**eval**

**Description**

The `eval` command calculates an expression and puts the resulting value into a search results field.

- If the field name that you specify does not match a field in the output, a new field is added to the search results.
- If the field name that you specify matches a field name that already exists in the search results, the results of the `eval` expression overwrite the values in that field.

The `eval` command evaluates mathematical, string, and boolean expressions.

You can chain multiple `eval` expressions in one search using a comma to separate subsequent expressions. The search processes multiple `eval` expressions left-to-right and lets you reference previously evaluated fields in subsequent expressions.

**Difference between eval and stats commands**

The `stats` command calculates statistics based on fields in your events. The `eval` command creates new fields in your events by using existing fields and an arbitrary expression.

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

... | `eval E = ...`

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Syntax**

```
eval <field>=<expression>["", <field>=<expression>]...
```

**Required arguments**

`field`

*Syntax: <string>*
Description: A destination field name for the resulting calculated value. If the field name already exists in your events, eval overwrites the value.

expression

Syntax: <string>

Description: A combination of values, variables, operators, and functions that will be executed to determine the value to place in your destination field.

The syntax of the eval expression is checked before running the search, and an exception is thrown for an invalid expression.

* The result of an eval expression cannot be a Boolean.
* If, at search time, the expression cannot be evaluated successfully for a given event, the eval command erases the resulting field.
* If the expression references a field name that contains non-alphanumeric characters, other than the underscore ( _ ) character, the field name needs to be surrounded by single quotation marks. For example, if the field name is server-1 you specify the field name like this new=count+'server-1'.
* If the expression references a literal string, that string needs to be surrounded by double quotation marks. For example, if the string you want to use is server- you specify the string like this new="server-".host.

Usage

The eval command is a distributable streaming command. See Command types.

General

You must specify a field name for the results that are returned from your eval command expression. You can specify a name for a new field or for an existing field.

If the field name that you specify matches an existing field name, the values in the existing field are replaced by the results of the eval expression.

Numbers and strings can be assigned to fields, while booleans cannot be assigned. However you can convert booleans and nulls to strings using the tostring() function, which can be assigned to fields.
During calculations, numbers are double precision floating point numbers subject to all the usual behaviors of floating point numbers. Operations resulting in NaN assigned to a field will result in "nan". Positive and negative overflow will result in "inf" and "-inf". Division by zero will result in a null field.

If you are using a search as an argument to the **eval** command and functions, you cannot use a saved search name; you must pass a literal search string or a field that contains a literal search string (like the 'search' field extracted from index=_audit events).

**Functions**

You can use a wide range of functions with the **eval** command. For general information about using functions, see Evaluation functions.

The following table lists the supported functions by type of function. Use the links in the table to learn more about each function, and to see examples.

<table>
<thead>
<tr>
<th>Type of function</th>
<th>Supported functions and syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison and Conditional functions</strong></td>
<td>case(X,&quot;Y&quot;,...) in(VALUE-LIST) nullif(X,Y,...)</td>
</tr>
<tr>
<td></td>
<td>cidrmatch(&quot;X&quot;,Y) like(TEXT, PATTERN) searchmatch(X)</td>
</tr>
<tr>
<td></td>
<td>coalesce(X,...) match(SUBJECT, &quot;REGEX&quot;) true() validate(X)</td>
</tr>
<tr>
<td></td>
<td>false() null()</td>
</tr>
<tr>
<td></td>
<td>if(X,Y,Z)</td>
</tr>
<tr>
<td><strong>Conversion functions</strong></td>
<td>printf(&quot;format&quot;,arguments) tonumber(NUMSTR,BASE) tostring(X,Y)</td>
</tr>
<tr>
<td></td>
<td>md5(X) sha256(X) sha512(X)</td>
</tr>
<tr>
<td></td>
<td>shal(X)</td>
</tr>
<tr>
<td><strong>Date and Time functions</strong></td>
<td>now() strftime(X,Y) time()</td>
</tr>
<tr>
<td></td>
<td>relative_time(X,Y) strptime(X,Y)</td>
</tr>
<tr>
<td><strong>Informational functions</strong></td>
<td>isbool(X) isnull(X) isstr(X)</td>
</tr>
<tr>
<td></td>
<td>isint(X) isnum(X) typeof(X)</td>
</tr>
<tr>
<td></td>
<td>isnotnull(X)</td>
</tr>
<tr>
<td><strong>Mathematical functions</strong></td>
<td>abs(X) floor(X)</td>
</tr>
<tr>
<td></td>
<td>pow(X,Y)</td>
</tr>
</tbody>
</table>
Type of function | Supported functions and syntax
--- | ---
ceiling(X) | ln(X)
exact (X) | log(X,Y)
exp(X) | pi()
commands (X) | mvfilter(X)
Multivalued eval functions | mvappend(X,...)
mvcount (MVFIELD) | mvfind(MVFIELD,"REGEX")
mvdedup(X) | mvindex(MVFIELD,STARTINDEX,ENDINDEX)
mvfilter(X) | mvjoin(MVFIELD,STR)
Statistical eval functions | max(X,...)
min(X,...) | random()
len(X) | rtrim(X,Y)
Text functions | lower(X)
spath(X,Y) | upper(X)
ltrim(X,Y) | substr(X,Y,Z)
replace(X,Y,Z) | trim(X,Y)
atan2(X,Y) | 
Trigonometry and Hyperbolic functions | acos(X)
acosh(X) | atanh(X)
asin(X) | sinh(X)
asinh(X) | cos(X)
atan(X) | tan(X)
atanh(X) | tanh(X)
hypot(X,Y) | 

Operators

The following table lists the basic operations you can perform with the eval command. For these evaluations to work, the values need to be valid for the type of operation. For example, with the exception of addition, arithmetic operations might not produce valid results if the values are not numerical. When concatenating values, Splunk software reads the values as strings, regardless of the value.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+ - * / %</td>
</tr>
<tr>
<td>Concatenation</td>
<td>.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AND OR NOT XOR &lt; &gt; &lt;= &gt;= != = == LIKE</td>
</tr>
</tbody>
</table>
Operators that produce numbers

- The plus (+) operator accepts two numbers for addition, or two strings for concatenation.
- The subtraction (-), multiplication (*), division (/), and modulus (%) operators accept two numbers.

Operators that produce strings

- The period (.) operator concatenates both strings and number. Numbers are concatenated in their string represented form.

Operators that produce booleans

- The AND, OR, and XOR operators accept two Boolean values.
- The <, >, <=, >=, !=, =, and == operators accept two numbers or two strings.
- In expressions, the single equal sign (=) is a synonym for the double equal sign (==).
- The LIKE operator accepts two strings. This is a pattern match similar to what is used in SQL. For example string LIKE pattern. The pattern operator supports literal text, a percent (%) character for a wildcard, and an underscore (_) character for a single character match. For example, field LIKE "a%b_" matches any string starting with a, followed by anything, followed by b, followed by one character.

Field names

To specify a field name with multiple words, you can either concatenate the words, or use single quotation marks when you specify the name. For example, to specify the field name Account ID you can specify AccountID or 'Account ID'.

To specify a field name with special characters, such as a period, use single quotation marks. For example, to specify the field name Last.Name use 'Last.Name'.

You can use the value of another field as the name of the destination field by using curly brackets, { }. For example, if you have an event with the following fields, aName=counter and aValue=1234. Use | eval {aName}=aValue to return counter=1234.
**Calculated fields**

You can use eval statements to define calculated fields by defining the eval statement in props.conf. If you are using Splunk Cloud, you can define calculated fields using Splunk Web, by choosing Settings > Fields > Calculated Fields. When you run a search, Splunk software evaluates the statements and creates fields in a manner similar to that of search time field extraction. Setting up calculated fields means that you no longer need to define the eval statement in a search string. Instead, you can search on the resulting calculated field directly.

You can use calculated fields to move your commonly used eval statements out of your search string and into props.conf, where they will be processed behind the scenes at search time. With calculated fields, you can change the search from:

```plaintext
sourcetype="cisco_esa" mailfrom=* | eval accountname=split(mailfrom,"@"), from_user=mvindex(accountname,0), from_domain=mvindex(accountname,-1) | table mailfrom, from_user, from_domain
```

to this search:

```plaintext
sourcetype="cisco_esa" mailfrom=* | table mailfrom, from_user, from_domain
```

In this example, the three eval statements that were in the search--that defined the accountname, from_user, and from_domain fields--are now computed behind the scenes when the search is run for any event that contains the extracted field mailfrom field. You can also search on those fields independently once they're set up as calculated fields in props.conf. You could search on from_domain=email.com, for example.

For more information about calculated fields, see About calculated fields in the Knowledge Manager Manual.

**Search event tokens**

If you are using the eval command in search event tokens, some of the evaluation functions might be unavailable or have a different behavior. See Custom logic for search tokens in Dashboards and Visualizations for information about the evaluation functions that you can use with search event tokens.
Basic Examples

1. Create a new field that contains the result of a calculation

Create a new field called velocity in each event. Calculate the velocity by dividing the values in the distance field by the values in the time field.

```
... | eval velocity=distance/time
```

2. Use the if function to analyze field values

Create a field called error in each event. Using the if function, set the value in the error field to OK if the status value is 200. Otherwise set the error field value to Problem.

```
... | eval error = if(status == 200, "OK", "Problem")
```

3. Convert values to lowercase

Create a new field in each event called low-user. Using the lower function, populate the field with the lowercase version of the values in the username field. Because the field name contains a dash (-), the name must be enclosed in single quotation marks.

```
... | eval 'low-user' = lower(username)
```

4. Use the value of one field as the name for a new field

In this example, use each value of the field counter to make a new field name. Assign to the new field the value of the Value field. See Field names under the Usage section.

```
index=perfmon sourcetype=Perfmon* counter=* Value=* | eval {counter} = Value
```

5. Set sum_of_areas to be the sum of the areas of two circles

```
... | eval sum_of_areas = pi() * pow(radius_a, 2) + pi() * pow(radius_b, 2)
```

6. Set status to some simple http error codes

```
... | eval error_msg = case(error == 404, "Not found", error == 500, "Internal Server Error", error == 200, "OK")
```
7. Concatenate values from two fields

Use the period (.) character to concatenate the values in first_name field with the values in the last_name field. Quotation marks are used to insert a space character between the two names. When concatenating, the values are read as strings, regardless of the actual value.

... | eval full_name = first_name.""" .last_name

8. Separate multiple eval operations with a comma

You can specify multiple eval operations by using a comma to separate the operations. In the following search the full_name evaluation uses the period (.) character to concatenate the values in the first_name field with the values in the last_name field. The low_name evaluation uses the lower function to convert the full_name evaluation into lowercase.

... | eval full_name = first_name.""" .last_name, low_name = lower(full_name)

9. Convert a numeric field value to a string with commas and 2 decimals

If the original value of x is 1000000, this returns x as 1,000,000.

... | eval x= tostring(x,"""commas"")

To include a currency symbol at the beginning of the string:

... | eval x="$".tostring(x,"""commas"")

This returns x as $1,000,000.

Extended Examples

1. Coalesce a field from two different source types, create a transaction of events

This example shows how you might coalesce a field from two different source types and use that to create a transaction of events. sourcetype=A has a field called number, and sourcetype=B has the same information in a field called subscriberNumber.

sourcetype=A OR sourcetype=B | eval phone=coalesce(number,subscriberNumber) | transaction phone maxspan=2m
The *eval* command is used to add a common field, called *phone*, to each of the events whether they are from *sourcetype=A* or *sourcetype=B*. The value of *phone* is defined, using the *coalesce()* function, as the values of *number* and *subscriberNumber*. The *coalesce()* function takes the value of the first non-NULL field (that means, it exists in the event).

Now, you're able to group events from either source type *A* or *B* if they share the same *phone* value.

2. *Separate events into categories, count and display minimum and maximum values*

This example uses recent earthquake data downloaded from the USGS Earthquakes website. The data is a comma separated ASCII text file that contains magnitude (mag), coordinates (latitude, longitude), region (place), and so forth, for each earthquake recorded.

You can download a current CSV file from the USGS Earthquake Feeds and upload the file to your Splunk instance if you want follow along with this example.

Earthquakes occurring at a depth of less than 70 km are classified as **shallow-focus** earthquakes, while those with a focal-depth between 70 and 300 km are commonly termed **mid-focus** earthquakes. In subduction zones, **deep-focus** earthquakes may occur at much greater depths (ranging from 300 up to 700 kilometers).

To classify recent earthquakes based on their depth, you use the following search.

```plaintext
source=all_month.csv | eval Description=case(depth<=70, "Shallow", depth>70 AND depth<=300, "Mid", depth>300, "Deep") | stats count min(mag) max(mag) by Description
```

The *eval* command is used to create a field called *Description*, which takes the value of "Shallow", "Mid", or "Deep" based on the Depth of the earthquake. The *case()* function is used to specify which ranges of the depth fits each description. For example, if the depth is less than 70 km, the earthquake is characterized as a shallow-focus quake; and the resulting *Description* is *Shallow*.

The search also pipes the results of the *eval* command into the *stats* command to count the number of earthquakes and display the minimum and maximum magnitudes for each *Description*. 
The results appear on the Statistics tab and look something like this:

<table>
<thead>
<tr>
<th>Description</th>
<th>count</th>
<th>min(Mag)</th>
<th>max(Mag)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep</td>
<td>35</td>
<td>4.1</td>
<td>6.7</td>
</tr>
<tr>
<td>Mid</td>
<td>635</td>
<td>0.8</td>
<td>6.3</td>
</tr>
<tr>
<td>Shallow</td>
<td>6236</td>
<td>-0.60</td>
<td>7.70</td>
</tr>
</tbody>
</table>

3. **Find IP addresses and categorize by network using eval functions **

This example uses the sample data from the Search Tutorial but should work with any format of Apache web access log. To try this example on your own Splunk instance, you must download the sample data and follow the instructions to **get the tutorial data into Splunk.** Use the time range **Yesterday** when you run the search.

In this search, you’re finding IP addresses and classifying the network they belong to.

```
sourcetype=access_* | eval network=if(cidrmatch("182.236.164.11/16", clientip), "local", "other")
```

This example uses the `cidrmatch()` function to compare the IP addresses in the `clientip` field to a subnet range. The search also uses the `if()` function, which says that if the value of `clientip` falls in the subnet range, then the `network` field value is `local`. Otherwise, `network=other`.

The `eval` command does not do any special formatting to your results. The command creates a new field based on the `eval` expression you specify.

In the fields sidebar, click on the `network` field. In the popup, next to **Selected** click **Yes** and close the popup. Now you can see, inline with your search results, which IP addresses are part of your `local` network and which are not. Your events list looks something like this:
Another option for formatting your results is to pipe the results of `eval` to the `table` command to display only the fields of interest to you.

**Note:** This example just illustrates how to use the `cidrmatch` function. If you want to classify your events and quickly search for those events, the better approach is to use event types. Read more about event types in the *Knowledge manager manual*.

4. Extract information from an event into a separate field, create a multivalue field

This example uses sample email data. You should be able to run this search on any email data by replacing the `sourcetype=cisco:esa` with the `sourcetype` value and the `mailfrom` field with email address field name in your data. For example, the email might be To, From, or Cc).

Use the email address field to extract the name and domain. The `eval` command in this search contains multiple expressions, separated by commas.

```bash
sourcetype="cisco:esa" mailfrom=* | eval accountname=split(mailfrom,"@"), from_user=mvindex(accountname,0), from_domain=mvindex(accountname,-1) | table mailfrom, from_user, from_domain
```

- The `split()` function is used to break the `mailfrom` field into a multivalue field called `accountname`. The first value of `accountname` is everything before the "@" symbol, and the second value is everything after.
- The `mvindex()` function is used to set `from_user` to the first value in `accountname` and to set `from_domain` to the second value in `accountname`.
- The results of the `eval` expressions are then piped into the `table` command.
You can see the the original `mailfrom` values and the new `from_user` and `from_domain` values in the results table. The results appear on the Statistics tab and look something like this:

<table>
<thead>
<tr>
<th>mailfrom</th>
<th>from_user</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:na.lui@sample.net">na.lui@sample.net</a></td>
<td>na.lui</td>
</tr>
<tr>
<td><a href="mailto:MAILER-DAEMON@hcp2mailsec.sample.net">MAILER-DAEMON@hcp2mailsec.sample.net</a></td>
<td>MAILER-DAEMON</td>
</tr>
<tr>
<td>M&amp;<a href="mailto:MService@example.com">MService@example.com</a></td>
<td>M&amp;MService</td>
</tr>
<tr>
<td><a href="mailto:AlexMartin@oursample.de">AlexMartin@oursample.de</a></td>
<td>AlexMartin</td>
</tr>
<tr>
<td><a href="mailto:Exit_Desk@sample.net">Exit_Desk@sample.net</a></td>
<td>Exit_Desk</td>
</tr>
<tr>
<td><a href="mailto:buttercup-forum+SEMAG8PUC4RETTUB@groups.com">buttercup-forum+SEMAG8PUC4RETTUB@groups.com</a></td>
<td>buttercup-forum+SEMAG8PUC4RETTUB</td>
</tr>
<tr>
<td><a href="mailto:eduardo.rodriguez@sample.net">eduardo.rodriguez@sample.net</a></td>
<td>eduardo.rodriguez</td>
</tr>
<tr>
<td><a href="mailto:VC00110489@techexamples.com">VC00110489@techexamples.com</a></td>
<td>VC00110489</td>
</tr>
</tbody>
</table>

Note: This example was written to demonstrate how to use an `eval` function to identify the individual values of a multivalue fields. Because this particular set of email data did not have any multivalue fields, the example creates a multivalue filed, `accountname`, from a single value field, `mailfrom`.

5. Categorize events using the `match` function

This example uses sample email data. You should be able to run this search on any email data by replacing the `sourcetype=cisco:esa` with the `sourcetype` value and the `mailfrom` field with email address field name in your data. For example, the email might be `To`, `From`, or `Cc`.

This example classifies where an email came from based on the email address domain. The .com, .net, and .org addresses are considered local, while anything else is considered abroad. There are many domain names. Of course, domains that are not .com, .net, or .org are not necessarily from abroad. This is just an example.

The `eval` command in this search contains multiple expressions, separated by commas.

```bash
sourcetype="cisco:esa" mailfrom=*| eval accountname=split(mailfrom,"@"), from_domain=mvindex(accountname,-1), location=if(match(from_domain, "[^\n\r\s]+.(com|net|org)"), "local", "abroad") | stats count BY location
```

The first half of this search is similar to previous example. The `split()` function is used to break up the email address in the `mailfrom` field. The `mvindex` function
defines the `from_domain` as the portion of the `mailfrom` field after the @ symbol.

Then, the `if()` and `match()` functions are used.

- If the `from_domain` value ends with a `.com`, `.net.`, or `.org`, the `location` field is assigned the value `local`.
- If `from_domain` does not match, `location` is assigned the value `abroad`.

The eval results are then piped into the `stats` command to count the number of results for each `location` value.

The results appear on the Statistics tab and look something like this:

<table>
<thead>
<tr>
<th>location</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>abroad</td>
<td>3543</td>
</tr>
<tr>
<td>local</td>
<td>14136</td>
</tr>
</tbody>
</table>

**Note:** This example merely illustrates using the `match()` function. If you want to classify your events and quickly search for those events, the better approach is to use event types. Read more about event types in the Knowledge manager manual.

6. **Convert the duration of transactions into more readable string formats**

This example uses the sample data from the Search Tutorial but should work with any format of Apache web access log. To try this example on your own Splunk instance, you must download the sample data and follow the instructions to get the tutorial data into Splunk. Use the time range `Yesterday` when you run the search.

When you use the `transaction` command, as shown in the following search, it calculates the length of time for the transaction. A new field, called `duration`, is automatically added to the results. The `duration` is the time between the first and last events in the transaction.

```
sourcetype=access_*  | transaction clientip maxspan=10m
```

In the **Interesting fields** list, click on the `duration` field to see the top 10 values for duration. The values are displayed in seconds. Click Yes to add the field to the **Selected fields** list.

You can use the `eval` command to reformat a numeric field into a more readable string format. The following search uses the `tostring()` function with the
"duration" option to convert the values in the duration field into a string formatted as HH:MM:SS.

sourcetype=access_* | transaction clientip maxspan=10m | eval durationstr=tostring(duration,"duration")

The search defines a new field, durationstr, for the reformatted duration values. In the Interesting fields list, click on the durationstr field and select Yes to add the field to the Selected fields list. The values for the fields now appear in the set of fields below each transaction. The following image shows how your search results should look:

![Image showing search results]

See also

Functions
Evaluation functions

Commands
where

Answers

Have questions? Visit Splunk Answers and see what questions and answers the Splunk community has using the eval command.