Example:

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

The above example 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.



user	score	result
John	90	Hello John. Your user score is 90.
Adele	97	Hello Adele. Your user score is 97.
Ray	48	Hello Ray. Your user score is 48.
Adam	18	Hello Adam. Your user score is 18.
Brad	78	Hello Brad. Your user score is 78.
Jessie	67	Hello Jessie. Your user score is 67.

Fig. 1: Printf function

8.2 tonumber

This function accepts a string value *X*, and a *BASE* as inputs. 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*. *BASE* can range from 2 to 36.

Example:

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

The above example converts the string value *0A5* to a number by taking base 12 and returns it in the *result* identifier.



log_ts=2018/06/26 06:11:12	user=Col	device_ip=192.168.8.43	col_type=syslog	sig_id=500001	repo_name=eval	col_ts=2018/06/26 06:11:12	collected_at=LogPoint	radius=2	result=125
user= Col;radius=2.5;									

Fig. 2: Tonumber function

8.3 tostring

This function 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=tosting(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, i.e., *True* or *False*.

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

Example 1:

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

The above example converts the numeric value of 12 to string.

Example 2:

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

The above example converts the numeric value of the *score* field to its corresponding hexadecimal string value and assigns it in the *result* identifier.



Fig. 3: Tosting function

Example 3:

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

The above example converts the numeric value 65 into the readable time format HH:MM: and returns it in the x identifier. The result is 00:01:05.000.

Example 4:

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

The above example 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. The result is 65,132.68.

EXPRESSION WITH PARENTHESES

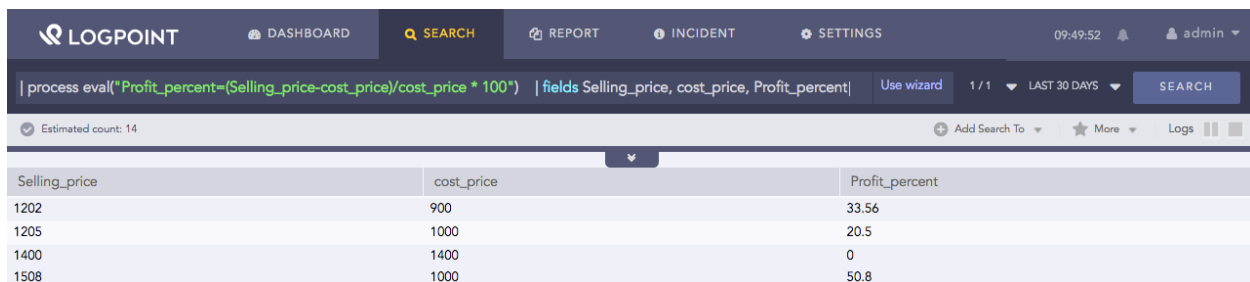
Parentheses (curved marks) can be used to group arguments in an eval expression. Parentheses clarify the order of expressions, i.e., the eval command evaluates the parts of expression within the parentheses at first.

Example:

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

The above example 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.



The screenshot shows the Logpoint dashboard interface. At the top, there is a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The user is logged in as 'admin'. Below the navigation bar, a search bar contains the query: `| process eval("Profit_percent=(Selling_price-cost_price)/cost_price * 100") | fields Selling_price, cost_price, Profit_percent`. To the right of the search bar are buttons for 'Use wizard', '1 / 1', 'LAST 30 DAYS', and a 'SEARCH' button. Below the search bar, a status bar indicates 'Estimated count: 14'. The main content area displays a table with three columns: 'Selling_price', 'cost_price', and 'Profit_percent'. The table contains four rows of data.

Selling_price	cost_price	Profit_percent
1202	900	33.56
1205	1000	20.5
1400	1400	0
1508	1000	50.8

Fig. 1: Expression with parentheses

STRING FUNCTIONS

The **String functions** evaluate string values and fields.

10.1 len

This function accepts a string value *X* as input. It evaluates the character length of the string and returns the count of the number of characters in the string.

Syntax:

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

Example:

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

The above example counts the length of the character in the *message* field 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.



The screenshot shows the Logpoint search interface. The top navigation bar includes 'LOGPOINT', 'DASHBOARD', 'SEARCH', 'REPORT', 'INCIDENT', and 'SETTINGS'. The search bar contains the query: `| process eval("message_length=len(message)") | fields message, message_length`. Below the search bar, there is a table of results. The table has two columns: the first column contains log messages, and the second column contains the corresponding `message_length` values. The results are: 'An account was successfully logged on' with length 37, 'The Password Policy Checking API was called' with length 43, and 'A user account was unlocked' with length 27. The interface also shows an estimated count of 200, a 'Display maximum: 25' dropdown, and a 'logs' button.

Message	message_length
An account was successfully logged on	37
The Password Policy Checking API was called	43
A user account was unlocked	27

Fig. 1: Len function

10.2 substr

This function accepts up to three arguments, a string value *X*, a start index and an end index. It evaluates the substring of the string *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)")
```

The above example returns the substring of the value of the *col_type* event, starting at index 0 and ending at index 4, in the *substring* identifier.

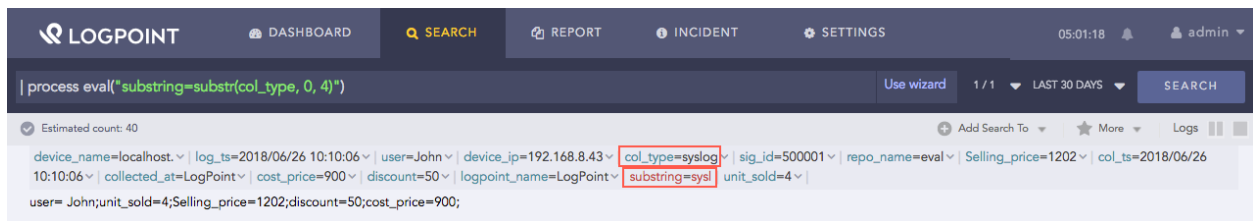


Fig. 2: Substr function

10.3 lower

This function 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
```

The above example converts the value of the *user* field to lowercase and returns it in the *username* identifier.

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

user	username
Col	col
Rachel	rachel
John	john
John	john
Jolly	jolly
Ray	rav

Fig. 3: Lower function

10.4 upper

This function 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
```

The above example converts the value of the *user* field to uppercase and returns it in the *username* identifier.

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

user	username
Col	COL
Rachel	RACHEL
John	JOHN
John	JOHN
Jolly	JOLLY
Ray	RAY
Kate	KATE

Fig. 4: Upper function

10.5 trim

This function 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)")
```

The above example removes the spaces to the left and right from the *Bob* and returns the trimmed value in the *username* identifier.

Example:

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



Fig. 5: Trim function

10.6 ltrim

This function accepts up to two string arguments *X* and *Y* as inputs. 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')")
```

The above example 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.

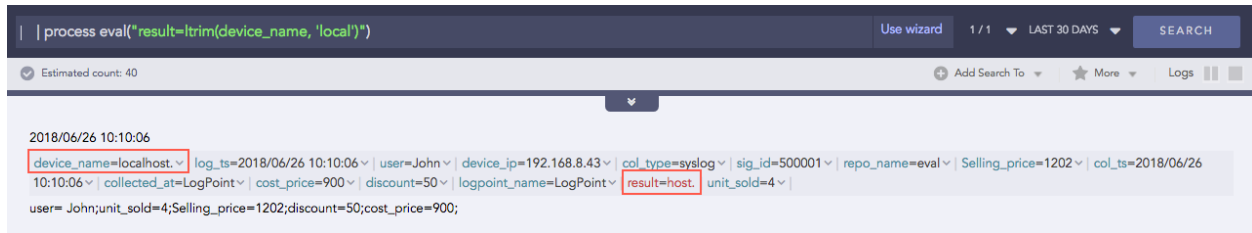


Fig. 6: Ltrim function

10.7 rtrim

This function takes up to two string arguments *X* and *Y* as inputs. It trims the strings *Y* from the right side of the field *X* 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')")
```

The above example removes the string *host* from the right side in the value of the *device_name* field and returns the trimmed value in the *result* identifier.

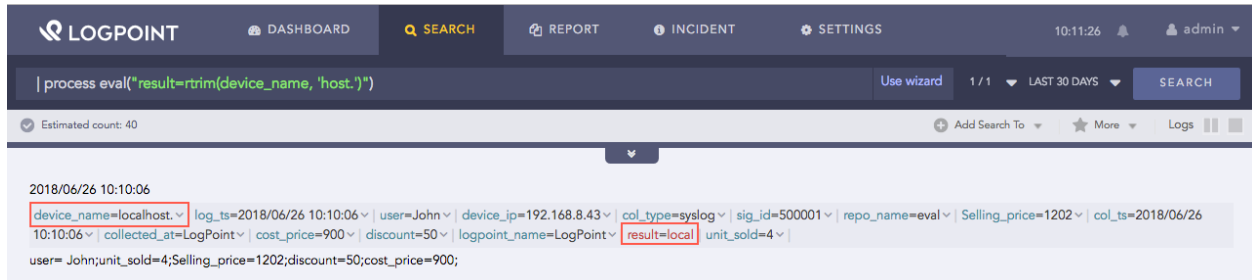


Fig. 7: Rtrim function

10.8 replace

This function accepts three arguments as inputs: 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')")
```

The above example substitutes 'X' in the string 123 for the every occurrence of the regex string [0-9] and returns the replaced value in the *result* identifier.

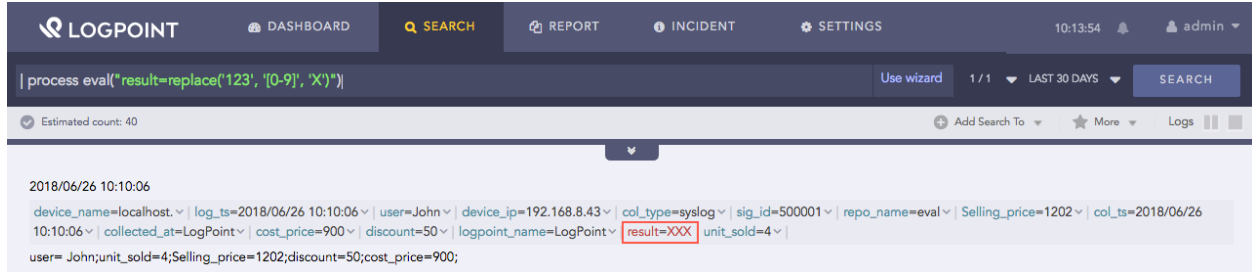


Fig. 8: Replace function

10.9 spath

This function accepts two arguments, *X* and *Y*, where *X* is the structured data type in XML or JSON format, and *Y* is the XML or JSON formatted location path. 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)")
```

Example 1:

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

The above example extracts the value from the location *name* and returns it in the *usern* identifier.

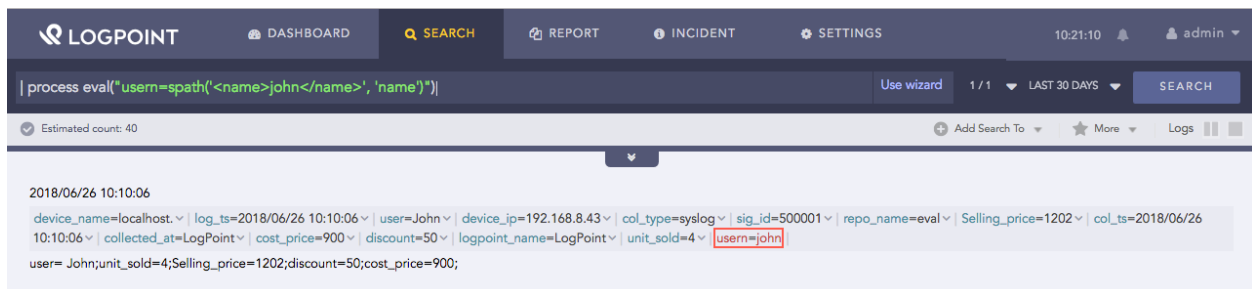


Fig. 9: Spath function

Example 2:

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

The above example extracts the value from the location `name:` and returns it in the `usern` identifier. The result is `usern=john`.

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\}`.

10.10 urldecode

This function accepts an escaped URL character `X` 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')")
```

The above example decodes an escaped url and returns the decoded url, i.e., `http://www.logpoint.com/download?r=header` in the `decoded_url` identifier.

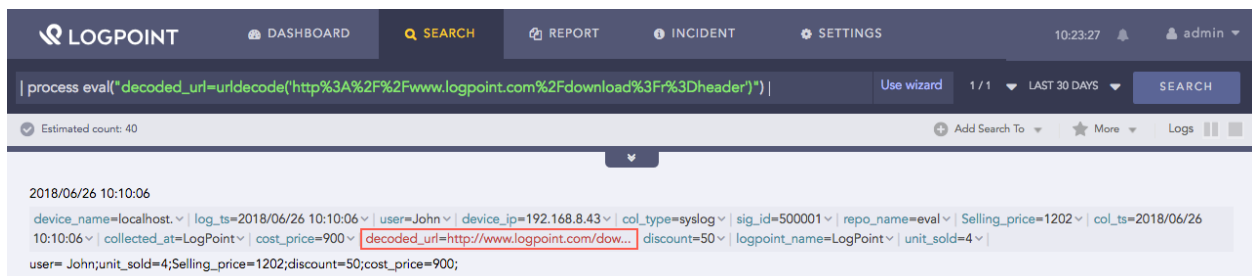


Fig. 10: Urldecode function

CRYPTOGRAPHIC FUNCTIONS

The **Cryptographic functions** evaluate hash functions and return a fixed-size alphanumeric string.

11.1 md5

This function 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)")
```

The above example converts the value of the *device_name* field into its corresponding *md5* hash value and returns the value *421AA90E079FA326B6494F812AD13E79* in the *hash* identifier.



Fig. 1: Md5 function

11.2 sha1

This function 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)")
```

The above example returns the corresponding *sha1* hash value of *device_name* in the *sha1_value* identifier.

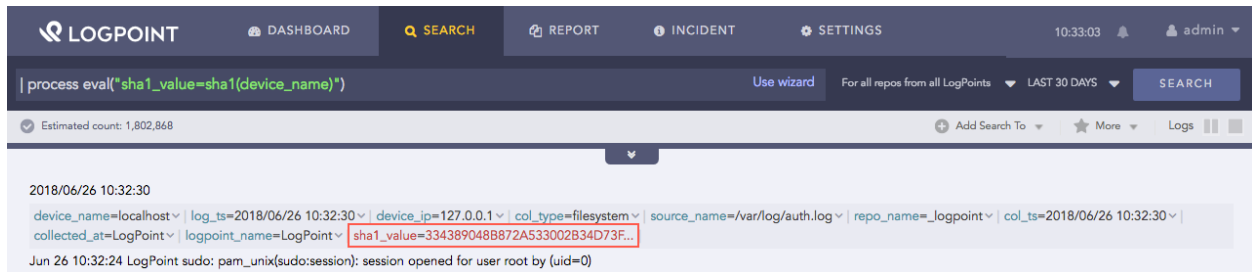


Fig. 2: Sha1 function

11.3 sha256

This function 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)")
```

The above example returns the corresponding *sha256* hash value of the *device_name* in the *sha256_value* identifier.

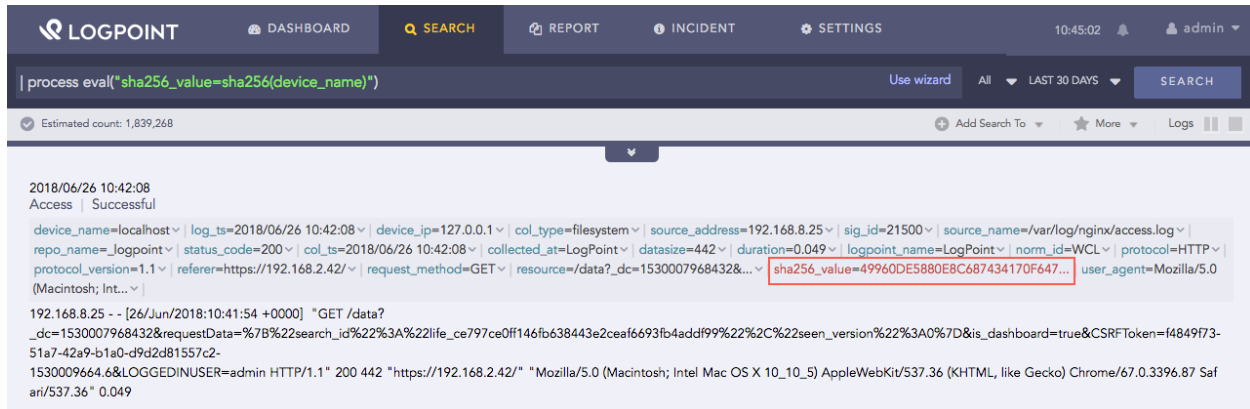


Fig. 3: Sha256 function

11.4 sha512

This function 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)")
```

The above example returns the corresponding *sha512* hash value of the *device_name* in the *sha512_value* identifier.

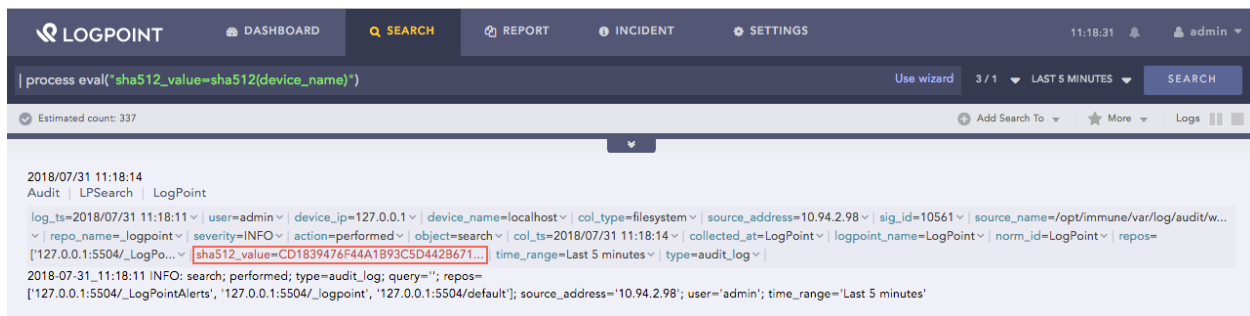


Fig. 4: Sha512 function

MATHEMATICAL FUNCTIONS

The **Mathematical functions** evaluate mathematical data.

12.1 abs

This function accepts a numerical value *X* as input and returns the absolute value of the number as the output.

Syntax:

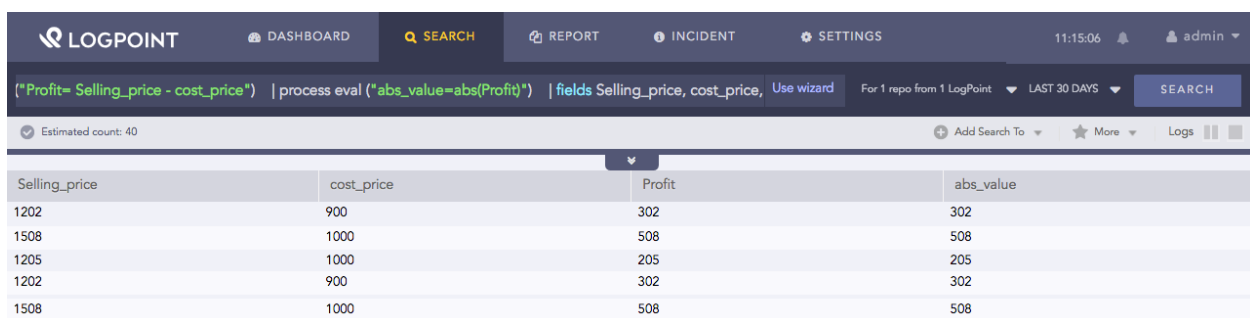
```
| process eval("identifier=abs(X)")
```

Example:

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

The above example first calculates value for a *Profit* field. It then computes the absolute value of the *Profit* field 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.



The screenshot shows the LogPoint interface with a search bar containing the query: `("Profit= Selling_price - cost_price") | process eval ("abs_value=abs(Profit)") | fields Selling_price, cost_price, Profit, abs_value`. Below the search bar, a table displays the results of the query. The table has four columns: *Selling_price*, *cost_price*, *Profit*, and *abs_value*. The data rows are as follows:

Selling_price	cost_price	Profit	abs_value
1202	900	302	302
1508	1000	508	508
1205	1000	205	205
1202	900	302	302
1508	1000	508	508

Fig. 1: Abs function

12.2 floor

This function accepts a numerical value X as input 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
```

The above example first calculates value for a *price_per_unit* field. It then computes the greatest integer less than or equal to the value of *Profit* field 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.



The screenshot shows the LogPoint dashboard with a search bar containing the query: `| process eval("final_price = floor(price_per_unit)") | fields Selling_price, unit_sold, price_per_unit, final_price`. Below the search bar, a table displays the results of the query. The table has four columns: *Selling_price*, *unit_sold*, *price_per_unit*, and *final_price*. The data rows are as follows:

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

12.3 ceiling

This function accepts a numerical value X as input and rounds the number up to the smallest following integer value.

Syntax:

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

Example:

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

The above example accepts the value of the *duration* field and returns the smallest following integer value in the *ceiling_duration* identifier.

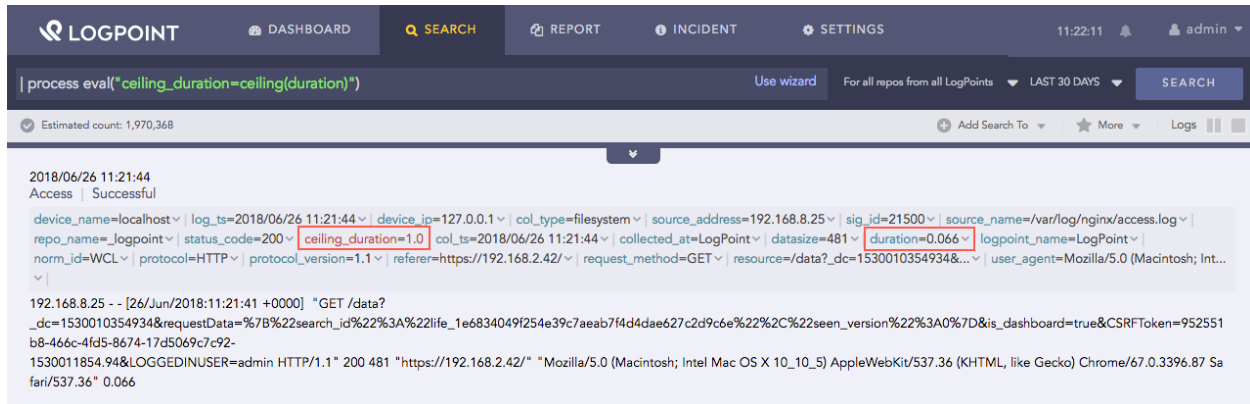


Fig. 3: Ceiling function

12.4 exp

This function accepts a numerical value X as input and evaluates the exponentiation with base e and X as the exponent, i.e., e^X . You can also use the function **expe** instead of the function **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
```

The above example evaluates the exponentiation to the base e 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.

discount	result
99	9.889030319346946e+42
11	59874.14
10	22026.47

Fig. 4: Exp function

discount	result
99	9.889030319346946e+42
11	59874.14
10	22026.47

Fig. 5: Expe function

12.5 exp2

This function accepts a numerical value X as input and evaluates the exponentiation with base 2 and X as the exponent, i.e., 2^X .

Syntax:

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

Example:

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

The above example evaluates the exponentiation to the base 2 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.

unit_sold	result
4	16
14	16384
2	4
20	1048576

Fig. 6: Exp2 function

12.6 exp10

This function accepts a numerical value X as input and evaluates the exponentiation with base 10 and X as the exponent, i.e., 10^X .

Syntax:

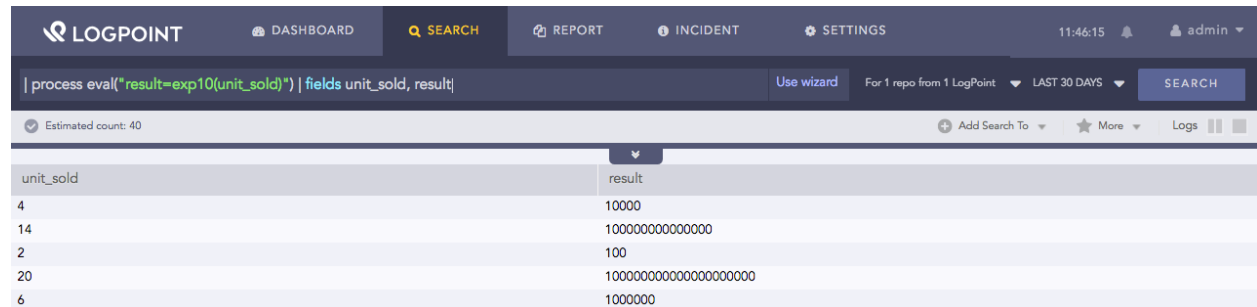
```
| process eval("identifier=exp10(X)")
```

Example:

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

The above example evaluates the exponentiation to the base 10 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.



unit_sold	result
4	10000
14	1000000000000000
2	100
20	100000000000000000000
6	1000000

Fig. 7: Exp10 function

12.7 log

This function accepts a numerical value X as input and evaluates the logarithm of X with base e, i.e., $\log_e(X)$. You can also use the function **loge** instead of the function **log**.

Syntax:

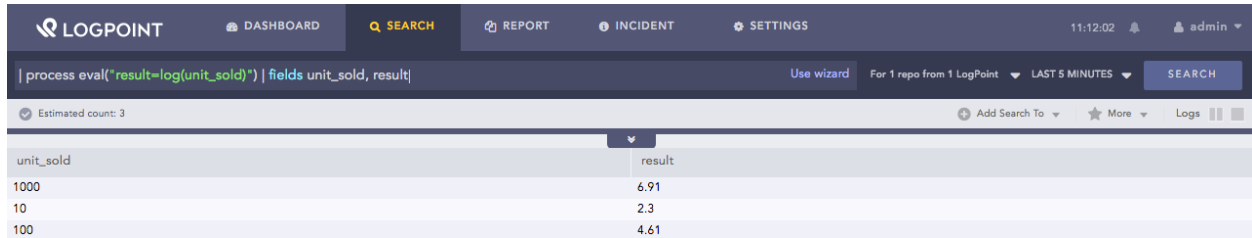
```
| process eval("identifier=log(X)")
or
| process eval("identifier=loge(X)")
```

Example:

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

The above example evaluates the logarithm to the base e of the `unit_sold` field and returns it in the `result` identifier.

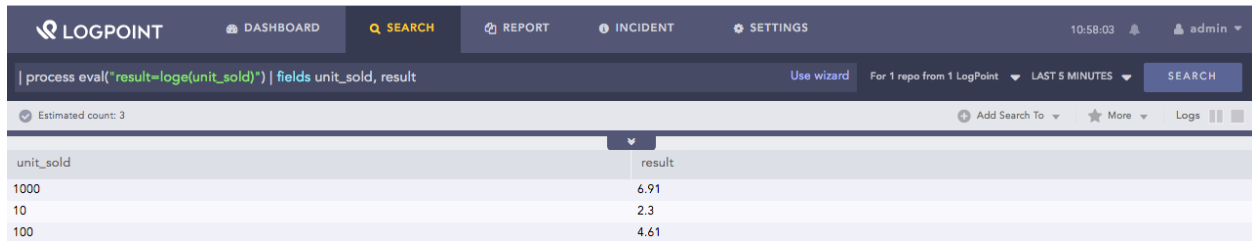
The `fields` command displays the value of `unit_sold` and `result` in a tabular form.



The screenshot shows the LogPoint search interface. The search bar contains the query: `| process eval("result=log(unit_sold)") | fields unit_sold, result`. The results table has two columns: `unit_sold` and `result`. The estimated count is 3.

unit_sold	result
1000	6.91
10	2.3
100	4.61

Fig. 8: Log function



The screenshot shows the LogPoint search interface. The search bar contains the query: `| process eval("result=loge(unit_sold)") | fields unit_sold, result`. The results table has two columns: `unit_sold` and `result`. The estimated count is 3.

unit_sold	result
1000	6.91
10	2.3
100	4.61

Fig. 9: Loge function

12.8 log2

This function accepts a numerical value X as input and evaluates the logarithm of X with base 2, i.e., $\log_2(X)$.

Syntax:

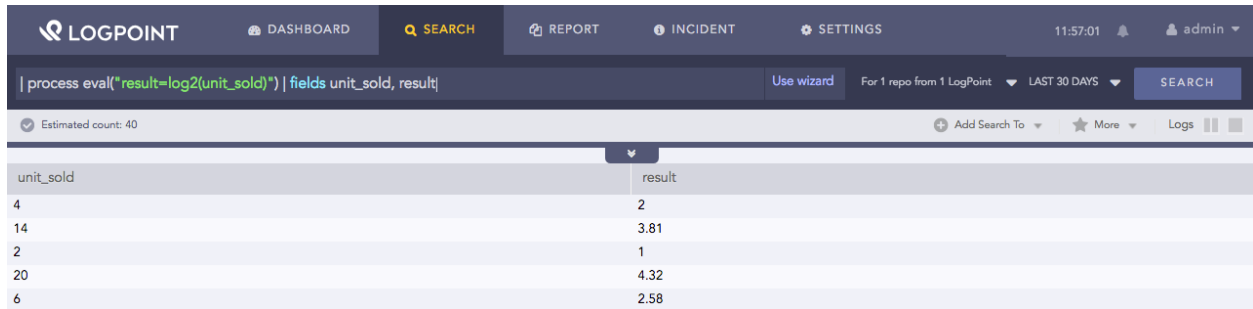
```
| process eval("identifier=log2(X)")
```

Example:

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

The above example evaluates the logarithm to the base 2 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.



unit_sold	result
4	2
14	3.81
2	1
20	4.32
6	2.58

Fig. 10: Log2 function

12.9 log10

This function accepts a numerical value X as input and evaluates the logarithm of X with base 10, i.e., $\log_{10}(X)$.

Syntax:

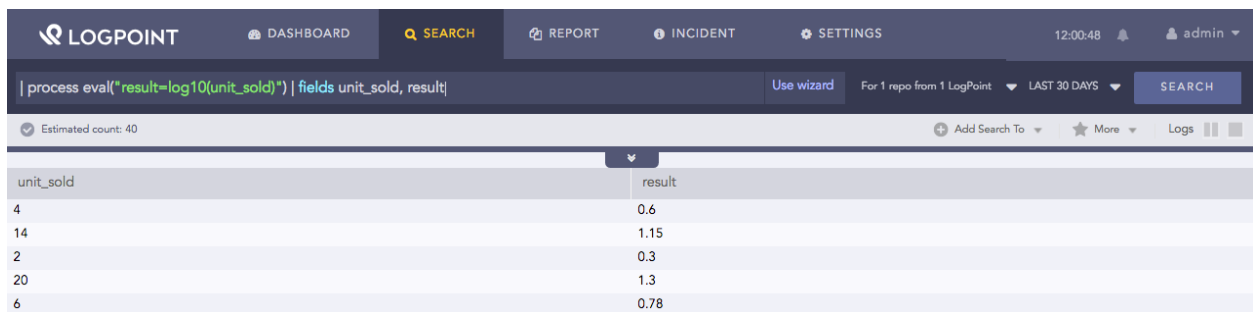
```
| process eval("identifier=log10(X)")
```

Example:

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

The above example evaluates the logarithm to the base 10 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.



unit_sold	result
4	0.6
14	1.15
2	0.3
20	1.3
6	0.78

Fig. 11: Log10 function

12.10 pi

This function 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
```

The above example calculates the area of the circle where *pi()* gives the value of the mathematical constant (π). The function 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.

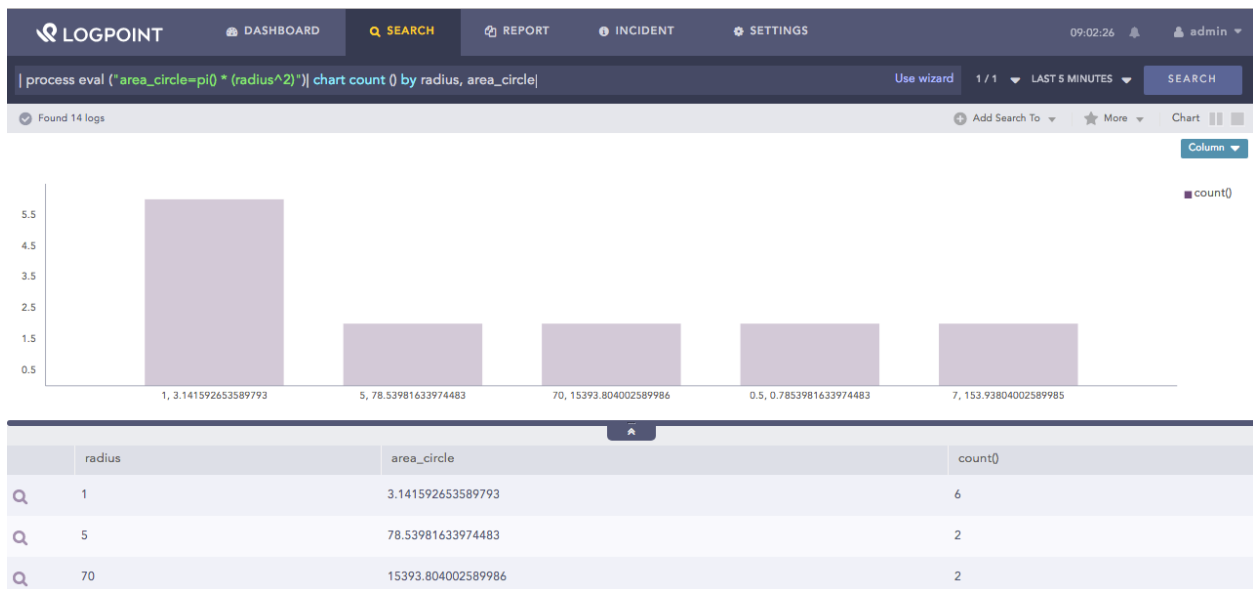


Fig. 12: pi function

12.11 sqrt

This function accepts a numeric value *X* as input and returns the square root of the numeric value.

Syntax:

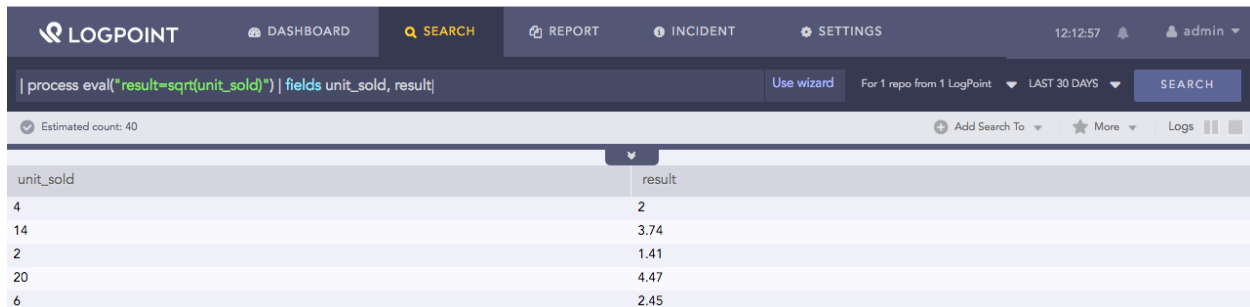
```
| process eval("identifier=sqrt(X)")
```

Example:

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

The above example returns the square root of the value of the `unit_sold` field in the `unit_sold` identifier.

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



The screenshot shows the LogPoint dashboard with a search bar containing the query `| process eval("result=sqrt(unit_sold)") | fields unit_sold, result|`. Below the search bar, a table displays the results of the query. The table has two columns: `unit_sold` and `result`. The results are as follows:

unit_sold	result
4	2
14	3.74
2	1.41
20	4.47
6	2.45

Fig. 13: Sqrt function

12.12 random

This function 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.

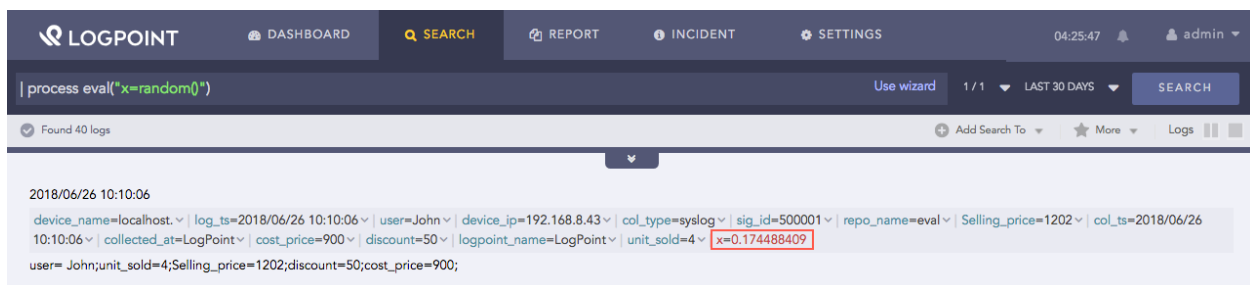
Syntax:

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

Example:

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

The above example returns a random number between 0 and 1 in the `x` identifier.



The screenshot shows the LogPoint dashboard with a search bar containing the query `| process eval("x=random()")`. Below the search bar, a table displays the results of the query. The table has two columns: `unit_sold` and `result`. The results are as follows:

unit_sold	result
4	0.174488409

Fig. 14: Random function

12.13 exact

This function accepts a numeric calculation X as input and returns a result with a significant amount of precision.

Syntax:

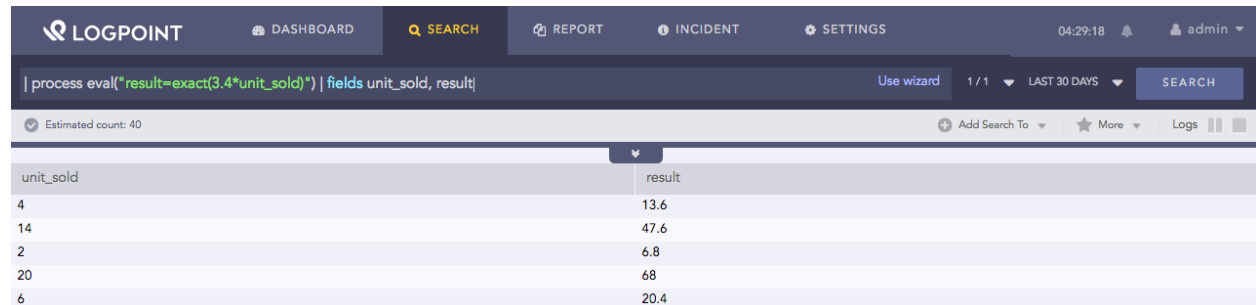
```
| process eval("identifier=exact(X)")
```

Example:

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

The above example 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.



The screenshot shows the Logpoint dashboard interface. At the top, there's a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The current view is the SEARCH tab. Below the navigation bar, a search query is entered: `| process eval("result=exact(3.4*unit_sold)") | fields unit_sold, result`. The results are displayed in a table with two columns: *unit_sold* and *result*. The table contains five rows of data.

unit_sold	result
4	13.6
14	47.6
2	6.8
20	68
6	20.4

Fig. 15: Exact function

12.14 round

This function accepts up to two numeric arguments, X and Y , as inputs. It then 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)")
```

The above example rounds up 12.234 to the hundredths (second decimal place) and returns its value in the *result* identifier.

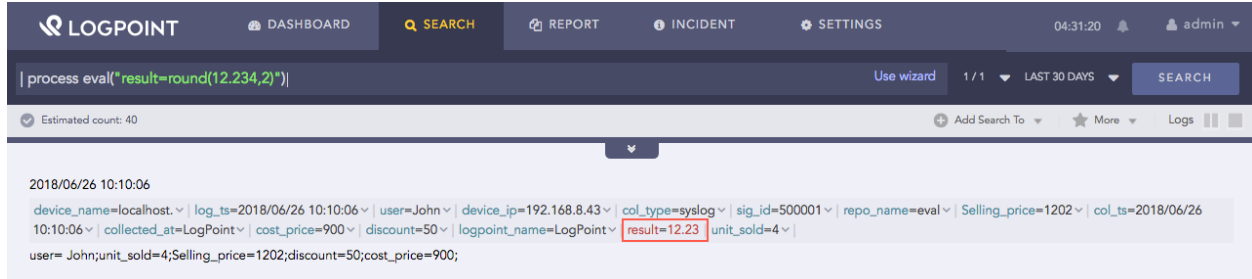


Fig. 16: Round function

12.15 sigfig

This function accepts one numeric field *X* as input and rounds that number to the appropriate number of significant figures.

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)")
```

The above example rounds up 1111 to the nearest 10 and returns its value, i.e., 1110 in the *result* identifier.



Fig. 17: Sigfig function

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

Example 2:

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

The above example rounds up 11111 to the nearest 100 and returns its value, i.e., 11100 in the *result* identifier.

Note: This function accepts only integer value as input. 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.

TRIGONOMETRIC FUNCTIONS

The **Trigonometric functions** evaluate trigonometric and hyperbolic values.

13.1 sin

This function accepts one argument *X* as input and returns the sine of *X*.

Syntax:

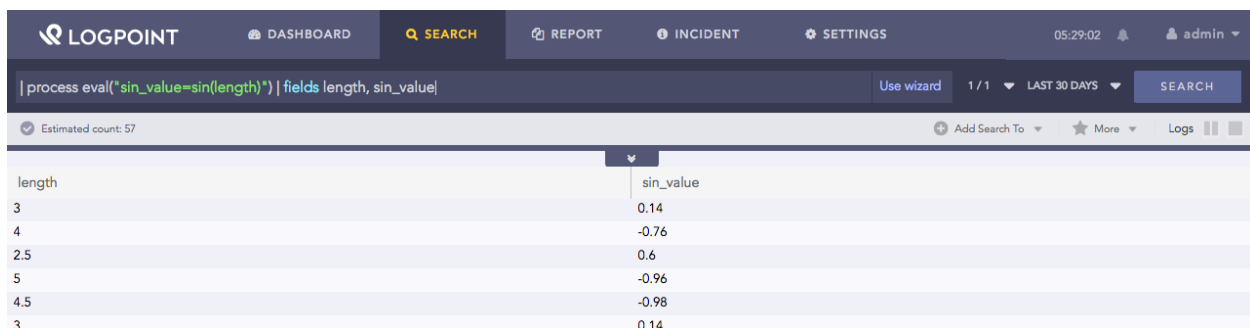
```
| process eval("identifier=sin(X)")
```

Example:

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

The above example returns the sine of the *length* field in the *sin_value* identifier.

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



The screenshot shows the Logpoint search interface. The search bar contains the query: `| process eval("sin_value=sin(length)") | fields length, sin_value`. Below the search bar, the results are displayed in a table with two columns: *length* and *sin_value*. The table contains six rows of data. The estimated count is 57.

length	sin_value
3	0.14
4	-0.76
2.5	0.6
5	-0.96
4.5	-0.98
3	0.14

Fig. 1: Sin function

13.2 sinh

This function accepts one argument *X* as input and returns the hyperbolic sine of *X*.

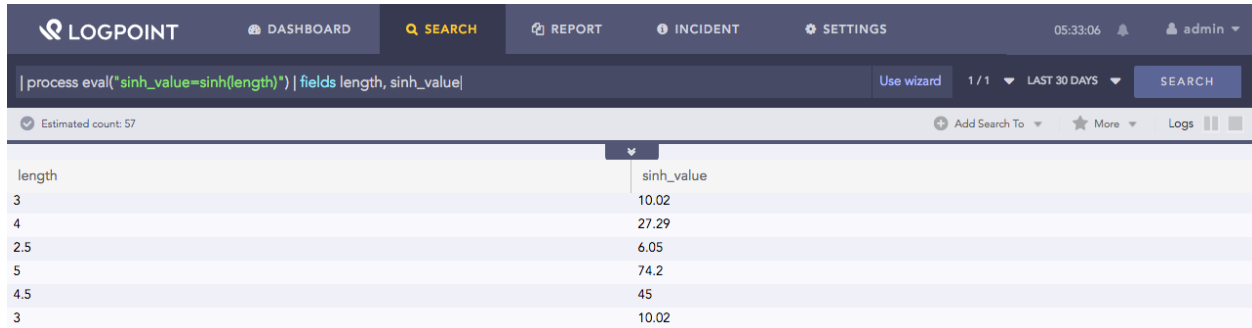
Syntax:

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

Example:

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

The above example returns the hyperbolic sine of the *length* field in the *sinh_value* identifier.



The screenshot shows the Logpoint dashboard with a search query: `| process eval("sinh_value=sinh(length)") | fields length, sinh_value`. The results table has two columns: *length* and *sinh_value*. The estimated count is 57.

length	sinh_value
3	10.02
4	27.29
2.5	6.05
5	74.2
4.5	45
3	10.02

Fig. 2: Sinh function

13.3 asin

This function accepts one argument *X* as input and returns the inverse sine of *X*. *X* must be in the range from -1 to 1 inclusive.

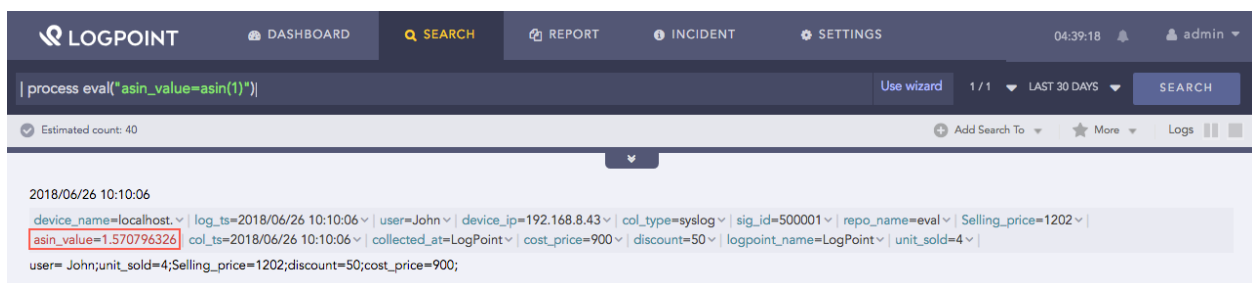
Syntax:

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

Example:

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

The above example returns the inverse sine of 1 in the *asin_value* identifier.



The screenshot shows the Logpoint dashboard with a search query: `| process eval("asin_value=asin(1)")`. The results table shows a single row with the *asin_value* field highlighted in red. The estimated count is 40.

2018/06/26 10:10:06
<code>device_name=localhost log_ts=2018/06/26 10:10:06 user=John device_ip=192.168.8.43 col_type=syslog sig_id=500001 repo_name=eval Selling_price=1202 asin_value=1.570796326 col_ts=2018/06/26 10:10:06 collected_at=LogPoint cost_price=900 discount=50 logpoint_name=LogPoint unit_sold=4 user= John;unit_sold=4;Selling_price=1202;discount=50;cost_price=900;</code>

Fig. 3: Asin function

13.4 asinh

This function 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)")
```

The above example returns the inverse hyperbolic sine of 1 in the *asinh_value* identifier.

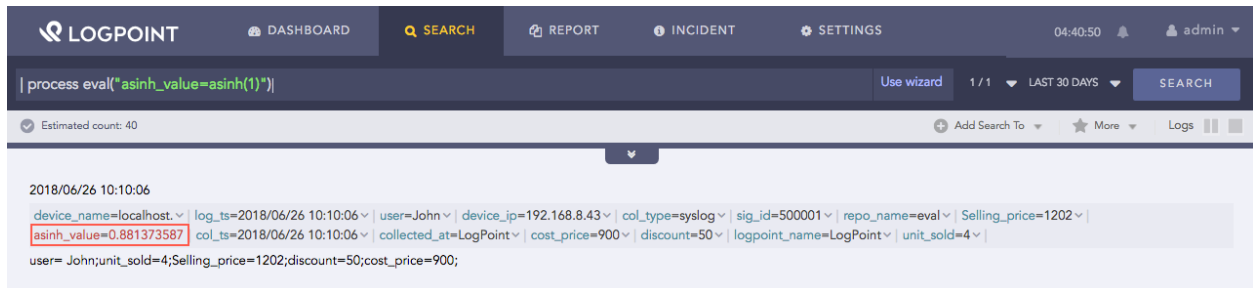


Fig. 4: Asinh function

13.5 cos

This function accepts one argument X as input and returns the cosine of X .

Syntax:

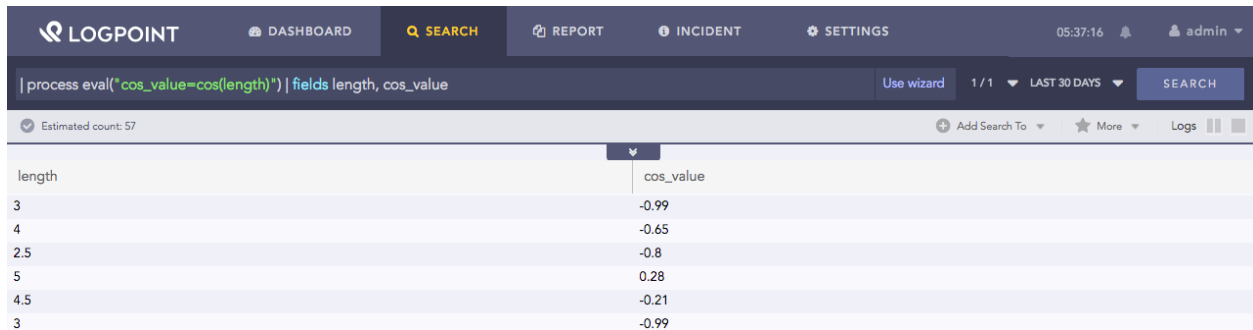
```
| process eval("identifier=cos(X)")
```

Example:

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

The above example returns the cosine of the *length* field in the *cos_value* identifier.

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



The screenshot shows the Logpoint search interface with a search query: `| process eval("cos_value=cos(length)") | fields length, cos_value`. The results are displayed in a table with two columns: `length` and `cos_value`. The estimated count is 57.

length	cos_value
3	-0.99
4	-0.65
2.5	-0.8
5	0.28
4.5	-0.21
3	-0.99

Fig. 5: Cos function

13.6 cosh

This function accepts one argument X as input and returns the hyperbolic cosine of X .

Syntax:

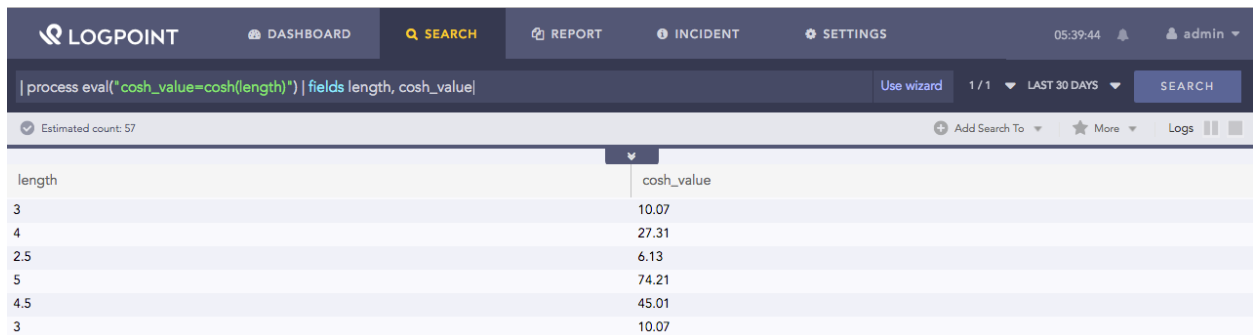
```
| process eval("identifier=cosh(X)")
```

Example:

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

The above example returns the hyperbolic cosine of the `length` field in the `cosh_value` identifier.

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



The screenshot shows the Logpoint search interface with a search query: `| process eval("cosh_value=cosh(length)") | fields length, cosh_value`. The results are displayed in a table with two columns: `length` and `cosh_value`. The estimated count is 57.

length	cosh_value
3	10.07
4	27.31
2.5	6.13
5	74.21
4.5	45.01
3	10.07

Fig. 6: Cosh function

13.7 acos

This function 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)")
```

The above example returns the inverse cosine of 0 in the `acos_value` identifier.

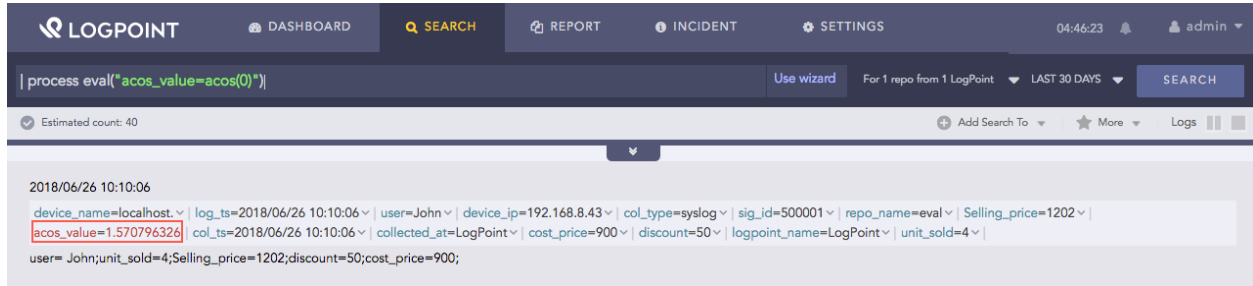


Fig. 7: Acos function

13.8 acosh

This function 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)")
```

The above example returns the inverse hyperbolic cosine of 1 in the `acosh_value` identifier.

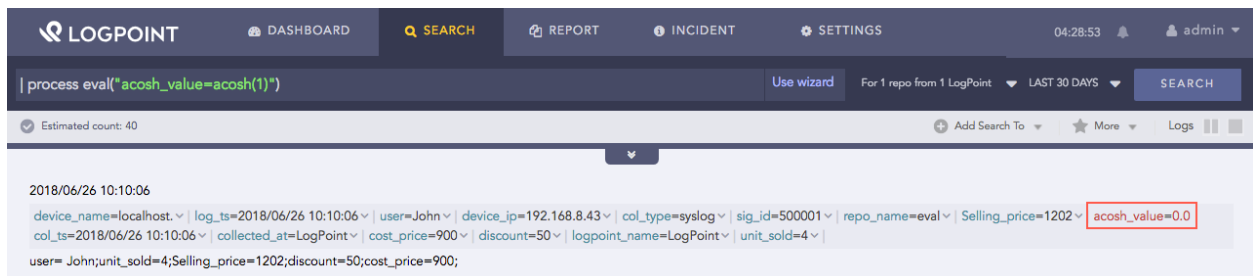


Fig. 8: Acosh function

13.9 tan

This function accepts one argument X as input and returns the tangent of X .

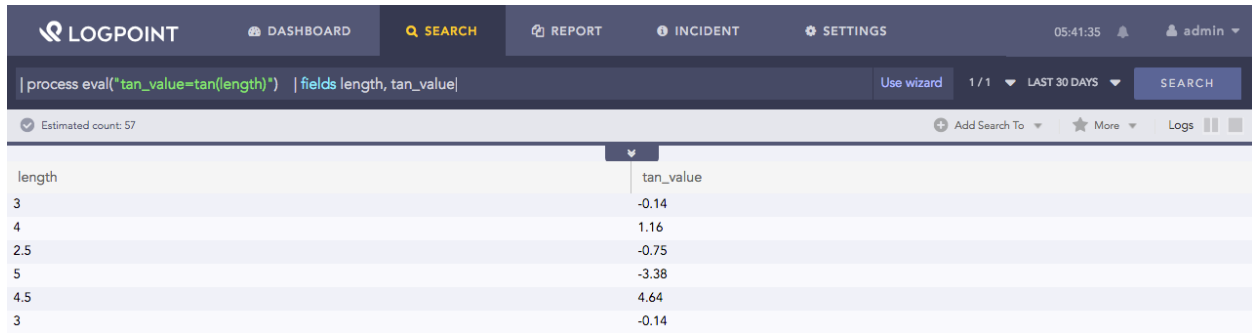
Syntax:

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

Example:

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

The above example returns the tangent of the *length* field in the *tan_value* identifier. The *fields* command displays the value of *length* and *tan_value* in a tabular form.



The screenshot shows the Logpoint dashboard interface. At the top, there's a navigation bar with tabs for DASHBOARD, SEARCH, REPORT, INCIDENT, and SETTINGS. The current view is the SEARCH tab. Below the navigation bar, there's a search bar containing the query: `| process eval("tan_value=tan(length)") | fields length, tan_value`. To the right of the search bar are buttons for "Use wizard", "1 / 1", "LAST 30 DAYS", and a "SEARCH" button. Below the search bar, there's a status bar showing "Estimated count: 57" and options to "Add Search To", "More", and "Logs". The main area displays a table with two columns: "length" and "tan_value". The table contains six rows of data.

length	tan_value
3	-0.14
4	1.16
2.5	-0.75
5	-3.38
4.5	4.64
3	-0.14

Fig. 9: Tan function

13.10 tanh

This function 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
```

The above example returns the hyperbolic tangent of the *length* field in the *tanh_value* identifier.

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



The screenshot shows the LogPoint search interface. The search bar contains the query `| process eval("tanh_value=tanh(length)") | fields length, tanh_value|`. The results table has two columns: `length` and `tanh_value`. The estimated count is 57.

length	tanh_value
3	1
4	1
2.5	0.99
5	1
4.5	1
3	1

Fig. 10: Tanh function

13.11 atan

This function accepts one argument X as input and returns the inverse tangent of X .

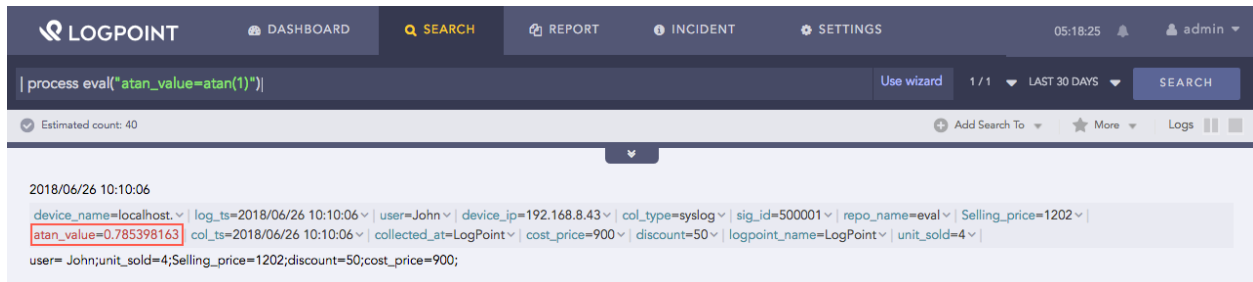
Syntax:

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

Example:

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

The above example returns the inverse tangent of 1 in the `atan_value` identifier.



The screenshot shows the LogPoint search interface. The search bar contains the query `| process eval("atan_value=atan(1)")`. The results table shows a single entry with the `atan_value` field highlighted in red. The estimated count is 40.

2018/06/26 10:10:06
<code>device_name=localhost log_ts=2018/06/26 10:10:06 user=John device_ip=192.168.8.43 col_type=syslog sig_id=500001 repo_name=eval Selling_price=1202 atan_value=0.785398163 col_ts=2018/06/26 10:10:06 collected_at=LogPoint cost_price=900 discount=50 logpoint_name=LogPoint unit_sold=4 user= John;unit_sold=4;Selling_price=1202;discount=50;cost_price=900;</code>

Fig. 11: Atan function

13.12 atanh

This function 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)")
```

The above example returns the inverse hyperbolic tangent of 1 in the *atanh_value* identifier.



Fig. 12: Atanh function

13.13 hypot

This function accepts two arguments *X* and *Y* as inputs and returns the hypotenuse of a right-angled triangle whose length and base are *X* and *Y*. It follows the equation of the Pythagorean theorem, i.e., $hypotenuse = \sqrt{length^2 + base^2}$.

Syntax:

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

Example:

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

The above example calculates the value of the hypotenuse of the triangle whose length and base are 3 and 4 respectively and returns its value in the *tan_value* identifier.

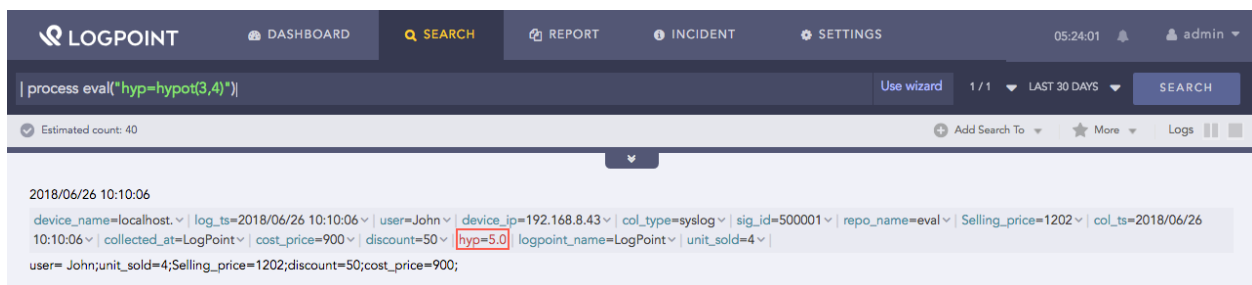


Fig. 13: Hypot function

DATE/TIME FUNCTIONS

The **Date/Time functions** evaluate the date and time values.

14.1 now

This function 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()")
```

The above example returns the UNIX time of the search process in the *search* identifier.

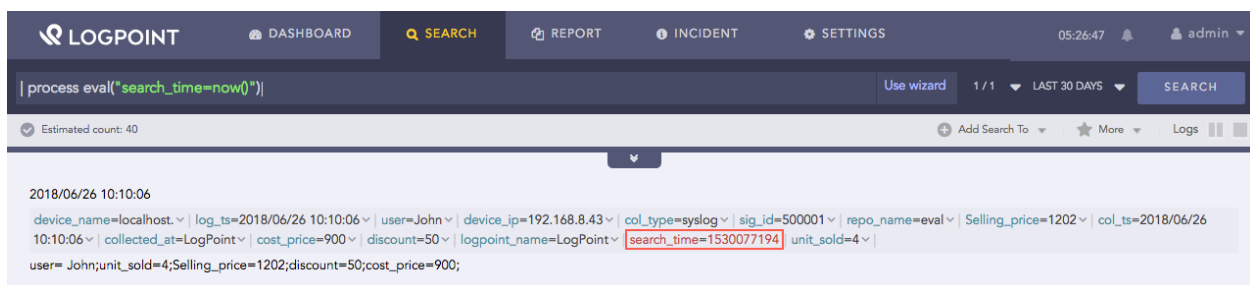


Fig. 1: Now function

14.2 relative_time

This function accepts two arguments, a UNIX time X, and a relative time specifier Y as inputs, 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')")
```

The above example adds time equivalent of 1 day to the current UNIX time and returns it in the *result* identifier.

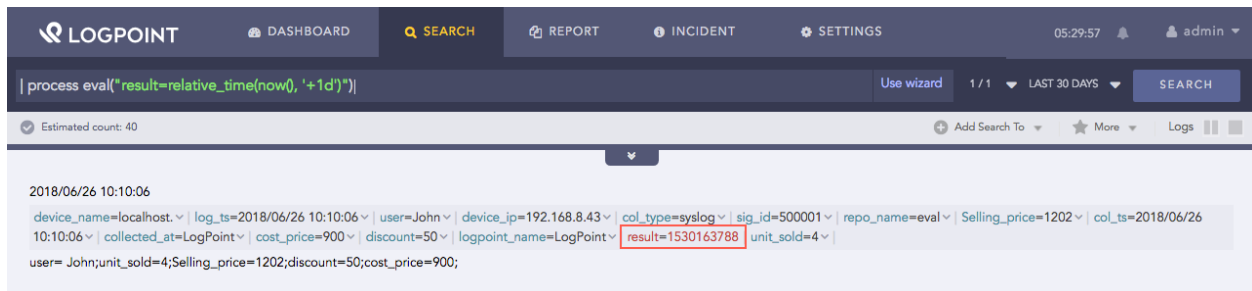


Fig. 2: Relative time function

14.3 strftime

This function 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')")
```

The above example accepts the current UNIX time and returns the time in YYYY/MM/DD format in the *search_date* identifier.

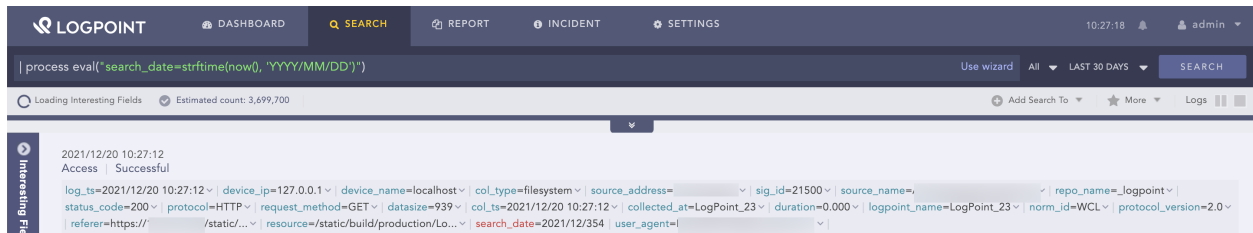


Fig. 3: Strftime function for day in year

Example:

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

The above example accepts the current UNIX time and returns the time in YYYY/MM/dd format in the `search_date` identifier.

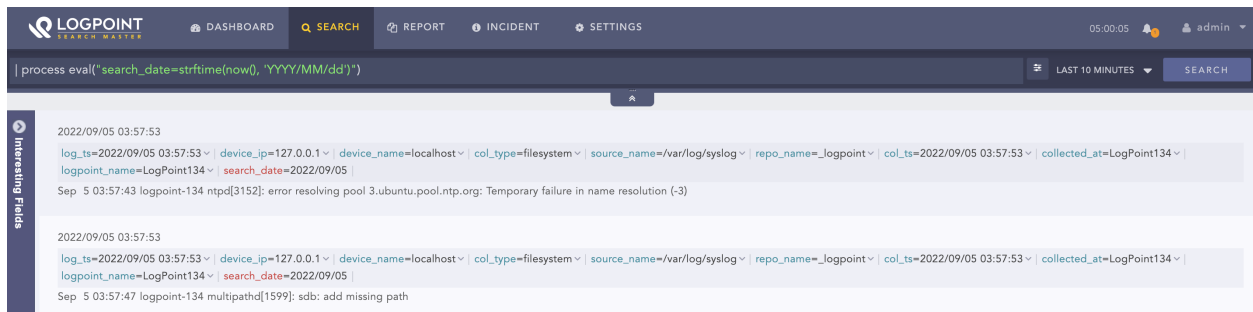


Fig. 4: Strftime function for day in month

14.4 strftime

This function 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=strftime(X, Y)")
```

Example:

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

The above example accepts the human readable time and converts it to a UNIX timestamp. It returns the converted time in the `searchtime` identifier.

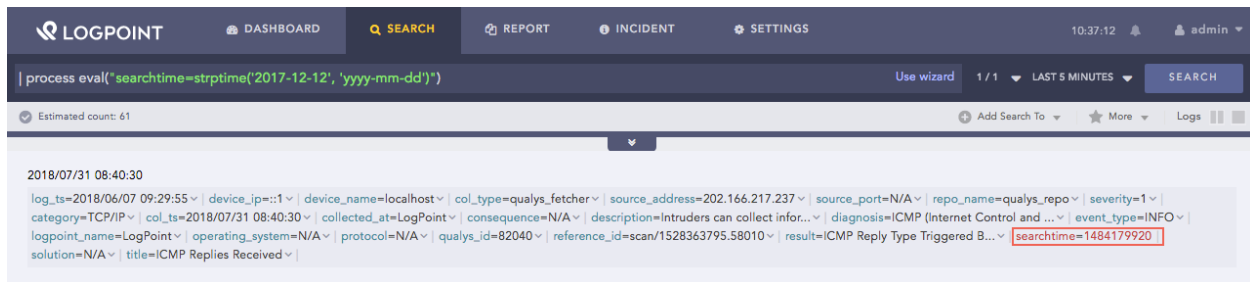


Fig. 5: Strptime function

14.5 time

This function 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()")
```

The above example returns the time of the eval command execution in the *process_time* identifier.

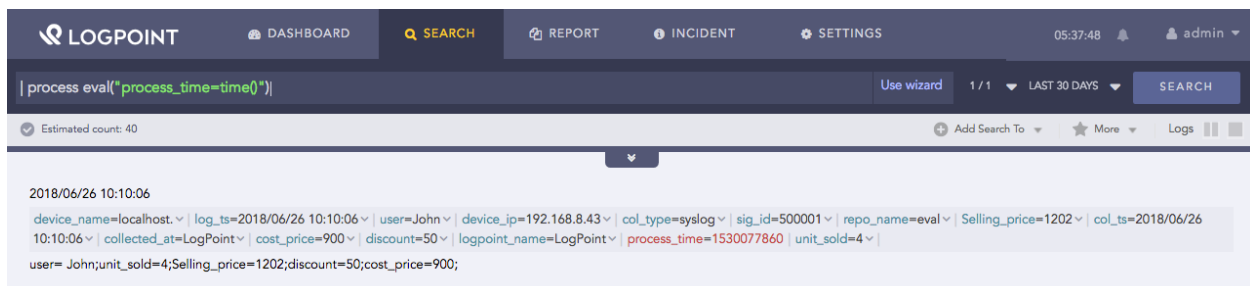


Fig. 6: Time function

14.6 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

INFORMATIONAL FUNCTIONS

The **Informational functions** evaluate the information of arguments.

15.1 isbool

This function accepts one argument *X* as input and returns *True* if the value of *X* is Boolean. If the value is not a Boolean, the function 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
```

The above example first evaluates if the *Selling_price* is less than *cost_price* and returns its value in the *is_loss* identifier. Then, the *isbol* function 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.

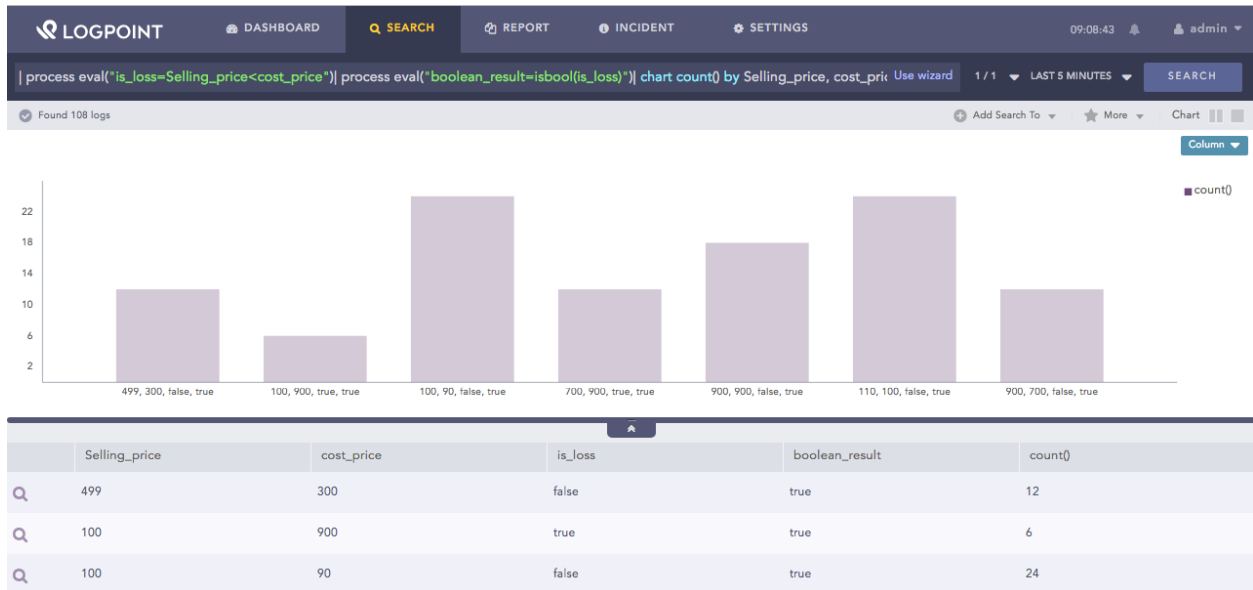


Fig. 1: Isbool function

15.2 isint

This function accepts one argument *X* as input 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
```

The above example 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.



The screenshot shows the Logpoint dashboard with a search bar containing the query: `| process eval("isscore_int=isint(score)") | fields score, isscore_int`. Below the search bar, a table displays the results of the query. The table has two columns: `score` and `isscore_int`. The results are as follows:

score	isscore_int
90	true
76	true
42	true
90	true
76	true

Fig. 2: Isint function

15.3 isnotnull

This function accepts one argument *X* as input 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
```

The above example first evaluates if the *Selling_price* is less than *cost_price* and returns its value in the *is_loss* identifier. Then, the *isnotnull* function 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.

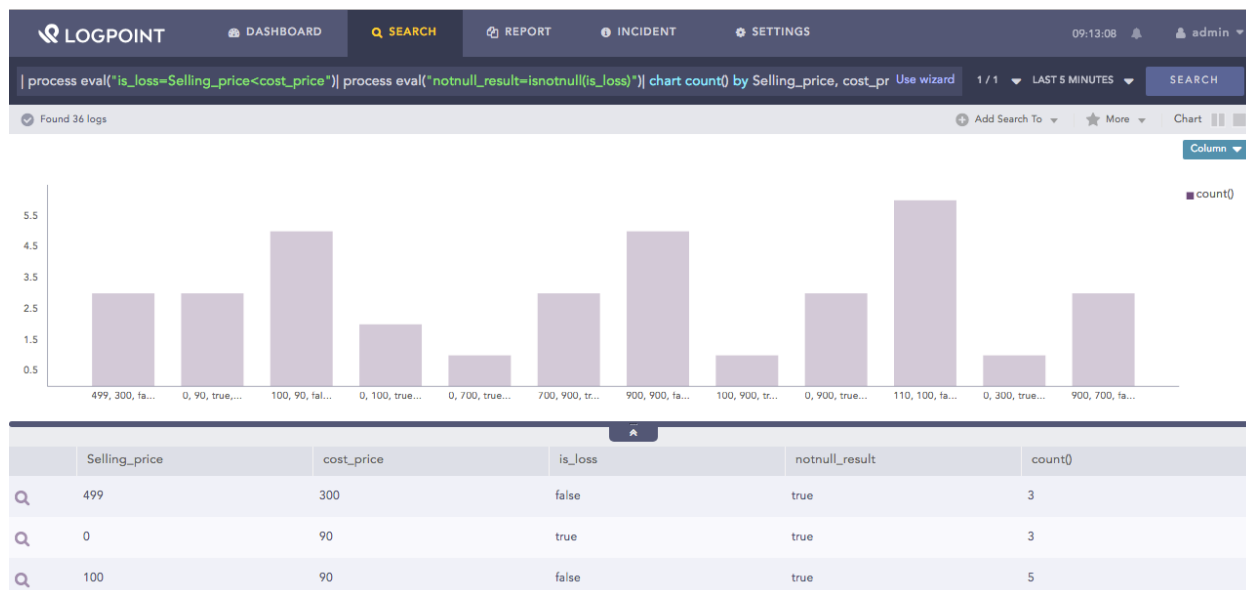


Fig. 3: Isnotnull function

15.4 isnull

This function accepts one argument *X* as input 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
```

The above example first evaluates if the *Selling_price* is less than *cost_price* and returns its value in the *is_loss* identifier. Then, the *isnull* function 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.

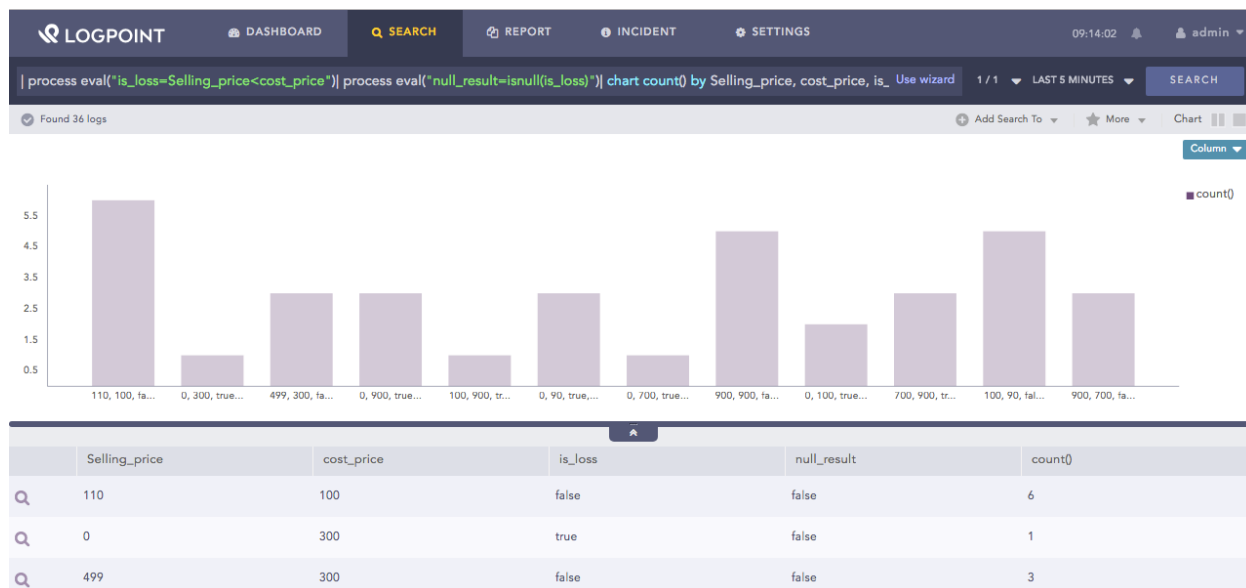


Fig. 4: Isnull function

15.5 isnum

This function accepts one argument *X* as input 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
```

The above example 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.

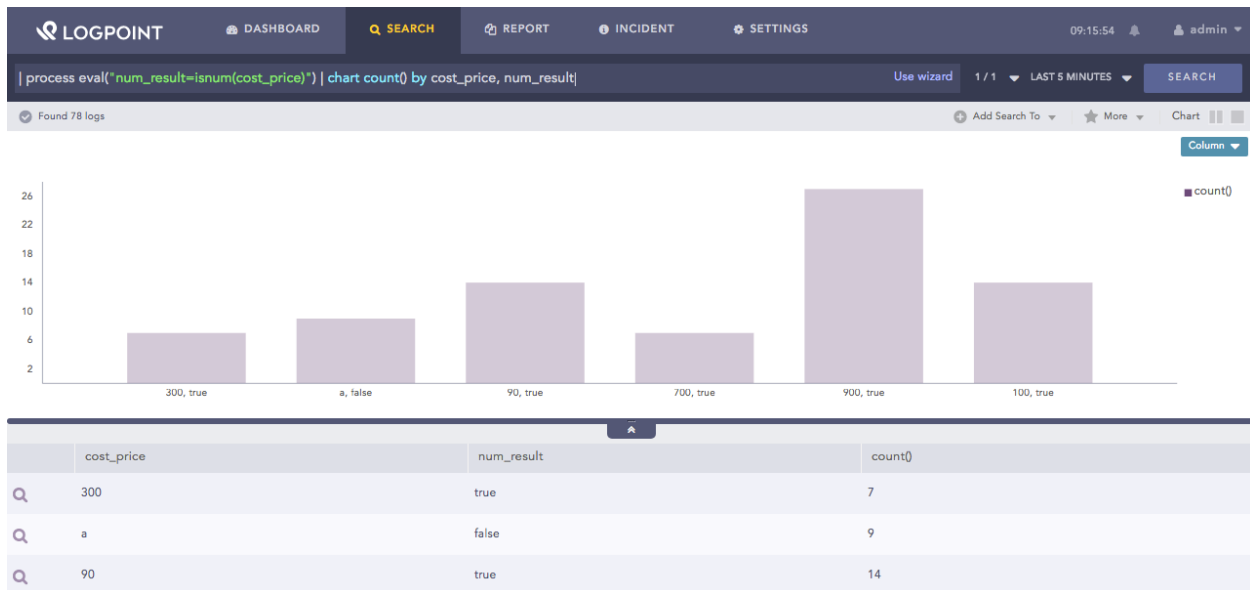


Fig. 5: Isnum function

15.6 isstr

This function accepts one argument *X* as input 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
```

The above example 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.

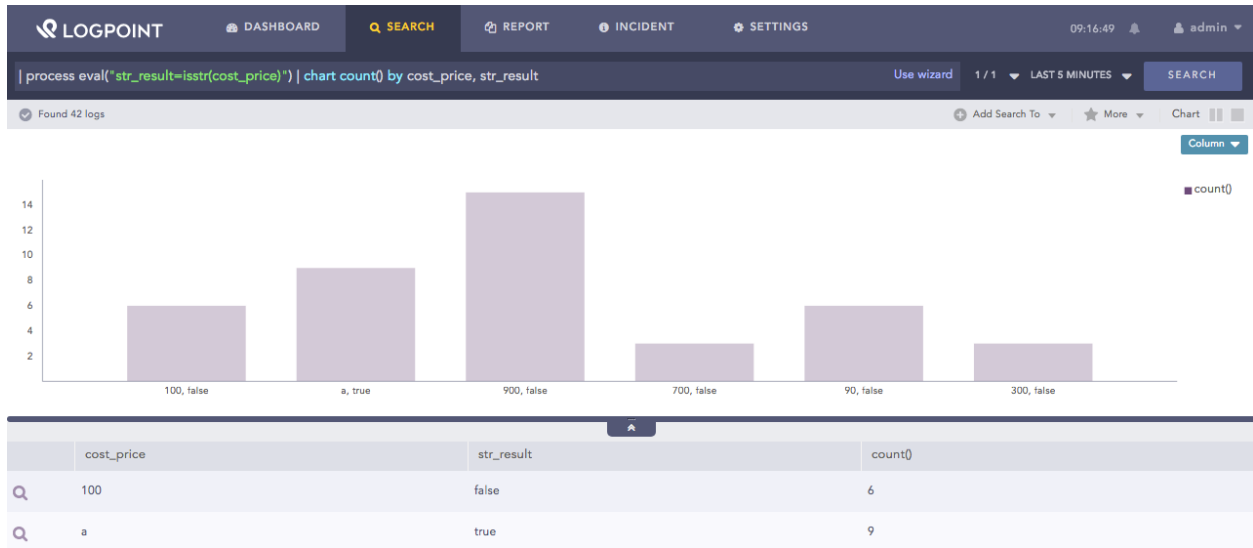


Fig. 6: Lsstr function

15.7 typeof

This function accepts one argument *X* as input 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
```

The above example 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.

The screenshot shows the Logpoint dashboard with a search query: `| process eval("event_type=typeof(event_id)") | fields event_id, event_type`. The search results show an estimated count of 14 logs. A table displays the values of `event_id` and `event_type`.

event_id	event_type
4771	Integer
4900	Integer
4768	Integer

Fig. 7: Typeof function

UNINSTALLING THE APPLICATION

16.1 Uninstalling the Evaluation Process Plugin Application in LogPoint

1. Go to *Settings >> System >> Applications*.
2. Click the **Uninstall** (⇒) icon from the **Actions** column.

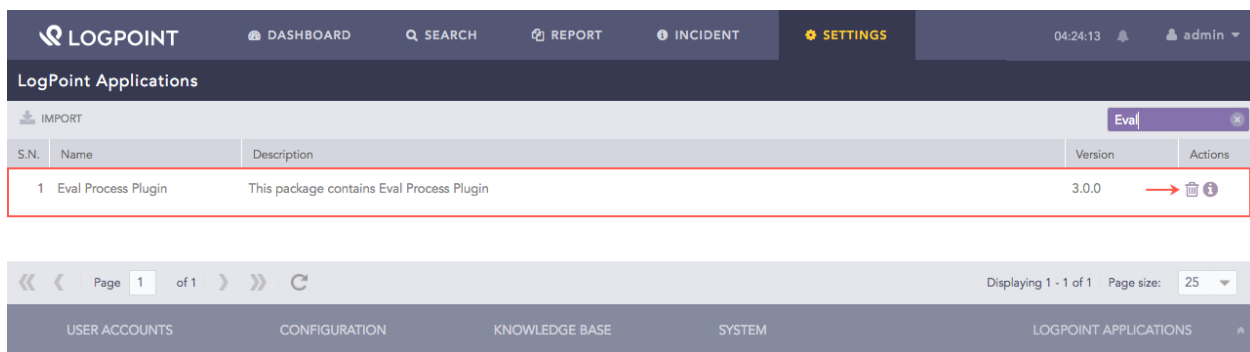


Fig. 1: Evaluation Process Plugin Uninstallation