



Splunk® Enterprise Search Reference 9.1.1

Generated: 10/11/2023 5:49 pm

Table of Contents

Introduction.....	1
Welcome to the Search Reference.....	1
Understanding SPL syntax.....	2
How to use this manual.....	6
Quick Reference.....	8
Splunk Quick Reference Guide.....	8
Command quick reference.....	8
Commands by category.....	14
Command types.....	22
Splunk SPL for SQL users.....	26
SPL data types and clauses.....	31
Evaluation Functions.....	36
Evaluation functions.....	36
Comparison and Conditional functions.....	46
Conversion functions.....	58
Cryptographic functions.....	65
Date and Time functions.....	66
Informational functions.....	72
JSON functions.....	75
Mathematical functions.....	92
Multivalue eval functions.....	99
Statistical eval functions.....	109
Text functions.....	113
Trig and Hyperbolic functions.....	118
Statistical and Charting Functions.....	124
Statistical and charting functions.....	124
Aggregate functions.....	129
Event order functions.....	150
Multivalue stats and chart functions.....	156
Time functions.....	159
Time Format Variables and Modifiers.....	167
Date and time format variables.....	167
Time modifiers.....	170
Search Commands.....	176
abstract.....	176
accum.....	178
addcoltotals.....	179
addinfo.....	181
addtotals.....	183
analyzefields.....	187
anomalies.....	188
anomalousvalue.....	192

Table of Contents

Search Commands

anomalydetection.....	197
append.....	200
appendcols.....	204
appendpipe.....	206
arules.....	207
associate.....	209
autoregress.....	212
awssnsalert.....	213
bin.....	213
bucket.....	216
bucketdir.....	217
chart.....	218
cluster.....	232
cofilter.....	236
collect.....	238
concurrency.....	244
contingency.....	249
convert.....	252
correlate.....	256
ctable.....	258
datamodel.....	258
datamodelsimple.....	262
dbinspect.....	262
dbxquery.....	267
dedup.....	267
delete.....	270
delta.....	272
diff.....	276
entitymerge.....	278
erex.....	278
eval.....	281
eventcount.....	292
eventstats.....	294
extract.....	303
fieldformat.....	305
fields.....	309
fieldsummary.....	312
filldown.....	314
fillnull.....	315
findtypes.....	318
folderize.....	319
foreach.....	321
format.....	333
from.....	337
fromjson.....	340
gauge.....	342

Table of Contents

Search Commands

gentimes.....	344
geom.....	346
geomfilter.....	351
geostats.....	352
head.....	357
highlight.....	360
history.....	361
iconify.....	364
inputcsv.....	365
inputintelligence.....	368
inputlookup.....	368
iplocation.....	372
join.....	377
kmeans.....	383
kvform.....	385
loadjob.....	387
localize.....	390
localop.....	391
lookup.....	392
makecontinuous.....	398
makemv.....	400
makeresults.....	402
map.....	409
mcollect.....	411
metadata.....	416
metasearch.....	420
meventcollect.....	422
mpreview.....	425
msearch.....	429
mstats.....	429
multikv.....	442
multisearch.....	444
mvcombine.....	445
mvexpand.....	449
nomv.....	452
outlier.....	453
outputcsv.....	454
outputlookup.....	457
outputtext.....	462
overlap.....	463
pivot.....	464
predict.....	467
rangemap.....	473
rare.....	476
regex.....	479
reltime.....	481

Table of Contents

Search Commands

rename.....	484
replace.....	486
require.....	488
rest.....	489
return.....	492
reverse.....	494
rex.....	495
rtorder.....	500
run.....	501
savedsearch.....	501
script.....	503
scrub.....	505
search.....	506
searchtxn.....	515
selfjoin.....	516
sendalert.....	521
sendemail.....	523
set.....	528
setfields.....	530
sichart.....	531
sirare.....	532
sistats.....	533
sitimechart.....	535
sitop.....	536
snowincident.....	538
snowincidentstream.....	538
snowevent.....	538
snoweventstream.....	538
sort.....	539
spath.....	543
stats.....	549
strcat.....	561
streamstats.....	563
table.....	573
tags.....	576
tail.....	580
timechart.....	581
timewrap.....	596
tojson.....	600
top.....	605
transaction.....	608
transpose.....	618
trendline.....	624
tscollect.....	625
tstats.....	627
typeahead.....	639

Table of Contents

Search Commands	
typelearner.....	643
typer.....	644
union.....	646
uniq.....	650
untable.....	651
walklex.....	655
where.....	658
x11.....	661
xmlkv.....	662
xmlunescape.....	664
xpath.....	664
xseries.....	667
3rd party custom commands.....	672
Internal Commands	673
About internal commands.....	673
collapse.....	673
dump.....	674
findkeywords.....	675
makejson.....	677
mcatalog.....	680
noop.....	684
pjob.....	690
redistribute.....	692
runshellscript.....	698
Search in the CLI	700
About searches in the CLI.....	700
Syntax for searches in the CLI.....	700

Introduction

Welcome to the Search Reference

This manual is a reference guide for the Search Processing Language (SPL). In this manual you will find a catalog of the search commands with complete syntax, descriptions, and examples. Additionally, this manual includes quick reference information about the categories of commands, the functions you can use with commands, and how SPL relates to SQL.

Getting Started

If you are new to Splunk software and searching, start with the *Search Tutorial*. This tutorial introduces you to the Search & Reporting application. The tutorial guides you through uploading data to your Splunk deployment, searching your data, and building simple charts, reports, and dashboards.

After you complete the Search Tutorial, and before you start using Splunk software on your own data you should:

- Add data to your Splunk instance. See *Getting Data In*.
- Understand how indexing works and how data is processed. See *Managing Indexers and Clusters of Indexers*.
- Learn about fields and knowledge objects, such as hosts, source types, and event types. See the *Knowledge Manager Manual*.

Search Manual

The *Search Manual* is a companion manual to the *Search Reference*. The *Search Manual* contains detailed information about creating and optimizing searches.

- Types of searches
- Retrieving events
- Specifying time ranges
- Optimizing searches
- Using subsearches
- Creating statistical tables and charts
- Grouping and correlating events
- Predicting future events
- Managing jobs

Quick Reference Information

The [Quick Reference Guide](#) contains:

- Explanations about Splunk features
- Common search commands
- Tips on optimizing searches
- Functions for the `eval` and `stats` commands
- Search examples
- Regular expressions
- Formats for converting strings into timestamps

SPL commands

The Search Processing Language (SPL) includes a wide range of commands.

There are two quick reference guides for the commands:

- The [Command quick reference](#) topic contains an alphabetical list of each command, along with a brief description of what the command does and a link to the specific documentation for the command.
- The [Commands by category](#) topic organizes the commands by the type of action that the command performs. This topic contains a brief description of what the command does and a link to the specific documentation for the command.

SQL users

If you're familiar with SQL, see [Splunk SPL for SQL users](#) to see how to use your SQL knowledge to learn SPL.

Command syntax

Before you continue, see [Understanding SPL syntax](#) for the conventions and rules used in this manual.

Understanding SPL syntax

The following sections describe the syntax used for the Splunk **SPL commands**. For additional information about using keywords, phrases, wildcards, and regular expressions, see [Search command primer](#).

Required and optional arguments

SPL commands consist of required and optional arguments.

- Required arguments are shown in angle brackets < >.
- Optional arguments are enclosed in square brackets [].

Consider this command syntax:

```
bin [<bins-options>...] <field> [AS <newfield>]
```

The required argument is <field>. To use this command, at a minimum you must specify bin <field>.

The optional arguments are [<bins-options>...] and [AS <newfield>].

User input arguments

Consider this command syntax:

```
replace (<wc-string> WITH <wc-string>)... [IN <field-list>]
```

The user input arguments are: <wc-string> and <field-list>. The argument <wc-string> is an abbreviation for <wildcard-string> and indicates that the argument accepts a wildcard character in the string that you provide. See [Wildcards](#) in the [Search Reference](#).

Repeating arguments

Some arguments can be specified multiple times. The syntax displays ellipsis ... to specify which part of an argument can be repeated. The ellipsis always appear **immediately after** the part of the syntax that you can repeat.

Consider this command:

```
convert [timeformat=string] (<convert-function> [AS <field>])...
```

The required argument is `<convert-function>`, with an option to specify a field with the `[AS <field>]` clause.

Notice the ellipsis at the end of the syntax, just after the close parenthesis. In this example, the syntax that is inside the parenthesis can be repeated `<convert-function> [AS <field>]`.

In the following syntax, you can repeat the `<bins-options>`....

```
bin [<bins-options>...] <field> [AS <newfield>]
```

Grouped arguments

Sometimes the syntax must display arguments as a group to show that the set of arguments are used together. Parenthesis () are used to group arguments.

For example in this syntax:

```
replace (<wc-string> WITH <wc-string>)... [IN <field-list>]
```

The grouped argument is `(<wc-string> WITH <wc-string>)`.... This is a required set of arguments that you can repeat multiple times.

Keywords

Many commands use keywords with some of the arguments or options. Examples of keywords include:

- AS
- BY
- OVER
- WHERE

You can specify these keywords in uppercase or lowercase in your search. However, for readability, the syntax in the Splunk documentation uses uppercase on all keywords.

Quoted elements

If an element is in quotation marks, you must include that element in your search. The most common quoted elements are parenthesis.

Consider the syntax for the `chart` command:

```
chart [<chart-options>] [agg=<stats-agg-term>]  
( <stats-agg-term> | <sparkline-agg-term> | "(<eval-expression>)" )...
```

[BY <row-split> <column-split>] | [OVER <row-split>] [BY <column-split>]]

There are quotation marks on the parenthesis surrounding the <eval-expression>. This means that you must enclose the <eval-expression> in parenthesis in your search.

In the following search example, the <eval-expression> is avg(size)/max(delay) and is enclosed in parenthesis.

```
... | chart eval(avg(size)/max(delay)) AS ratio BY host user
```

Argument order

In the command syntax, the command arguments are presented in the order in which the arguments are meant to be used.

In the descriptions of the arguments, the **Required arguments** and **Optional argument** sections, the arguments are listed alphabetically. For each argument, there is a **Syntax** and **Description**. Additionally, for Optional arguments, there might be a **Default**.

Data types

The nomenclature used for the data types in SPL syntax are described in the following table.

Syntax	Data type	Notes
<bool>	boolean	Use true or false. Other variations are accepted. For example, for true you can also use 't', 'T', 'TRUE', 'yes', or the number one (1). For false you can also specify 'no', the number zero (0), and variations of the word false, similar to the variations of the word true.
<field>	A field name. You cannot specify a wild card for the field name.	See <wc-field>.
<int> or <integer>	An integer that can be a positive or negative value.	Sometimes referred to as a "signed" integer. See <unsigned int>.
<string>	string	See <wc-string>.
<unsigned int>	unsigned integer	An unsigned integer must be positive value. Unsigned integers can be larger numbers than signed integers.
<wc-field>	A field name or a partial name with a wildcard character to specify multiple, similarly named fields.	Use the asterisk (*) character as the wildcard character.
<wc-string>	A string value or partial string value with a wildcard character.	Use the asterisk (*) character as the wildcard character.

Boolean operators

When a boolean operator is included in the syntax of a command, you must always specify the operator in uppercase. Boolean operators include:

- AND
- OR
- NOT

To learn more about the order in which boolean expressions are evaluated, along with some examples, see Boolean expressions in the *Search Manual*.

To learn more about the NOT operator, see Difference between NOT and != in the *Search Manual*.

BY clauses

A <by-clause> and a <split-by-clause> are not the same argument.

When you use a <by-clause>, one row is returned for each distinct value <by-clause> field. A <by-clause> displays each unique item in a separate **row**. Think of the <by-clause> as a grouping.

The <split-by-clause> displays each unique item in a separate **column**. Think of the <split-by-clause> as a splitting or dividing.

Wildcard characters (*) are not accepted in BY clauses.

Fields and wildcard fields

When the syntax contains <field> you specify a field name from your events.

Consider this syntax:

```
bin [<bins-options>...] <field> [AS <newfield>]
```

The <field> argument is required. You can specify that the field displays a different name in the search results by using the [AS <newfield>] argument. This argument is optional.

For example, if the field is `categoryId` and you want the field to be named `CategoryID` in the output, you would specify:

```
categoryId AS CategoryID
```

The <wc-field> argument indicates that you can use wild card characters when specifying field names. For example, if you have a set of fields that end with "log" you can specify `*log` to return all of those fields.

If you use a wild card character in the middle of a value, especially as a wild card for punctuation, the results might be unpredictable.

Repeating expressions

With many commands you can specify multiple expressions. Some commands use a space between expressions, while other commands use a comma between expressions. In the syntax for a command you will see something like <field-list> to indicate that you can specify one or more expressions.

For example, the `stats` command includes a <field-list> argument. The list of fields must be separated by commas:

```
sourcetype=access_* | stats count BY status, host
```

With the `outlier` command, the <field-list> argument expects the field names to be space-separated:

```
...| outlier bytes clientip
```

See also

In the *Search Manual*:

- Anatomy of a search
- Wildcards
- Field expressions
- Quotes and escaping characters

How to use this manual

This manual serves as a reference guide for the Splunk user who is looking for a catalog of the search commands with complete syntax, descriptions, and examples for usage.

Quick Reference Information

There are two quick reference guides for the commands:

- The [Command quick reference](#) topic contains an alphabetical list of each command, along with a brief description of what the command does and a link to the specific documentation for the command.
- The [Commands by category](#) topic organizes the commands by the type of action that the command performs. This topic contains a brief description of what the command does and a link to the specific documentation for the command.

Functions

Command topics

Each search command topic contains the following sections: Description, Syntax, Examples, and See also. Many of the command topics also have a Usage section.

Description

Describes what the command is used for. This section might include details about how to use the command. For more complex commands, there might be a separate Usage section.

Syntax

The syntax includes the complete syntax for each search command, and a description for each argument. Some commands have arguments that have a set of options that you can specify. Each of these sets of options follow the argument descriptions.

Required arguments

Displays the syntax and describes the required arguments.

Optional arguments

Displays the syntax and describes the optional arguments. Default values, if applicable, are also listed.

Usage

Contains additional information about using the command.

Examples

This section includes examples of how to use the command.

See also

This section contains links to all related or similar commands.

Command syntax conventions

The command arguments are presented in the syntax in the order in which the arguments are meant to be used.

Arguments are either Required or Optional and are listed alphabetically under their respective subheadings. For each argument, there are **Syntax** and **Description** sections. Additionally, there might be other sections, such as **Default** that provide information about the argument.

See [Understanding SPL syntax](#).

Formatting conventions

Italic

When referring to another manual in the set of Splunk documentation, the name of the manual appears in italic.

Quick Reference

Splunk Quick Reference Guide

The Splunk Quick Reference Guide is a six-page reference card that provides fundamental search concepts, commands, functions, and examples. This guide is available online as a PDF file.

Note: The examples in this quick reference use a leading ellipsis (...) to indicate that there is a search before the pipe operator. A leading pipe indicates that the search command is a generating command and prevents the command-line interface and Splunk Web from prepending the `search` command to your search.

See also

- [Search commands by category](#)

Splunk Answers

If you cannot find what you are looking for in this search language reference, check out Splunk Answers and see what questions and answers other Splunk users have about the search language.

Command quick reference

The table below lists all of the search commands in alphabetical order. There is a short description of the command and links to related commands. For the complete syntax, usage, and detailed examples, click the command name to display the specific topic for that command.

Some of these commands share functions. For a list of the functions with descriptions and examples, see [Evaluation functions](#) and [Statistical and charting functions](#).

If you don't find a command in the table, that command might be part of a third-party app or add-on. For information about commands contributed by apps and add-ons, see the documentation on [Splunkbase](#).

Command	Description	Related commands
abstract	Produces a summary of each search result.	highlight
accum	Keeps a running total of the specified numeric field.	autoregress , delta , trendline , streamstats
addcoltotals	Computes an event that contains sum of all numeric fields for previous events.	addtotals , stats
addinfo	Add fields that contain common information about the current search.	search
addtotals	Computes the sum of all numeric fields for each result.	addcoltotals , stats
analyzefields	Analyze numerical fields for their ability to predict another discrete field.	anomalousvalue
anomalies	Computes an "unexpectedness" score for an event.	anomalousvalue , cluster , kmeans , outlier
anomalousvalue	Finds and summarizes irregular, or uncommon, search results.	

Command	Description	Related commands
		analyzefields, anomalies, cluster, kmeans, outlier
anomalydetection	Identifies anomalous events by computing a probability for each event and then detecting unusually small probabilities.	analyzefields, anomalies, anomalousvalue, cluster, kmeans, outlier
append	Appends subsearch results to current results.	appendcols, appendcsv, appendlookup, join, set
appendcols	Appends the fields of the subsearch results to current results, first results to first result, second to second, etc.	append, appendcsv, join, set
appendpipe	Appends the result of the subpipeline applied to the current result set to results.	append, appendcols, join, set
arules	Finds association rules between field values.	associate, correlate
associate	Identifies correlations between fields.	correlate, contingency
autoregress	Sets up data for calculating the moving average.	accum, autoregress, delta, trendline, streamstats
bin (bucket)	Puts continuous numerical values into discrete sets.	chart, timechart
bucketdir	Replaces a field value with higher-level grouping, such as replacing filenames with directories.	cluster, dedup
chart	Returns results in a tabular output for charting. See also, Statistical and charting functions .	bin, sichart, timechart
cluster	Clusters similar events together.	anomalies, anomalousvalue, cluster, kmeans, outlier
cofilter	Finds how many times field1 and field2 values occurred together.	associate, correlate
collect	Puts search results into a summary index.	overlap
concurrency	Uses a duration field to find the number of "concurrent" events for each event.	timechart
contingency	Builds a contingency table for two fields.	associate, correlate
convert	Converts field values into numerical values.	eval
correlate	Calculates the correlation between different fields.	associate, contingency
datamodel	Examine data model or data model dataset and search a data model dataset.	pivot
dbinspect	Returns information about the specified index.	
dedup	Removes subsequent results that match a specified criteria.	uniq
delete	Delete specific events or search results.	
delta	Computes the difference in field value between nearby results.	accum, autoregress, trendline, streamstats
diff	Returns the difference between two search results.	
erex	Allows you to specify example or counter example values to automatically extract fields that have similar values.	extract, kvform, multikv, regex, rex, xmlkv
eval	Calculates an expression and puts the value into a field. See also, Evaluation functions .	where

Command	Description	Related commands
<code>eventcount</code>	Returns the number of events in an index.	<code>dbinspect</code>
<code>eventstats</code>	Adds summary statistics to all search results.	<code>stats</code>
<code>extract (kv)</code>	Extracts field-value pairs from search results.	<code>kvform, multikv, xmlkv, rex</code>
<code>fieldformat</code>	Expresses how to render a field at output time without changing the underlying value.	<code>eval, where</code>
<code>fields</code>	Keeps or removes fields from search results based on the field list criteria.	
<code>fieldsummary</code>	Generates summary information for all or a subset of the fields.	<code>analyzefields, anomalies, anomalousvalue, stats</code>
<code>filldown</code>	Replaces NULL values with the last non-NUL value.	<code>fillnull</code>
<code>fillnull</code>	Replaces null values with a specified value.	
<code>findtypes</code>	Generates a list of suggested event types.	<code>typer</code>
<code>folderize</code>	Creates a higher-level grouping, such as replacing filenames with directories.	
<code>foreach</code>	Run a templated streaming subsearch for each field in a wildcarded field list.	<code>eval</code>
<code>format</code>	Takes the results of a subsearch and formats them into a single result.	
<code>from</code>	Retrieves data from a dataset, such as a data model dataset, a CSV lookup, a KV Store lookup, a saved search, or a table dataset.	
<code>gauge</code>	Transforms results into a format suitable for display by the Gauge chart types.	
<code>gentimes</code>	Generates time-range results.	
<code>geom</code>	Adds a field, named <code>geom</code> , to each event. This field contains geographic data structures for polygon geometry in JSON and is used for the choropleth map visualization.	<code>geomfilter</code>
<code>geomfilter</code>	Accepts two points that specify a bounding box for clipping a choropleth map. Points that fall outside of the bounding box are filtered out.	<code>geom</code>
<code>geostats</code>	Generate statistics which are clustered into geographical bins to be rendered on a world map.	<code>stats, xyseries</code>
<code>head</code>	Returns the first number <code>n</code> of specified results.	<code>reverse, tail</code>
<code>highlight</code>	Highlights the specified terms.	<code>iconify</code>
<code>history</code>	Returns a history of searches formatted as an events list or as a table.	<code>search</code>
<code>iconify</code>	Displays a unique icon for each different value in the list of fields that you specify.	<code>highlight</code>
<code>inputcsv</code>	Loads search results from the specified CSV file.	<code>loadjob, outputcsv</code>
<code>inputlookup</code>	Loads search results from a specified static lookup table.	<code>inputcsv, join, lookup, outputlookup</code>
<code>iplocation</code>	Extracts location information from IP addresses.	
<code>join</code>	Combine the results of a subsearch with the results of a main search.	<code>appendcols, lookup, selfjoin</code>
<code>kmeans</code>	Performs k-means clustering on selected fields.	

Command	Description	Related commands
		anomalies, anomalousvalue, cluster, outlier
kvform	Extracts values from search results, using a form template.	extract, kvform, multikv, xmlkv, rex
loadjob	Loads events or results of a previously completed search job.	inputcsv
localize	Returns a list of the time ranges in which the search results were found.	map, transaction
localop	Run subsequent commands, that is all commands following this, locally and not on remote peers.	
lookup	Explicitly invokes field value lookups.	
makecontinuous	Makes a field that is supposed to be the x-axis continuous (invoked by chart/timechart)	chart, timechart
makemv	Change a specified field into a multivalued field during a search.	mvcombine, mvexpand, nomv
makeresults	Creates a specified number of empty search results.	
map	A looping operator, performs a search over each search result.	
mcollect	Converts search results into metric data and inserts the data into a metric index on the search head.	collect, meventcollect
metadata	Returns a list of source, sourcetypes, or hosts from a specified index or distributed search peer.	dbinspect
metasearch	Retrieves event metadata from indexes based on terms in the logical expression.	metadata, search
meventcollect	Converts search results into metric data and inserts the data into a metric index on the indexers.	collect, mcollect
mpreview	Returns a preview of the raw metric data points in a specified metric index that match a provided filter.	mcatalog, mstats, msearch
msearch	Alias for the mpreview command.	mcatalog, mstats, mpreview
mstats	Calculates statistics for the measurement, metric_name, and dimension fields in metric indexes.	stats, tstats
multikv	Extracts field-values from table-formatted events.	
multisearch	Run multiple streaming searches at the same time.	append, join
mvcombine	Combines events in search results that have a single differing field value into one result with a multivalue field of the differing field.	mvexpand, makemv, nomv
mvexpand	Expands the values of a multivalue field into separate events for each value of the multivalue field.	mvcombine, makemv, nomv
nomv	Changes a specified multivalued field into a single-value field at search time.	makemv, mvcombine, mvexpand
outlier	Removes outlying numerical values.	anomalies, anomalousvalue, cluster, kmeans
outputcsv	Outputs search results to a specified CSV file.	inputcsv, outputtext
outputlookup	Writes search results to the specified static lookup table.	inputlookup, lookup, outputcsv
outputtext	Outputs the raw text field (<code>_raw</code>) of results into the <code>_xml</code> field.	outputcsv

Command	Description	Related commands
<code>overlap</code>	Finds events in a summary index that overlap in time or have missed events.	<code>collect</code>
<code>pivot</code>	Run pivot searches against a particular data model dataset.	<code>datamodel</code>
<code>predict</code>	Enables you to use time series algorithms to predict future values of fields.	<code>x11</code>
<code>rangemap</code>	Sets RANGE field to the name of the ranges that match.	
<code>rare</code>	Displays the least common values of a field.	<code>sirare, stats, top</code>
<code>redistribute</code>	Implements parallel reduce search processing to shorten the search runtime of high-cardinality dataset searches.	
<code>regex</code>	Removes results that do not match the specified regular expression.	<code>rex, search</code>
<code>reltime</code>	Converts the difference between 'now' and '_time' to a human-readable value and adds this value to the field, 'reltime', in your search results.	<code>convert</code>
<code>rename</code>	Renames a specified field; wildcards can be used to specify multiple fields.	
<code>replace</code>	Replaces values of specified fields with a specified new value.	
<code>require</code>	Causes a search to fail if the queries and commands that precede it in the search string return zero events or results.	
<code>rest</code>	Access a REST endpoint and display the returned entities as search results.	
<code>return</code>	Specify the values to return from a subsearch.	<code>format, search</code>
<code>reverse</code>	Reverses the order of the results.	<code>head, sort, tail</code>
<code>rex</code>	Specify a Perl regular expression named groups to extract fields while you search.	<code>extract, kvform, multikv, xmlkv, regex</code>
<code>rtorder</code>	Buffers events from real-time search to emit them in ascending time order when possible.	
<code>savedsearch</code>	Returns the search results of a saved search.	
<code>script (run)</code>	Runs an external Perl or Python script as part of your search.	
<code>scrub</code>	Anonymizes the search results.	
<code>search</code>	Searches indexes for matching events.	
<code>searchtxn</code>	Finds transaction events within specified search constraints.	<code>transaction</code>
<code>selfjoin</code>	Joins results with itself.	<code>join</code>
<code>sendemail</code>	Emails search results to a specified email address.	
<code>set</code>	Performs set operations (union, diff, intersect) on subsearches.	<code>append, appendcols, join, diff</code>
<code>setfields</code>	Sets the field values for all results to a common value.	<code>eval, fillnull, rename</code>
<code>sichart</code>	Summary indexing version of the chart command.	<code>chart, sitimechart, timechart</code>
<code>sirare</code>	Summary indexing version of the rare command.	<code>rare</code>
<code>sistats</code>	Summary indexing version of the stats command.	<code>stats</code>
<code>sitimechart</code>	Summary indexing version of the timechart command.	<code>chart, sichart, timechart</code>

Command	Description	Related commands
<code>sitop</code>	Summary indexing version of the top command.	<code>top</code>
<code>sort</code>	Sorts search results by the specified fields.	<code>reverse</code>
<code>spath</code>	Provides a straightforward means for extracting fields from structured data formats, XML and JSON.	<code>xpath</code>
<code>stats</code>	Provides statistics, grouped optionally by fields. See also, Statistical and charting functions .	<code>eventstats, top, rare</code>
<code>strcat</code>	Concatenates string values.	
<code>streamstats</code>	Adds summary statistics to all search results in a streaming manner.	<code>eventstats, stats</code>
<code>table</code>	Creates a table using the specified fields.	<code>fields</code>
<code>tags</code>	Annotates specified fields in your search results with tags.	<code>eval</code>
<code>tail</code>	Returns the last number n of specified results.	<code>head, reverse</code>
<code>timechart</code>	Create a time series chart and corresponding table of statistics. See also, Statistical and charting functions .	<code>chart, bucket</code>
<code>timewrap</code>	Displays, or wraps, the output of the <code>timechart</code> command so that every timewrap-span range of time is a different series.	<code>timechart</code>
<code>tojson</code>	Converts events into JSON objects.	
<code>top</code>	Displays the most common values of a field.	<code>rare, stats</code>
<code>transaction</code>	Groups search results into transactions.	
<code>transpose</code>	Reformats rows of search results as columns.	
<code>trendline</code>	Computes moving averages of fields.	<code>timechart</code>
<code>tscollect</code>	Writes results into tsidx file(s) for later use by the tstats command.	<code>collect, stats, tstats</code>
<code>tstats</code>	Calculates statistics over tsidx files created with the tscollect command.	<code>stats, tscollect</code>
<code>typeahead</code>	Returns typeahead information on a specified prefix.	
<code>typelearner</code>	Deprecated. Use <code>findtypes</code> instead. Generates suggested eventtypes.	<code>typer</code>
<code>typer</code>	Calculates the eventtypes for the search results.	<code>findtypes</code>
<code>union</code>	Merges the results from two or more datasets into one dataset.	
<code>uniq</code>	Removes any search that is an exact duplicate with a previous result.	<code>dedup</code>
<code>untable</code>	Converts results from a tabular format to a format similar to <code>stats</code> output. Inverse of <code>xseries</code> and <code>maketable</code> .	
<code>walklex</code>	Generates a list of terms or indexed fields from each bucket of event indexes.	<code>metadata, tstats</code>
<code>where</code>	Performs arbitrary filtering on your data. See also, Evaluations functions .	<code>eval</code>
<code>x11</code>	Enables you to determine the trend in your data by removing the seasonal pattern.	<code>predict</code>
<code>xmlkv</code>	Extracts XML key-value pairs.	<code>extract, kvform, multikv, rex</code>
<code>xmlunescape</code>	Unescapes XML.	
<code>xpath</code>	Redefines the XML path.	

Command	Description	Related commands
<code>xyseries</code>	Converts results into a format suitable for graphing.	

Commands by category

The following tables list all the search commands, categorized by their usage. Some commands fit into more than one category based on the options that you specify.

Correlation

These commands can be used to build correlation searches.

Command	Description
<code>append</code>	Appends subsearch results to current results.
<code>appendcols</code>	Appends the fields of the subsearch results to current results, first results to first result, second to second, etc.
<code>appendpipe</code>	Appends the result of the subpipeline applied to the current result set to results.
<code>arules</code>	Finds association rules between field values.
<code>associate</code>	Identifies correlations between fields.
<code>contingency, counttable, ctable</code>	Builds a contingency table for two fields.
<code>correlate</code>	Calculates the correlation between different fields.
<code>diff</code>	Returns the difference between two search results.
<code>join</code>	Combines the results from the main results pipeline with the results from a subsearch.
<code>lookup</code>	Explicitly invokes field value lookups.
<code>selfjoin</code>	Joins results with itself.
<code>set</code>	Performs set operations (union, diff, intersect) on subsearches.
<code>stats</code>	Provides statistics, grouped optionally by fields. See Statistical and charting functions .
<code>transaction</code>	Groups search results into transactions.

Data and indexes

These commands can be used to learn more about your data, add and delete data sources, or manage the data in your summary indexes.

View data

These commands return information about the data you have in your indexes. They do not modify your data or indexes in any way.

Command	Description
<code>datamodel</code>	Return information about a data model or data model object.
<code>dbinspect</code>	Returns information about the specified index.

Command	Description
<code>eventcount</code>	Returns the number of events in an index.
<code>metadata</code>	Returns a list of source, sourcetypes, or hosts from a specified index or distributed search peer.
<code>typeahead</code>	Returns typeahead information on a specified prefix.

Manage data

These are some commands you can use to add data sources to or delete specific data from your indexes.

Command	Description
<code>delete</code>	Delete specific events or search results.

Manage summary indexes

These commands are used to create and manage your summary indexes.

Command	Description
<code>collect, stash</code>	Puts search results into a summary index.
<code>overlap</code>	Finds events in a summary index that overlap in time or have missed events.
<code>sichart</code>	Summary indexing version of chart. Computes the necessary information for you to later run a chart search on the summary index.
<code>sirare</code>	Summary indexing version of rare. Computes the necessary information for you to later run a rare search on the summary index.
<code>sistats</code>	Summary indexing version of stats. Computes the necessary information for you to later run a stats search on the summary index.
<code>sitimechart</code>	Summary indexing version of timechart. Computes the necessary information for you to later run a timechart search on the summary index.
<code>sitop</code>	Summary indexing version of top. Computes the necessary information for you to later run a top search on the summary index.

Fields

These are commands you can use to add, extract, and modify fields or field values. The most useful command for manipulating fields is `eval` and its [statistical and charting functions](#).

Add fields

Use these commands to add new fields.

Command	Description
<code>accum</code>	Keeps a running total of the specified numeric field.
<code>addinfo</code>	Add fields that contain common information about the current search.
<code>addtotals</code>	Computes the sum of all numeric fields for each result.
<code>delta</code>	Computes the difference in field value between nearby results.
<code>eval</code>	Calculates an expression and puts the value into a field. See also, evaluation functions .

Command	Description
<code>iplocation</code>	Adds location information, such as city, country, latitude, longitude, and so on, based on IP addresses.
<code>lookup</code>	For configured lookup tables, explicitly invokes the field value lookup and adds fields from the lookup table to the events.
<code>multikv</code>	Extracts field-values from table-formatted events.
<code>rangemap</code>	Sets RANGE field to the name of the ranges that match.
<code>strcat</code>	Concatenates string values and saves the result to a specified field.

Extract fields

These commands provide different ways to extract new fields from search results.

Command	Description
<code>erex</code>	Allows you to specify example or counter example values to automatically extract fields that have similar values.
<code>extract, kv</code>	Extracts field-value pairs from search results.
<code>kvform</code>	Extracts values from search results, using a form template.
<code>rex</code>	Specify a Perl regular expression named groups to extract fields while you search.
<code>spath</code>	Provides a straightforward means for extracting fields from structured data formats, XML and JSON.
<code>xmlkv</code>	Extracts XML key-value pairs.

Modify fields and field values

Use these commands to modify fields or their values.

Command	Description
<code>convert</code>	Converts field values into numerical values.
<code>filldown</code>	Replaces NULL values with the last non-NUL value.
<code>fillnull</code>	Replaces null values with a specified value.
<code>makemv</code>	Change a specified field into a multivalue field during a search.
<code>nomv</code>	Changes a specified multivalue field into a single-value field at search time.
<code>reltime</code>	Converts the difference between 'now' and '_time' to a human-readable value and adds adds this value to the field, 'reltime', in your search results.
<code>rename</code>	Renames a specified field. Use wildcards to specify multiple fields.
<code>replace</code>	Replaces values of specified fields with a specified new value.

Find anomalies

These commands are used to find anomalies in your data. Either search for uncommon or outlying events and fields or cluster similar events together.

Command	Description
<code>analyzefields, af</code>	Analyze numerical fields for their ability to predict another discrete field.

Command	Description
<code>anomalies</code>	Computes an "unexpectedness" score for an event.
<code>anomalousvalue</code>	Finds and summarizes irregular, or uncommon, search results.
<code>anomalydetection</code>	Identifies anomalous events by computing a probability for each event and then detecting unusually small probabilities.
<code>cluster</code>	Clusters similar events together.
<code>kmeans</code>	Performs k-means clustering on selected fields.
<code>outlier</code>	Removes outlying numerical values.
<code>rare</code>	Displays the least common values of a field.

Geographic and location

These commands add geographical information to your search results.

Command	Description
<code>iplocation</code>	Returns location information, such as city, country, latitude, longitude, and so on, based on IP addresses.
<code>geom</code>	Adds a field, named "geom", to each event. This field contains geographic data structures for polygon geometry in JSON and is used for choropleth map visualization. This command requires an external lookup with <code>external_type=geo</code> to be installed.
<code>geomfilter</code>	Accepts two points that specify a bounding box for clipping choropleth maps. Points that fall outside of the bounding box are filtered out.
<code>geostats</code>	Generate statistics which are clustered into geographical bins to be rendered on a world map.

Metrics

These commands work with metrics data.

Command	Description
<code>mcollect</code>	Converts events into metric data points and inserts the data points into a metric index on the search head.
<code>meventcollect</code>	Converts events into metric data points and inserts the data points into a metric index on indexer tier.
<code>mpreview, msearch</code>	Provides samples of the raw metric data points in the metric time series in your metrics indexes. Helps you troubleshoot your metrics data.
<code>mstats</code>	Calculates visualization-ready statistics for the <code>measurement</code> , <code>metric_name</code> , and <code>dimension</code> fields in metric indexes.

Prediction and trending

These commands predict future values and calculate trendlines that can be used to create visualizations.

Command	Description
<code>predict</code>	Enables you to use time series algorithms to predict future values of fields.
<code>trendline</code>	Computes moving averages of fields.
<code>x11</code>	Enables you to determine the trend in your data by removing the seasonal pattern.

Reports

These commands are used to build **transforming searches**. These commands return statistical data tables that are required for charts and other kinds of data visualizations.

Command	Description
<code>addtotals</code>	Computes the sum of all numeric fields for each result.
<code>autoregress</code>	Prepares your events for calculating the autoregression, or moving average, based on a field that you specify.
<code>bin, discretize</code>	Puts continuous numerical values into discrete sets.
<code>chart</code>	Returns results in a tabular output for charting. See also, Statistical and charting functions .
<code>contingency, counttable, ctable</code>	Builds a contingency table for two fields.
<code>correlate</code>	Calculates the correlation between different fields.
<code>eventcount</code>	Returns the number of events in an index.
<code>eventstats</code>	Adds summary statistics to all search results.
<code>gauge</code>	Transforms results into a format suitable for display by the Gauge chart types.
<code>makecontinuous</code>	Makes a field that is supposed to be the x-axis continuous (invoked by <code>chart/timechart</code>)
<code>mstats</code>	Calculates statistics for the measurement, metric_name, and dimension fields in metric indexes.
<code>outlier</code>	Removes outlying numerical values.
<code>rare</code>	Displays the least common values of a field.
<code>stats</code>	Provides statistics, grouped optionally by fields. See also, Statistical and charting functions .
<code>streamstats</code>	Adds summary statistics to all search results in a streaming manner.
<code>timechart</code>	Create a time series chart and corresponding table of statistics. See also, Statistical and charting functions .
<code>top</code>	Displays the most common values of a field.
<code>trendline</code>	Computes moving averages of fields.
<code>tstats</code>	Performs statistical queries on indexed fields in <code>tsidx</code> files.
<code>untable</code>	Converts results from a tabular format to a format similar to <code>stats</code> output. Inverse of <code>xyseries</code> and <code>maketable</code> .
<code>xyseries</code>	Converts results into a format suitable for graphing.

Results

These commands can be used to manage search results. For example, you can append one set of results with another, filter more events from the results, reformat the results, and so on.

Alerting

Use this command to email the results of a search.

Command	Description
<code>sendemail</code>	Emails search results, either inline or as an attachment, to one or more specified email addresses.

Command	Description

Appending

Use these commands to append one set of results with another set or to itself.

Command	Description
<code>append</code>	Appends subsearch results to current results.
<code>appendcols</code>	Appends the fields of the subsearch results to current results, first results to first result, second to second, and so on.
<code>join</code>	SQL-like joining of results from the main results pipeline with the results from the subpipeline.
<code>selfjoin</code>	Joins results with itself.

Filtering

Use these commands to remove more events or fields from your current results.

Command	Description
<code>dedup</code>	Removes subsequent results that match a specified criteria.
<code>fields</code>	Removes fields from search results.
<code>from</code>	Retrieves data from a dataset, such as a data model dataset, a CSV lookup, a KV Store lookup, a saved search, or a table dataset.
<code>mvcombine</code>	Combines events in search results that have a single differing field value into one result with a multivalue field of the differing field.
<code>regex</code>	Removes results that do not match the specified regular expression.
<code>searchtxn</code>	Finds transaction events within specified search constraints.
<code>table</code>	Creates a table using the specified fields.
<code>uniq</code>	Removes any search that is an exact duplicate with a previous result.
<code>where</code>	Performs arbitrary filtering on your data. See also, Evaluation functions .

Formatting

Use these commands to reformat your current results.

Command	Description
<code>fieldformat</code>	Uses <code>eval</code> expressions to change the format of field values when they are rendered without changing their underlying values. Does not apply to exported data.
<code>transpose</code>	Reformats rows of search results as columns. Useful for fixing X- and Y-axis display issues with charts, or for turning sets of data into a series to produce a chart.
<code>untable</code>	Converts results from a tabular format to a format similar to <code>stats</code> output. Inverse of <code>xseries</code> and <code>maketable</code> .
<code>xseries</code>	Converts results into a format suitable for graphing.

Generating

Use these commands to generate or return events.

Command	Description
<code>gentimes</code>	Returns results that match a time-range.
<code>loadjob</code>	Loads events or results of a previously completed search job.
<code>makeresults</code>	Creates a specified number of empty search results.
<code>mvexpand</code>	Expands the values of a multivalue field into separate events for each value of the multivalue field.
<code>savedsearch</code>	Returns the search results of a saved search.
<code>search</code>	Searches indexes for matching events. This command is implicit at the start of every search pipeline that does not begin with another generating command.

Grouping

Use these commands to group or classify the current results.

Command	Description
<code>cluster</code>	Clusters similar events together.
<code>kmeans</code>	Performs k-means clustering on selected fields.
<code>mvexpand</code>	Expands the values of a multivalue field into separate events for each value of the multivalue field.
<code>transaction</code>	Groups search results into transactions.
<code>typelearner</code>	Generates suggested eventtypes.
<code>typer</code>	Calculates the eventtypes for the search results.

Reordering

Use these commands to change the order of the current search results.

Command	Description
<code>head</code>	Returns the first number n of specified results.
<code>reverse</code>	Reverses the order of the results.
<code>sort</code>	Sorts search results by the specified fields.
<code>tail</code>	Returns the last number N of specified results

Reading

Use these commands to read in results from external files or previous searches.

Command	Description
<code>inputcsv</code>	Loads search results from the specified CSV file.
<code>inputlookup</code>	Loads search results from a specified static lookup table.
<code>loadjob</code>	Loads events or results of a previously completed search job.

Writing

Use these commands to define how to output current search results.

Command	Description
collect, stash	Puts search results into a summary index.
meventcollect	Converts events into metric data points and inserts the data points into a metric index on indexer tier.
mcollect	Converts events into metric data points and inserts the data points into a metric index on the search head.
outputcsv	Outputs search results to a specified CSV file.
outputlookup	Writes search results to the specified static lookup table.
outputtext	Ouputs the raw text field (_raw) of results into the _xml field.
sendemail	Emails search results, either inline or as an attachment, to one or more specified email addresses.

Search

Command	Description
localop	Run subsequent commands, that is all commands following this, locally and not on a remote peer.
map	A looping operator, performs a search over each search result.
redistribute	Invokes parallel reduce search processing to shorten the search runtime of a set of supported SPL commands.
search	Searches indexes for matching events. This command is implicit at the start of every search pipeline that does not begin with another generating command.
sendemail	Emails search results, either inline or as an attachment, to one or more specified email addresses.

Subsearch

These are commands that you can use with **subsearches**.

Command	Description
append	Appends subsearch results to current results.
appendcols	Appends the fields of the subsearch results to current results, first results to first result, second to second, and so on.
appendpipe	Appends the result of the subpipeline applied to the current result set to results.
foreach	Runs a templated streaming subsearch for each field in a wildcarded field list.
format	Takes the results of a subsearch and formats them into a single result.
join	Combine the results of a subsearch with the results of a main search.
return	Specify the values to return from a subsearch.
set	Performs set operations (union, diff, intersect) on subsearches.

Time

Use these commands to search based on time ranges or add time information to your events.

Command	Description
<code>gentimes</code>	Returns results that match a time-range.
<code>localize</code>	Returns a list of the time ranges in which the search results were found.
<code>reltime</code>	Converts the difference between 'now' and ' <code>_time</code> ' to a human-readable value and adds this value to the field, ' <code>reltime</code> ', in your search results.

Command types

There are six broad types for all of the search commands: distributable streaming, centralized streaming, transforming, generating, orchestrating and dataset processing. These types are not mutually exclusive. A command might be streaming or transforming, and also generating.

The following tables list the commands that fit into each of these types. For detailed explanations about each of the types, see *Types of commands* in the *Search Manual*.

Streaming commands

A **streaming command** operates on each event as the event is returned by a search.

- A distributable streaming command runs on the indexer or the search head, depending on where in the search the command is invoked. Distributable streaming commands can be applied to subsets of indexed data in a parallel manner.
- A centralized streaming command applies a transformation to each event returned by a search. Unlike distributable streaming commands, a centralized streaming command only works on the search head.

Command	Notes
<code>addinfo</code>	Distributable streaming
<code>addtotals</code>	Distributable streaming. A <code>transforming</code> command when used to calculate column totals (not row totals).
<code>arules</code>	Some of the work is distributable streaming running on the indexer or the search head. The rest of the work is centralized streaming running on the search head.
<code>autoregress</code>	Centralized streaming.
<code>bin</code>	Streaming if specified with the <code>span</code> argument. Otherwise a dataset processing command.
<code>bucketdir</code>	Distributable streaming by default, but centralized streaming if the <code>local</code> setting specified for the command in the <code>commands.conf</code> file is set to true.
<code>cluster</code>	Streaming in some modes.
<code>convert</code>	Distributable streaming.
<code>dedup</code>	Distributable streaming in a prededup phase. Centralized streaming after the individual indexers perform their own dedup and the results are returned to the search head from each indexer. Using the <code>sortby</code> argument or specifying <code>keepevents=true</code> makes the <code>dedup</code> command a dataset processing command.
<code>eval</code>	Distributable streaming.
<code>extract</code>	Distributable streaming.

Command	Notes
fieldformat	Distributable streaming.
fields	Distributable streaming.
fillnull	Distributable streaming when a <code>field-list</code> is specified. A dataset processing command when no <code>field-list</code> is specified.
head	Centralized streaming.
highlight	Distributable streaming.
iconify	Distributable streaming.
iplocation	Distributable streaming.
join	Centralized streaming, if there is a defined set of fields to join to. A dataset processing command when no <code>field-list</code> is specified.
lookup	Distributable streaming when specified with <code>local=false</code> , which is the default. An orchestrating command when <code>local=true</code> .
makemv	Distributable streaming.
multikv	Distributable streaming.
mvexpand	Distributable streaming.
nomv	Distributable streaming.
rangemap	Distributable streaming.
regex	Distributable streaming.
reltime	Distributable streaming.
rename	Distributable streaming.
replace	Distributable streaming.
rex	Distributable streaming.
search	Distributable streaming if used further down the search pipeline. A generating command when it is the first command in the search.
spath	Distributable streaming.
strcat	Distributable streaming.
streamstats	Centralized streaming.
tags	Distributable streaming.
transaction	Centralized streaming.
typer	Distributable streaming.
where	Distributable streaming.
untable	Distributable streaming.
xmlkv	Distributable streaming.
xmlunescape	Distributable streaming by default, but centralized streaming if the <code>local</code> setting specified for the command in the <code>commands.conf</code> file is set to true.
xpath	Distributable streaming.
xyseries	Distributable streaming if the argument <code>grouped=false</code> is specified, which is the default. Otherwise a transforming command.

Generating commands

A **generating command** generates events or reports from one or more indexes without transforming the events.

Command	Notes
datamodel	Report-generating
dbinspect	Report-generating.
eventcount	Report-generating.
from	Can be either report-generating or event-generating depending on the search or knowledge object that is referenced by the command.
gentimes	Event-generating.
inputcsv	Event-generating (centralized).
Inputlookup	Event-generating (centralized) when <code>append=false</code> , which is the default.
loadjob	Event-generating (centralized).
makeresults	Report-generating.
metadata	Report-generating. Although metadata fetches data from all peers, any command run after it runs only on the search head.
metasearch	Event-generating.
mstats	Report-generating, except when <code>append=true</code> is specified.
multisearch	Event-generating.
pivot	Report-generating.
rest	
search	Event-generating (distributable) when the first command in the search, which is the default. A streaming (distributable) command if used later in the search pipeline.
searchtxn	Event-generating.
set	Event-generating.
tstats	Report-generating (distributable), except when <code>prestats=true</code> . When <code>prestats=true</code> , the <code>tstats</code> command is event-generating.

Transforming commands

A **transforming command** orders the results into a data table. The command "transforms" the specified cell values for each event into numerical values for statistical purposes.

In earlier versions of Splunk software, transforming commands were called reporting commands.

Command	Notes
addtotals	Transforming when used to calculate column totals (not row totals). A distributable streaming command when used to calculate row totals, which is the default.
anomalydetection	
append	

Command	Notes
associate	
chart	
cofilter	
contingency	
history	
makecontinuous	
mvcombine	
rare	
stats	
table	
timechart	
top	
xyseries	Transforming if grouped=true. A streaming (distributable) command when grouped=false, which is the default setting.

Orchestrating commands

Orchestrating commands control some aspect of how a search is processed. They do not directly affect the final result set of the search. For example, you might apply an orchestrating command to a search to enable or disable a search optimization that helps the overall search complete faster.

Command	Notes
localop	
lookup	Only becomes an orchestrating command when local=true. This forces the lookup command to run on the search head and not on any remote peers. A streaming (distributable) command when local=false, which is the default setting.
noop	
redistribute	
require	

Dataset processing commands

A dataset processing command is a command that requires the entire dataset before the command can run. Some of these commands fit into other command types in specific situations or when specific arguments are used.

Command	Notes
anomalousvalue	Some modes
anomalydetection	Some modes
append	Some modes
appendcols	
appendpipe	
bin	Some modes. A streaming command if the span argument is specified.

Command	Notes
cluster	Some modes
concurrency	
datamodel	
dedup	Using the <code>sortby</code> argument or specifying <code>keepevents=true</code> makes the <code>dedup</code> command a dataset processing command. Otherwise, <code>dedup</code> is a distributable streaming command in a prededup phase. Centralized streaming after the individual indexers perform their own dedup and the results are returned to the search head from each indexer.
eventstats	
fieldsummary	
fillnull	When no <code>field-list</code> is specified, a dataset processing command. If a <code>field-list</code> is specified <code>fillnull</code> is a distributable streaming command.
from	Some modes
join	Some modes. A centralized streaming command when there is a defined set of fields to join to.
map	
outlier	
reverse	
sort	
tail	
transaction	Some modes
union	Some modes

Splunk SPL for SQL users

This is not a perfect mapping between SQL and Splunk Search Processing Language (SPL), but if you are familiar with SQL, this quick comparison might be helpful as a jump-start into using the search commands.

Concepts

The Splunk platform does not store data in a conventional database. Rather, it stores data in a distributed, non-relational, semi-structured database with an **implicit time dimension**. Relational databases require that all table columns be defined up-front and they do not automatically scale by just plugging in new hardware. However, there are analogues to many of the concepts in the database world.

Database Concept	Splunk Concept	Notes
SQL query	Splunk search	A Splunk search retrieves indexed data and can perform transforming and reporting operations. Results from one search can be "piped", or transferred, from command to command, to filter, modify, reorder, and group your results.
table/view	search results	Search results can be thought of as a database view, a dynamically generated table of rows, with columns.

Database Concept	Splunk Concept	Notes
index	index	All values and fields are indexed by Splunk software, so there is no need to manually add, update, drop, or even think about indexing columns. Everything can be quickly retrieved automatically.
row	result/event	A result in a Splunk search is a list of fields (i.e., column) values, corresponding to a table row. An event is a result that has a timestamp and raw text. Typically an event is a record from a log file, such as: 173.26.34.223 -- [01/Jul/2009:12:05:27 -0700] "GET /trade/app?action=logout HTTP/1.1" 200 2953
column	field	Fields are returned dynamically from a search, meaning that one search might return a set of fields, while another search might return another set. After teaching Splunk software how to extract more fields from the raw underlying data, the same search will return more fields than it previously did. Fields are not tied to a datatype.
database/schema	index/app	A Splunk index is a collection of data, somewhat like a database has a collection of tables. Domain knowledge of that data, how to extract it, what reports to run, etc, are stored in a Splunk application.

From SQL to Splunk SPL

SQL is designed to search relational database tables which are comprised of **columns**. SPL is designed to search events, which are comprised of **fields**. In SQL, you often see examples that use "mytable" and "mycolumn". In SPL, you will see examples that refer to "fields". In these examples, the "source" field is used as a proxy for "table". In Splunk software, "source" is the name of the file, stream, or other input from which a particular piece of data originates, for example `/var/log/messages` or `UDP:514`.

When translating from any language to another, often the translation is longer because of idioms in the original language. Some of the Splunk search examples shown below could be more concise and more efficient, but for parallelism and clarity, the SPL table and field names are kept the same as the SQL example.

- SPL searches rarely need the FIELDS command to filter out columns because the user interface provides a more convenient method for filtering. The FIELDS command is used in the SPL examples for parallelism.
- With SPL, you never have to use the AND operator in Boolean searches, because AND is implied between terms. However when you use the AND or OR operators, they must be specified in uppercase.
- SPL commands do not need to be specified in uppercase. In these SPL examples, the commands are specified in uppercase for easier identification and clarity.
- Although some SPL commands loosely correspond to specific SQL commands as shown in the following table, your SPL searches might not produce the desired results if you "think in SQL." For this reason, avoid directly translating from SQL to SPL when you design your searches. See About the search language in the *Search Manual* for an overview of SPL.

SQL command	SQL example	Splunk SPL example
SELECT *	SELECT * FROM mytable	source=mytable
WHERE	SELECT * FROM mytable WHERE mycolumn=5	source=mytable mycolumn=5
SELECT	SELECT mycolumn1, mycolumn2	source=mytable

SQL command	SQL example	Splunk SPL example
	FROM mytable	FIELDS mycolumn1, mycolumn2
AND/OR	<pre>SELECT * FROM mytable WHERE (mycolumn1="true" OR mycolumn2="red") AND mycolumn3="blue"</pre>	<pre>source=mytable AND (mycolumn1="true" OR mycolumn2="red") AND mycolumn3="blue"</pre> <p>Note: The AND operator is implied in SPL and does not need to be specified. For this example you could also use:</p> <pre>source=mytable (mycolumn1="true" OR mycolumn2="red") mycolumn3="blue"</pre>
AS (alias)	SELECT mycolumn AS column_alias FROM mytable	<pre>source=mytable RENAME mycolumn as column_alias FIELDS column_alias</pre>
BETWEEN	SELECT * FROM mytable WHERE mycolumn BETWEEN 1 AND 5	<pre>source=mytable mycolumn>=1 mycolumn<=5</pre>
GROUP BY	<pre>SELECT mycolumn, avg(mycolumn) FROM mytable WHERE mycolumn=value GROUP BY mycolumn</pre>	<pre>source=mytable mycolumn=value STATS avg(mycolumn) BY mycolumn FIELDS mycolumn, avg(mycolumn)</pre> <p>Several commands use a by-clause to group information, including chart, rare, sort, stats, and timechart.</p>
HAVING	<pre>SELECT mycolumn, avg(mycolumn) FROM mytable WHERE mycolumn=value GROUP BY mycolumn HAVING avg(mycolumn)=value</pre>	<pre>source=mytable mycolumn=value STATS avg(mycolumn) BY mycolumn SEARCH avg(mycolumn)=value FIELDS mycolumn, avg(mycolumn)</pre>
LIKE	<pre>SELECT * FROM mytable WHERE mycolumn LIKE "%some text%"</pre>	<pre>source=mytable mycolumn="*some text*"</pre> <p>Note: The most common search in Splunk SPL is nearly impossible in SQL - to search all fields for a substring. The following SPL search returns all rows that contain "some text" anywhere:</p> <pre>source=mytable "some text"</pre>
ORDER BY	<pre>SELECT * FROM mytable ORDER BY mycolumn desc</pre>	<pre>source=mytable SORT -mycolumn</pre>

SQL command	SQL example	Splunk SPL example
		In SPL you use a negative sign (-) in front of a field name to sort in descending order.
SELECT DISTINCT	SELECT DISTINCT mycolumn1, mycolumn2 FROM mytable	source=mytable DEDUP mycolumn1, mycolumn2 FIELDS mycolumn1, mycolumn2
SELECT TOP	SELECT TOP (5) mycolumn1, mycolumn2 FROM mytable1 WHERE mycolumn3 = "bar" ORDER BY mycolumn1 mycolumn2	Source=mytable1 mycolumn3="bar" FIELDS mycolumn1 mycolumn2 SORT mycolumn1 mycolumn2 HEAD 5
INNER JOIN	SELECT * FROM mytable1 INNER JOIN mytable2 ON mytable1.mycolumn= mytable2.mycolumn	<p>index=myIndex1 OR index=myIndex2 stats values(*) AS * BY myField</p> <p>Note: There are two other methods to join tables:</p> <ul style="list-style-type: none"> • Use the <code>lookup</code> command to add fields from an external table: <pre>... LOOKUP myvalue lookup mycolumn OUTPUT myoutputcolumn</pre> <ul style="list-style-type: none"> • Use a subsearch: <pre>source=mytable1 [SEARCH source=mytable2 mycolumn2=myvalue FIELDS mycolumn2]</pre> <p>If the columns that you want to join on have different names, use the <code>rename</code> command to rename one of the columns. For example, to rename the column in mytable2:</p> <pre>source=mytable1 JOIN type=inner mycolumn [SEARCH source=mytable2 RENAME mycolumn2 AS mycolumn]</pre> <p>To rename the column in myindex1:</p> <pre>index=myIndex1 OR index=myIndex2 rename myfield1 as myField stats values(*) AS * BY myField</pre> <p>You can rename a column regardless of whether you use the search command, a lookup, or a subsearch.</p>

SQL command	SQL example	Splunk SPL example
LEFT (OUTER) JOIN	<pre>SELECT * FROM mytable1 LEFT JOIN mytable2 ON mytable1.mycolumn= mytable2.mycolumn</pre>	<pre>source=mytable1 JOIN type=left mycolumn [SEARCH source=mytable2]</pre>
SELECT INTO	<pre>SELECT * INTO new_mytable IN mydb2 FROM old_mytable</pre>	<pre>source=old_mytable EVAL source=new_mytable COLLECT index=mydb2</pre> <p>Note: COLLECT is typically used to store expensively calculated fields back into your Splunk deployment so that future access is much faster. This current example is atypical but shown for comparison to the SQL command. The source will be renamed orig_source</p>
TRUNCATE TABLE	<pre>TRUNCATE TABLE mytable</pre>	<pre>source=mytable DELETE</pre>
INSERT INTO	<pre>INSERT INTO mytable VALUES (value1, value2, value3,....)</pre>	<p>Note: see SELECT INTO. Individual records are not added via the search language, but can be added via the API if need be.</p>
UNION	<pre>SELECT mycolumn FROM mytable1 UNION SELECT mycolumn FROM mytable2</pre>	<pre>source=mytable1 APPEND [SEARCH source=mytable2] DEDUP mycolumn</pre>
UNION ALL	<pre>SELECT * FROM mytable1 UNION ALL SELECT * FROM mytable2</pre>	<pre>source=mytable1 APPEND [SEARCH source=mytable2]</pre>
DELETE	<pre>DELETE FROM mytable WHERE mycolumn=5</pre>	<pre>source=mytable1 mycolumn=5 DELETE</pre>
UPDATE	<pre>UPDATE mytable SET column1=value, column2=value,... WHERE some_column=some_value</pre>	<p>Note: There are a few things to think about when updating records in Splunk Enterprise. First, you can just add the new values to your Splunk deployment (see INSERT INTO) and not worry about deleting the old values, because Splunk software always returns the most recent results first. Second, on retrieval, you can always de-duplicate the results to ensure only the latest values are used (see SELECT DISTINCT). Finally, you can actually delete the old records (see DELETE).</p>

See also

- [Understanding SPL syntax](#)

SPL data types and clauses

Data types

bool

The <bool> argument value represents the Boolean data type. The documentation specifies 'true' or 'false'. Other variations of Boolean values are accepted in commands. For example, for 'true' you can also use 't', 'T', 'TRUE', or the number one '1'. For 'false', you can use 'f', 'F', 'FALSE', or the number zero '0'.

int

The <int> argument value represents the integer data type.

num

The <num> argument value represents the number data type.

float

The <float> argument value represents the float data type.

Common syntax clauses

bin-span

Syntax: span=<span-length> | <log-span>

Description: Sets the size of each bin.

Example: span=2d

Example: span=5m

Example: span=10

by-clause

Syntax: by <field-list>

Description: Fields to group by.

Example: BY addr, port

Example: BY host

eval-function

Syntax: abs | case | cidrmatch | coalesce | exact | exp | floor | if | ifnull | isbool | isint | isnonnull | isnull | isnum | isstr | len|like | In|log | lower | match | max | md5 | min | mvcount | mvindex | mvfilter | now | null | nullif | pi | pow | random | replace | round | searchmatch | sqrt | substr | tostring | trim | ltrim | rtrim | typeof | upper | urldecode | validate

Description: Function used by eval.

Example: md5(field)

Example: typeof(12) + typeof("string") + typeof(1==2) + typeof(badfield)

Example: searchmatch("foo AND bar")

Example: sqrt(9)
Example: round(3.5)
Example: replace(date, "^(\d{1,2})/(\d{1,2})/", "\2\1/")
Example: pi()
Example: nullif(fielda, fieldb)
Example: random()
Example: pow(x, y)
Example: mvfilter(match(email, "\.net\$") OR match(email, "\.org\$"))
Example: mvindex(multifield, 2)
Example: null()
Example: now()
Example: isbool(field)
Example: exp(3)
Example: floor(1.9)
Example: coalesce(null(), "Returned value", null())
Example: exact(3.14 * num)
Example: case(error == 404, "Not found", error == 500, "Internal Server Error", error == 200, "OK")
Example: cidrmatch("123.132.32.0/25", ip)
Example: abs(number)
Example: isnotnull(field)
Example: substr("string", 1, 3) + substr("string", -3)
Example: if(error == 200, "OK", "Error")
Example: len(field)
Example: log(number, 2)
Example: lower(username)
Example: match(field, "^\\d{1,3}\\.\d{1,3}\\.\d{1,3}\\.\d{1,3}\$")
Example: max(1, 3, 6, 7, "f" ^ "\\d{1,3}\\.\d{1,3}\\.\d{1,3}\\.\d{1,3}\$")oo", field)
Example: like(field, "foo%")
Example: ln(bytes)
Example: mvcount(multifield)
Example: urldecode("http%3A%2F%2Fwww.splunk.com%2Fdownload%3Fr%3Dheader")
Example: validate(isint(port), "ERROR: Port is not an integer", port >= 1 AND port <= 65535, "ERROR: Port is out of range")
Example: tostring(1==1) + " " + tostring(15, "hex") + " " + tostring(12345.6789, "commas")
Example: trim(" ZZZZabcZZ ", " Z")

evaluated-field

Syntax: eval(<eval-expression>)
Description: A dynamically evaluated field

field

field-list

regex-expression

Syntax: ("")?<string>("")?
Description: A Perl Compatible Regular Expression supported by the PCRE library.
Example: ... | regex _raw="(?<\d)10.\d{1,3}.\d{1,3}.\d{1,3}(?!d)"

single-agg

Syntax: count | stats-func (<field>)

Description: A single aggregation applied to a single field (can be evalued field). No wildcards are allowed. The field must be specified, except when using the special 'count' aggregator that applies to events as a whole.

Example: avg(delay)

Example: sum({date_hour * date_minute})

Example: count

sort-by-clause

Syntax: ("-"|"+"<sort-field> ", "

Description: List of fields to sort by and their sort order (ascending or descending)

Example: - time, host

Example: -size, +source

Example: _time, -host

span-length

Syntax: <int:span>(<timescale>)?

Description: Span of each bin. If using a timescale, this is used as a time range. If not, this is an absolute bucket "length."

Example: 2d

Example: 5m

Example: 10

split-by-clause

Syntax: <field> (<tc-option>)* (<where-clause>)?

Description: Specifies a field to split by. If field is numerical, default discretization is applied.

stats-agg

Syntax: <stats-func>(& " (<evalued-field> | <wc-field>)? ")")?

Description: A specifier formed by a aggregation function applied to a field or set of fields. As of 4.0, it can also be an aggregation function applied to a arbitrary eval expression. The eval expression must be wrapped by "{" and "}". If no field is specified in the parenthesis, the aggregation is applied independently to all fields, and is equivalent to calling a field value of * When a numeric aggregator is applied to a not-completely-numeric field no column is generated for that aggregation.

Example: count({sourcetype="splunkd"})

Example: max(size)

Example: stdev(*delay)

Example: avg(kbps)

stats-agg-term

Syntax: <stats-agg> (as <wc-field>)?

Description: A statistical specifier optionally renamed to a new field name.

Example: count(device) AS numdevices

Example: avg(kbps)

subsearch

Syntax: [<string>]

Description: Specifies a subsearch.

Example: [search 404 | select url]

tc-option

Syntax: <bins-options> | (usenull=<bool>) | (useother=<bool>) | (nullstr=<string>) |(otherstr=<string>)

Description: Options for controlling the behavior of splitting by a field. In addition to the bins-options: usenull controls whether or not a series is created for events that do not contain the split-by field. This series is labeled by the value of the nullstr option, and defaults to NULL. useother specifies if a series should be added for data series not included in the graph because they did not meet the criteria of the <where-clause>. This series is labeled by the value of the otherstr option, and defaults to OTHER.

Example: otherstr=OTHERFIELDS

Example: usenull=f

Example: bins=10

timeformat

Syntax: timeformat=<string>

Description: Set the time format for starttime and endtime terms.

Example: timeformat=%m/%d/%Y:%H:%M:%S

timestamp

Syntax: (MM/DD/YY)?:(HH:MM:SS)?|<int>

Description: None

Example: 10/1/07:12:34:56

Example: -5

where-clause

Syntax: where <single-agg> <where-comp>

Description: Specifies the criteria for including particular data series when a field is given in the tc-by-clause. This optional clause, if omitted, default to "where sum in top10". The aggregation term is applied to each data series and the result of these aggregations is compared to the criteria. The most common use of this option is to select for spikes rather than overall mass of distribution in series selection. The default value finds the top ten series by area under the curve. Alternately one could replace sum with max to find the series with the ten highest spikes.

Example: where max < 10

Example: where count notin bottom10

Example: where avg > 100

Example: where sum in top5

wc-field

Evaluation Functions

Evaluation functions

Use the evaluation functions to evaluate an expression, based on your events, and return a result.

Quick reference

See the [Supported functions and syntax](#) section for a quick reference list of the evaluation functions.

Commands

You can use evaluation functions with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions with other commands.

Usage

- All functions that accept strings can accept literal strings or any field.
- All functions that accept numbers can accept literal numbers or any numeric field.

String arguments and fields

For most evaluation functions, when a string argument is expected, you can specify either a literal string or a field name. Literal strings must be enclosed in double quotation marks. In other words, when the function syntax specifies a string you can specify any expression that results in a string. For example, you have a field called `name` that contains the names of your servers. If you want to append the literal string `server` at the end of the name, you would use dot notation like this in your search: `name."server"`.

Nested functions

You can specify a function as an argument to another function.

In the following example, the `cidrmatch` function is used as the first argument in the `if` function.

```
... | eval isLocal=if(cidrmatch("123.132.32.0/25",ip), "local", "not local")
```

The following example shows how to use the `true()` function to provide a default to the `case` function.

```
... | eval error=case(status == 200, "OK", status == 404, "Not found", true(), "Other")
```

Supported functions and syntax

There are two ways that you can see information about the supported evaluation functions:

- [Function list by category](#)
- [Alphabetical list of functions](#)

Function list by category

The following table is a quick reference of the supported evaluation functions, organized by category. This table provides a brief description for each function. Use the links in the table to learn more about each function and to see examples.

Type of function	Supported functions and syntax	Description
Comparison and Conditional functions	<code>case(<condition>,<value>,...)</code>	Accepts alternating conditions and values. Returns the first value for which the condition evaluates to TRUE.
	<code>cidrmatch(<cidr>,<ip>)</code>	Returns TRUE when an IP address, <ip>, belongs to a particular CIDR subnet, <cidr>.
	<code>coalesce(<values>)</code>	Takes one or more values and returns the first value that is not NULL.
	<code>false()</code>	Returns FALSE.
	<code>if(<predicate>,<true_value>,<false_value>)</code>	If the <predicate> expression evaluates to TRUE, returns the <true_value>, otherwise the function returns the <false_value>.
	<code>in(<field>,<list>)</code>	Returns TRUE if one of the values in the list matches a value that you specify.
	<code>like(<str>,<pattern>)</code>	Returns TRUE only if <str> matches <pattern>.
	<code>lookup(<lookup_table>,<json_object>,<json_array>)</code>	Performs a CSV lookup. Returns the output field or fields in the form of a JSON object. Note: The <code>lookup()</code> function is available only to Splunk Enterprise users.
	<code>match(<str>,<regex>)</code>	Returns TRUE if the regular expression <regex> finds a match against any substring of the string value <str>. Otherwise returns FALSE.
	<code>null()</code>	This function takes no arguments and returns NULL.
	<code>nullif(<field1>,<field2>)</code>	Compares the values in two fields and returns NULL if the value in <field1> is equal to the value in <field2>. Otherwise returns the value in <field1>.
	<code>searchmatch(<search_str>)</code>	Returns TRUE if the event matches the search string.
	<code>true()</code>	Returns TRUE.
	<code>validate(<condition>,<value>,...)</code>	Takes a list of conditions and values and returns the value that corresponds to the condition that evaluates to FALSE. This function defaults to NULL if all conditions evaluate to TRUE. This function is the opposite of the <code>case</code> function.
Conversion functions	<code>ipmask(<mask>,<ip>)</code>	Generates a new masked IP address by applying a mask to an IP address using a bitwise AND operation.
	<code>printf(<format>,<arguments>)</code>	Creates a formatted string based on a format description that you provide.
	<code>tonumber(<str>,<base>)</code>	Converts a string to a number.
	<code>tostring(<value>,<format>)</code>	Converts the input, such as a number or a Boolean value, to a string.
Cryptographic functions	<code>md5(<str>)</code>	Computes the md5 hash for the string value.
	<code>sha1(<str>)</code>	Computes the sha1 hash for the string value.
	<code>sha256(<str>)</code>	Computes the sha256 hash for the string value.
	<code>sha512(<str>)</code>	Computes the sha512 hash for the string value.

Type of function	Supported functions and syntax	Description
Date and Time functions	now()	Returns the time that the search was started.
	relative_time(<time>,<specifier>)	Adjusts the time by a relative time specifier.
	strftime(<time>,<format>)	Takes a UNIX time and renders it into a human readable format.
	strptime(<str>,<format>)	Takes a human readable time and renders it into UNIX time.
Informational functions	time()	The time that eval function was computed. The time will be different for each event, based on when the event was processed.
	isbool(<value>)	Returns TRUE if the field value is Boolean.
	isint(<value>)	Returns TRUE if the field value is an integer.
	isnotnull(<value>)	Returns TRUE if the field value is not NULL.
	isnull(<value>)	Returns TRUE if the field value is NULL.
	isnum(<value>)	Returns TRUE if the field value is a number.
	isstr(<value>)	Returns TRUE if the field value is a string.
JSON functions	typeof(<value>)	Returns a string that indicates the field type, such as Number, String, Boolean, and so forth
	json_object(<members>)	Creates a new JSON object from members of key-value pairs.
	json_append(<json>,<path_value_pairs>)	Appends values to the ends of indicated arrays within a JSON document.
	json_array(<values>)	Creates a JSON array using a list of values.
	json_array_to_mv(<json_array>,<boolean>)	Maps the elements of a proper JSON array into a multivalue field.
	json_extend(<json>,<path_value_pairs>)	Flattens arrays into their component values and appends those values to the ends of indicated arrays within a valid JSON document.
	json_extract(<json>,<paths>)	This function returns a value from a piece JSON and zero or more paths. The value is returned in either a JSON array, or a Splunk software native type value.
	json_extract_exact(<json>,<keys>)	Returns Splunk software native type values from a piece of JSON by matching literal strings in the event and extracting them as keys.
	json_keys(<json>)	Returns the keys from the key-value pairs in a JSON object as a JSON array.
	json_set(<json>,<path_value_pairs>)	Inserts or overwrites values for a JSON node with the values provided and returns an updated JSON object.
	json_set_exact(<json>,<key_value_pairs>)	Uses provided key-value pairs to generate or overwrite a JSON object.
Mathematical functions	json_valid(<json>)	Evaluates whether piece of JSON uses valid JSON syntax and returns either TRUE or FALSE.
	abs(<num>)	Returns the absolute value.
	ceiling(<num>)	Rounds the value up to the next highest integer.
	exact(<expression>)	Returns the result of a numeric eval calculation with a larger amount of precision in the formatted output.
	exp(<num>)	Returns the exponential function e^N .
	floor(<num>)	Rounds the value down to the next lowest integer.

	Supported functions and syntax	Description
Type of function	ln(<num>)	Returns the natural logarithm.
	log(<num>,<base>)	Returns the logarithm of <num> using <base> as the base. If <base> is omitted, base 10 is used.
	pi()	Returns the constant pi to 11 digits of precision.
	pow(<num>,<exp>)	Returns <num> to the power of <exp>, <num> <exp>.
	round(<num>,<precision>)	Returns <num> rounded to the amount of decimal places specified by <precision>. The default is to round to an integer.
	sigfig(<num>)	Rounds <num> to the appropriate number of significant figures.
	sqrt(<num>)	Returns the square root of the value.
	sum(<num>,...)	Returns the sum of numerical values as an integer.
Multivalue eval functions	commands(<value>)	Returns a multivalued field that contains a list of the commands used in <value>.
	mvappend(<values>)	Returns a multivalue result based on all of values specified.
	mvcount(<mv>)	Returns the count of the number of values in the specified field.
	mvdedup(<mv>)	Removes all of the duplicate values from a multivalue field.
	mvfilter(<predicate>)	Filters a multivalue field based on an arbitrary Boolean expression.
	mvfind(<mv>,<regex>)	Finds the index of a value in a multivalue field that matches the regular expression.
	mvindex(<mv>,<start>,<end>)	Returns a subset of the multivalue field using the start and end index values.
	mvjoin(<mv>,<delim>)	Takes all of the values in a multivalue field and appends the values together using a delimiter.
	mvmap(<mv>,<expression>)	This function iterates over the values of a multivalue field, performs an operation using the <expression> on each value, and returns a multivalue field with the list of results.
	mvrange(<start>,<end>,<step>)	Creates a multivalue field based on a range of specified numbers.
Statistical eval functions	mvsort(<mv>)	Returns the values of a multivalue field sorted lexicographically.
	mvzip(<mv_left>,<mv_right>,<delim>)	Combines the values in two multivalue fields. The delimiter is used to specify a delimiting character to join the two values.
	mv_to_json_array(<field>,<inver_types>)	Maps the elements of a multivalue field to a JSON array.
	split(<str>,<delim>)	Splits the string values on the delimiter and returns the string values as a multivalue field.
	avg(<values>)	Returns the average of numerical values as an integer.
Text functions	max(<values>)	Returns the maximum of a set of string or numeric values.
	min(<values>)	Returns the minimum of a set of string or numeric values.
	random()	Returns a pseudo-random integer ranging from zero to $2^{31}-1$.
	len(<str>)	Returns the count of the number of characters, not bytes, in the string.
	lower(<str>)	Converts the string to lowercase.

	Supported functions and syntax	Description
Type of function	<code>ltrim(<str>,<trim_chars>)</code>	Removes characters from the left side of a string.
	<code>replace(<str>,<regex>,<replacement>)</code>	Substitutes the replacement string for every occurrence of the regular expression in the string.
	<code>rtrim(<str>,<trim_chars>)</code>	Removes the trim characters from the right side of the string.
	<code>spath(<value>,<path>)</code>	Extracts information from the structured data formats XML and JSON.
	<code>substr(<str>,<start>,<length>)</code>	Returns a substring of a string, beginning at the start index. The length of the substring specifies the number of character to return.
	<code>trim(<str>,<trim_chars>)</code>	Trim characters from both sides of a string.
	<code>upper(<str>)</code>	Returns the string in uppercase.
	<code>urldecode(<url>)</code>	Replaces URL escaped characters with the original characters.
Trigonometry and Hyperbolic functions	<code>acos(X)</code>	Computes the arc cosine of X.
	<code>acosh(X)</code>	Computes the arc hyperbolic cosine of X.
	<code>asin(X)</code>	Computes the arc sine of X.
	<code>asinh(X)</code>	Computes the arc hyperbolic sine of X.
	<code>atan(X)</code>	Computes the arc tangent of X.
	<code>atan2(X,Y)</code>	Computes the arc tangent of X,Y.
	<code>atanh(X)</code>	Computes the arc hyperbolic tangent of X.
	<code>cos(X)</code>	Computes the cosine of an angle of X radians.
	<code>cosh(X)</code>	Computes the hyperbolic cosine of X radians.
	<code>hypot(X,Y)</code>	Computes the hypotenuse of a triangle.
	<code>sin(X)</code>	Computes the sine of X.
	<code>sinh(X)</code>	Computes the hyperbolic sine of X.
	<code>tan(X)</code>	Computes the tangent of X.
	<code>tanh(X)</code>	Computes the hyperbolic tangent of X.

Alphabetical list of functions

The following table is a quick reference of the supported evaluation functions, organized alphabetically. This table provides a brief description for each function. Use the links in the table to learn more about each function and to see examples.

Supported functions and syntax	Description	Type of function
<code>abs(<num>)</code>	Returns the absolute value.	Mathematical functions
<code>acos(X)</code>	Computes the arc cosine of X.	Trigonometry and Hyperbolic functions
<code>acosh(X)</code>	Computes the arc hyperbolic cosine of X.	Trigonometry and Hyperbolic

Supported functions and syntax	Description	Type of function
		functions
asin(X)	Computes the arc sine of X.	Trigonometry and Hyperbolic functions
asinh(X)	Computes the arc hyperbolic sine of X.	Trigonometry and Hyperbolic functions
atan(X)	Computes the arc tangent of X.	Trigonometry and Hyperbolic functions
atan2(X,Y)	Computes the arc tangent of X,Y.	Trigonometry and Hyperbolic functions
atanh(X)	Computes the arc hyperbolic tangent of X.	Trigonometry and Hyperbolic functions
avg(<values>)	Returns the average of numerical values as an integer.	Statistical eval functions
case(<condition>,<value>,...)	Accepts alternating conditions and values. Returns the first value for which the condition evaluates to TRUE.	Comparison and Conditional functions
cidrmatch(<cidr>,<ip>)	Returns TRUE when an IP address, <ip>, belongs to a particular CIDR subnet, <cidr>.	Comparison and Conditional functions
ceiling(<num>)	Rounds the value up to the next highest integer.	Mathematical functions
coalesce(<values>)	Takes one or more values and returns the first value that is not NULL.	Comparison and Conditional functions
commands(<value>)	Returns a multivalued field that contains a list of the commands used in <value>.	Multivalue eval functions
cos(X)	Computes the cosine of an angle of X radians.	Trigonometry and Hyperbolic functions
cosh(X)	Computes the hyperbolic cosine of X radians.	Trigonometry and Hyperbolic functions
exact(<expression>)	Returns the result of a numeric eval calculation with a larger amount of precision in the formatted output.	Mathematical functions
exp(<num>)	Returns the exponential function e^N .	Mathematical functions
false()	Returns FALSE.	Comparison and Conditional functions
floor(<num>)	Rounds the value down to the next lowest integer.	Mathematical functions

Supported functions and syntax	Description	Type of function
hypot(X,Y)	Computes the hypotenuse of a triangle.	Trigonometry and Hyperbolic functions
if(<predicate>,<true_value>,<false_value>)	If the <predicate> expression evaluates to TRUE, returns the <true_value>, otherwise the function returns the <false_value>.	Comparison and Conditional functions
in(<field>,<list>)	Returns TRUE if one of the values in the list matches a value that you specify.	Comparison and Conditional functions
ipmask(<mask>,<ip>)	The function generates a new masked IP address by applying a mask to an IP address using a bitwise AND operation.	Conversion functions
isbool(<value>)	Returns TRUE if the field value is Boolean.	Informational functions
isint(<value>)	Returns TRUE if the field value is an integer.	Informational functions
isnotnull(<value>)	Returns TRUE if the field value is not NULL.	Informational functions
isNull(<value>)	Returns TRUE if the field value is NULL.	Informational functions
isnum(<value>)	Returns TRUE if the field value is a number.	Informational functions
isstr(<value>)	Returns TRUE if the field value is a string.	Informational functions
json_append(<json>,<path_value_pairs>)	Appends values to the ends of indicated arrays within a JSON document.	JSON functions
json_array(<values>)	Creates a JSON array using a list of values.	JSON functions
json_array_to_mv(<json_array>,<boolean>)	Maps the elements of a proper JSON array into a multivalue field.	JSON functions
json_extend(<json>,<path_value_pairs>)	Flattens arrays into their component values and appends those values to the ends of indicated arrays within a valid JSON document.	JSON functions
json_extract(<json>,<paths>)	Returns a value from a piece JSON and zero or more paths. The value is returned in either a JSON array, or a Splunk software native type value.	JSON functions
json_extract_exact(<json>,<keys>)	Returns Splunk software native type values from a piece of JSON by matching literal strings in the event and extracting them as keys.	JSON functions
json_keys(<json>)	Returns the keys from the key-value pairs in a JSON object. The keys are returned as a JSON array.	JSON functions
json_object(<members>)	Creates a new JSON object from members of key-value pairs.	JSON functions
json_set(<json>,<path_value_pairs>)	Inserts or overwrites values for a JSON node with the values provided and returns an updated JSON object.	JSON functions
json_set_exact(<json>,<key_value_pairs>)	Uses provided key-value pairs to generate or overwrite a JSON object.	JSON functions
json_valid(<json>)	Evaluates whether piece of JSON uses valid JSON syntax and returns either TRUE or FALSE.	JSON functions
len(X)	Returns the count of the number of characters (not bytes) in the string.	Text functions

Supported functions and syntax	Description	Type of function
like(<str>,<pattern>))	Returns TRUE only if <str> matches <pattern>.	Comparison and Conditional functions
ln(<num>)	Returns the natural logarithm.	Mathematical functions
log(<num>,<base>)	Returns the logarithm of <num> using <base> as the base. If <base> is omitted, base 10 is used.	Mathematical functions
lookup(<lookup_table>, <json_object>, <json_array>)	Performs a CSV lookup. Returns the output field or fields in the form of a JSON object. Note: The lookup() function is available only to Splunk Enterprise users.	Comparison and Conditional functions
len(<str>)	Returns the count of the number of characters, not bytes, in the string.	Text functions
lower(<str>)	Converts the string to lowercase.	Text functions
ltrim(<str>,<trim_chars>)	Removes characters from the left side of a string.	Text functions
match(<str>, <regex>)	Returns TRUE if the regular expression <regex> finds a match against any substring of the string value <str>. Otherwise returns FALSE.	Comparison and Conditional functions
max(<values>)	Returns the maximum of a set of string or numeric values.	Statistical eval functions
md5(<str>)	Computes the md5 hash for the string value.	Cryptographic functions
min(<values>)	Returns the minimum of a set of string or numeric values.	Statistical eval functions
mvappend(<values>)	Returns a multivalue result based on all of values specified.	Multivalue eval functions
mvcount(<mv>)	Returns the count of the number of values in the specified field.	Multivalue eval functions
mvdedup(<mv>)	Removes all of the duplicate values from a multivalue field.	Multivalue eval functions
mvfilter(<predicate>)	Filters a multivalue field based on an arbitrary Boolean expression.	Multivalue eval functions
mvfind(<mv>,<regex>)	Finds the index of a value in a multivalue field that matches the regular expression.	Multivalue eval functions
mvindex(<mv>,<start>,<end>)	Returns a subset of the multivalue field using the start and end index values.	Multivalue eval functions
mvjoin(<mv>,<delim>)	Takes all of the values in a multivalue field and appends the values together using a delimiter.	Multivalue eval functions
mvmap(<mv>,<expression>)	This function iterates over the values of a multivalue field, performs an operation using the <expression> on each value, and returns a multivalue field with the list of results.	Multivalue eval functions
mvrangle(<start>,<end>,<step>)	Creates a multivalue field based on a range of specified numbers.	Multivalue eval functions
mvsort(<mv>)	Returns the values of a multivalue field sorted lexicographically.	Multivalue eval functions

Supported functions and syntax	Description	Type of function
<code>mvzip(<mv_left>,<mv_right>,<delim>)</code>	Combines the values in two multivalue fields. The delimiter is used to specify a delimiting character to join the two values.	Multivalue eval functions
<code>mv_to_json_array(<field>,<infer_types>)</code>	Maps the elements of a multivalue field to a JSON array.	JSON functions
<code>now()</code>	Returns the time that the search was started.	Date and Time functions
<code>null()</code>	This function takes no arguments and returns NULL.	Comparison and Conditional functions
<code>nullif(<field1>,<field2>)</code>	Compares the values in two fields and returns NULL if the value in <field1> is equal to the value in <field2>. Otherwise returns the value in <field1>.	Comparison and Conditional functions
<code>pi()</code>	Returns the constant pi to 11 digits of precision.	Mathematical functions
<code>pow(<num>,<exp>)</code>	Returns <num> to the power of <exp>, $<\text{num}>^{<\text{exp}>}$.	Mathematical functions
<code>printf(<format>,<arguments>)</code>	Creates a formatted string based on a format description that you provide.	Conversion functions
<code>random()</code>	Returns a pseudo-random integer ranging from zero to $2^{31}-1$.	Statistical eval functions
<code>relative_time(<time>,<specifier>)</code>	Adjusts the time by a relative time specifier.	Date and Time functions
<code>replace(<str>,<regex>,<replacement>)</code>	Substitutes the replacement string for every occurrence of the regular expression in the string.	Text functions
<code>round(<num>,<precision>)</code>	Returns <num> rounded to the amount of decimal places specified by <precision>. The default is to round to an integer.	Mathematical functions
<code>rtrim(<str>,<trim_chars>)</code>	Removes the trim characters from the right side of the string.	Text functions
<code>searchmatch(<search_str>)</code>	Returns TRUE if the event matches the search string.	Comparison and Conditional functions
<code>sha1(<str>)</code>	Computes the sha1 hash for the string value.	Cryptographic functions
<code>sha256(<str>)</code>	Computes the sha256 hash for the string value.	Cryptographic functions
<code>sha512(<str>)</code>	Computes the sha512 hash for the string value.	Cryptographic functions
<code>sigfig(<num>)</code>	Rounds <num> to the appropriate number of significant figures.	Mathematical functions
<code>sin(X)</code>	Computes the sine of X.	Trigonometry and Hyperbolic functions
<code>sinh(X)</code>	Computes the hyperbolic sine of X.	Trigonometry and Hyperbolic functions

Supported functions and syntax	Description	Type of function
<code>spath(<value>,<path>)</code>	Extracts information from the structured data formats XML and JSON.	Text functions
<code>split(<str>,<delim>)</code>	Splits the string values on the delimiter and returns the string values as a multivalue field.	Multivalue eval functions
<code>sqrt(<num>)</code>	Returns the square root of the value.	Mathematical functions
<code>strftime(<time>,<format>)</code>	Takes a UNIX time and renders it into a human readable format.	Date and Time functions
<code>strptime(<str>,<format>)</code>	Takes a human readable time and renders it into UNIX time.	Date and Time functions
<code>substr(<str>,<start>,<length>)</code>	Returns a substring of a string, beginning at the start index. The length of the substring specifies the number of character to return.	Text functions
<code>sum(<num>,...)</code>	Returns the sum of numerical values as an integer.	Mathematical functions
<code>tan(X)</code>	Computes the tangent of X.	Trigonometry and Hyperbolic functions
<code>tanh(X)</code>	Computes the hyperbolic tangent of X.	Trigonometry and Hyperbolic functions
<code>time()</code>	The time that eval function was computed. The time will be different for each event, based on when the event was processed.	Date and Time functions
<code>tonumber(<str>,<base>)</code>	Converts a string to a number.	Conversion functions
<code>toString(<value>,<format>)</code>	Converts the input, such as a number or a Boolean value, to a string.	Conversion functions
<code>trim(<str>,<trim_chars>)</code>	Trim characters from both sides of a string.	Text functions
<code>true()</code>	Returns TRUE.	Comparison and Conditional functions
<code>typeof(<value>)</code>	Returns a string that indicates the field type, such as Number, String, Boolean, and so forth.	Informational functions
<code>upper(<str>)</code>	Returns the string in uppercase.	Text functions
<code>urldecode(<url>)</code>	Replaces URL escaped characters with the original characters.	Text functions
<code>validate(<condition>,<value>,...)</code>	Takes a list of conditions and values and returns the value that corresponds to the condition that evaluates to FALSE. This function defaults to NULL if all conditions evaluate to TRUE. This function is the opposite of the <code>case</code> function.	Comparison and Conditional functions

See also

Topics:

[Statistical and charting functions](#)

Commands:

[eval](#)
[fieldformat](#)
[where](#)

Comparison and Conditional functions

The following list contains the functions that you can use to compare values or specify conditional statements.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

For information about Boolean operators, such as AND and OR, see [Boolean operators](#).

case(<condition>,<value>,...)

Description

Accepts alternating conditions and values. Returns the first value for which the condition evaluates to TRUE.

The <condition> arguments are Boolean expressions that are evaluated from first to last. When the first <condition> expression is encountered that evaluates to TRUE, the corresponding <value> argument is returned. The function defaults to NULL if none of the <condition> arguments are true.

Usage

You can use this function with the [eval](#), [fieldformat](#), and [where](#) commands, and as part of eval expressions.

Basic example

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.

The following example returns descriptions for the corresponding http status code.

```
sourcetype=access_* | eval description=case(status==200, "OK", status==404, "Not found", status==500, "Internal Server Error") | table status description
```

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

status	description
200	OK
200	OK
408	
200	OK
404	Not found
200	OK

status	description
406	
500	Internal Server Error
200	OK

For an example of how to display a default value when that status does not match one of the values specified, see the [True function](#).

Extended example

This example shows you how to use the `case` function in two different ways, to create categories and to create a custom sort order.

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.

You want classify earthquakes based on depth. Shallow-focus earthquakes occur at depths less than 70 km. Mid-focus earthquakes occur at depths between 70 and 300 km. Deep-focus earthquakes occur at depths greater than 300 km. We'll use Low, Mid, and Deep for the category names.

```
source=all_month.csv | eval Description=case(depth<=70, "Low", 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 "Low", "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 `Low`.

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:

Description	count	min(Mag)	max(Mag)
Deep	35	4.1	6.7
Low	6236	-0.60	7.70
Mid	635	0.8	6.3

You can sort the results in the `Description` column by clicking the sort icon in Splunk Web. However in this example the order would be alphabetical returning results in Deep, Low, Mid or Mid, Low, Deep order.

You can also use the `case` function to sort the results in a custom order, such as Low, Mid, Deep. You create the custom sort order by giving the values a numerical ranking and then sorting based on that ranking.

```
source=all_month.csv | eval Description=case(depth<=70, "Low", depth>70 AND depth<=300, "Mid", depth>300, "Deep") | stats count min(mag) max(mag) by Description | eval sort_field=case(Description="Low", 1, Description="Mid", 2, Description="Deep", 3) | sort sort_field
```

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

Description	count	min(Mag)	max(Mag)
Low	6236	-0.60	7.70
Mid	635	0.8	6.3
Deep	35	4.1	6.7

cidrmatch(<cidr>,<ip>)

Description

This function returns TRUE when an IP address, <ip>, belongs to a particular CIDR subnet, <cidr>.

Both <cidr> and <ip> are string arguments. If you specify a literal string value, instead of a field name, that value must be enclosed in double quotation marks.

The `cidrmatch` function supports IPv4 and IPv6 addresses and subnets that use CIDR notation.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `cidrmatch` and `if` functions to set a field, `isLocal`, to "local" if the field `ip` matches the subnet. If the `ip` field does not match the subnet, the `isLocal` field is set to "not local".

```
... | eval isLocal=if(cidrmatch("123.132.32.0/25", ip), "local", "not local")
```

The following example uses the `cidrmatch` function as a filter to remove events that do not match the `ip` address:

```
... | where cidrmatch("123.132.32.0/25", ip)
```

Extended examples for IPv4 addresses

You can use the `cidrmatch` function to identify CIDR IP addresses by subnet. The following example uses `cidrmatch` with the `eval` command to compare an IPv4 address with a subnet that uses CIDR notation to determine whether the IP address is a member of the subnet. If there is a match, the search returns true in a new field called `result`.

```
| makeresults | eval subnet="192.0.2.0/24", ip="192.0.3.0" | eval result=if(cidrmatch(subnet, ip), "true", "false")
```

The IP address is not in the subnet, so search displays `false` in the `result` field. The search results look something like this.

time	ip	result	subnet
2020-11-19 16:43:31	192.0.3.0	false	192.0.2.0/24

In the following example, `cidrmatch` evaluates the IPv4 address 192.0.2.56 to find out if it is in the subnet. This time,

instead of using the `eval` command with the `cidrmatch` function, we're using the `where` command, which eliminates any IP addresses that aren't within the subnet. This search compares the CIDR IP address with the subnet and filters the search results by returning the IP address only if it is true.

```
| makeresults | eval ip="192.0.2.56" | where cidrmatch("192.0.2.0/24", ip)
```

The IP address is located within the subnet, so it is displayed in the search results, which look like this.

time	ip
2020-11-19 16:43:31	192.0.2.56

It is worth noting that you can get the same results when using the `search` command, as shown in this example.

```
| makeresults | eval ip="192.0.2.56" | search ip="192.0.2.0/24"
```

The results of the search look like this.

time	ip
2020-11-19 16:43:31	192.0.2.56

Extended examples for IPv6 addresses

The following example uses `cidrmatch` with the `eval` command to compare an IPv6 address with a subnet that uses CIDR notation to determine whether the IP address is a member of the subnet. If there is a match, `search` returns true in a new field called `result`.

```
| makeresults | eval subnet="2001:0db8:ffff:ffff:ffff:ffff:ffff:ff00/120", ip="2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99" | eval result = if(cidrmatch(subnet, ip), "true", "false")
```

The IP address is located within the subnet, so `search` displays `true` in the `result` field. The search results look something like this.

time	ip	result	subnet
2020-11-19 16:43:31	2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99	true	2001:0db8:ffff:ffff:ffff:ffff:ffff:ff00/120

The following example is another way to use `cidrmatch` to identify which IP addresses are in a subnet. This time, instead of using the `eval` command with the `cidrmatch` function, we're using the `where` command. This search compares the CIDR IPv6 addresses with the specified subnet and filters the search results by returning only the IP addresses that are in the subnet.

```
| makeresults | eval ip="2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99" | where cidrmatch("2001:0db8:ffff:ffff:ffff:ffff:ff00/120", ip)
```

The search results look something like this.

time	ip
2020-11-19 16:43:31	2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99

See also

Commands

[iplocation](#)
[lookup](#)
[search](#)

coalesce(<values>)

Description

This function takes one or more values and returns the first value that is not NULL.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

You have a set of events where the IP address is extracted to either `clientip` or `ipaddress`. This example defines a new field called `ip`, that takes the value of either the `clientip` field or `ipaddress` field, depending on which field is not NULL (does not exist in that event). If both the `clientip` and `ipaddress` field exist in the event, this function returns the first argument, the `clientip` field.

```
... | eval ip=coalesce(clientip,ipaddress)
```

false()

Description

Use this function to return FALSE.

This function enables you to specify a conditional that is obviously false, for example `1==0`. You do not specify a field with this function.

Usage

This function is often used as an argument with other functions.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

if(<predicate>,<true_value>,<false_value>)

Description

If the `<predicate>` expression evaluates to TRUE, returns the `<true_value>`, otherwise the function returns the `<false_value>`.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `if` function is frequently used in combination with other functions.

Basic examples

The following example looks at the values of the field `error`. If `error=200`, the function returns `err=OK`. Otherwise the function returns `err=Error`.

```
... | eval err;if(error == 200, "OK", "Error")
```

The following example uses the `cidrmatch` and `if` functions to set a field, `isLocal`, to "local" if the field `ip` matches the subnet. If the `ip` field does not match the subnet, the `isLocal` field is set to "not local".

```
... | eval isLocal;if(cidrmatch("123.132.32.0/25",ip), "local", "not local")
```

in(<field>,<list>)

Description

The function returns TRUE if one of the values in the list matches a value that you specify.

This function takes a list of comma-separated values.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions with other commands.

The following syntax is supported:

```
...| where in(field,"value1","value2", ...)
...| where field in("value1","value2", ...)
...| eval new_field;if(in(field,"value1","value2", ...), "value-if_true","value-if-false")
```

The eval command cannot accept a Boolean value. You must specify the `in` function inside a function that can accept a Boolean value as input. Those functions are: `case`, `if`, and `validate`.

The string values must be enclosed in quotation marks. You cannot specify wildcard characters with the values to specify a group of similar values, such as HTTP error codes or CIDR IP address ranges. Use the IN operator instead.

The IN operator is similar to the `in` function. You can use the IN operator with the `search` and `tstats` commands. You can use wildcard characters in the VALUE-LIST with these commands.

Basic examples

The following example uses the `where` command to return `in=TRUE` if one of the values in the `status` field matches one of the values in the list.

```
... | where status in("400", "401", "403", "404")
```

The following example uses the `in` function as the first parameter for the `if` function. The evaluation expression returns TRUE if the value in the `status` field matches one of the values in the list.

```
... | eval error=if(in(status, "error", "failure", "severe"), "true", "false")
```

The following example uses the `where` command to return `in=TRUE` if the value 203.0.113.255 appears in either the `ipaddress` or `clientip` fields.

```
... | where "203.0.113.255" in(ipaddress, clientip)
```

Extended example

The following example combines the `in` function with the `if` function to evaluate the `status` field. The value of `true` is placed in the new field `error` if the `status` field contains one of the values 404, 500, or 503. Then a count is performed of the values in the `error` field.

```
... | eval error=if(in(status, "404", "500", "503"), "true", "false") | stats count by error
```

See also

Blogs

Smooth operator | Searching for multiple field values

like(<str>,<pattern>)

Description

This function returns TRUE only if `<str>` matches `<pattern>`. The match can be an exact match or a match using a wildcard:

- Use the percent (%) symbol as a wildcard for matching multiple characters
- Use the underscore (_) character as a wildcard to match a single character

The `<str>` can be a field name or a string value. The `<pattern>` must be a string expression enclosed in double quotation marks.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The following syntax is supported:

```
...|eval new_field=if(like(<str>, <pattern>)
...| where like(<str>, <pattern>)
...| where <str> LIKE <pattern>
```

The eval command cannot accept a Boolean value. You must specify the like function inside a function that can accept a Boolean value as input. Those functions are: case, if, and validate.

Basic examples

The following example returns `like=TRUE` if the field value starts with foo:

```
... | eval is_a_foo=if(like(field, "foo%"), "yes a foo", "not a foo")
```

The following example uses the `where` command to return `like=TRUE` if the `ipaddress` field starts with the value `198..`. The percent (%) symbol is a wildcard with the `like` function:

```
... | where like(ipaddress, "198.%")
```

lookup(<lookup_table>,<json_object>,<json_array>)

Description

This function performs a CSV lookup. It returns the output field or fields in the form of a JSON object.

The `lookup()` function is available only to Splunk Enterprise users.

Syntax

```
lookup("<lookup_table>", json_object("<input_field>","<match_field>,..."), json_array("<output_field>","..."))
```

Usage

You can use the `lookup()` function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `lookup()` function takes an `<input_field>` from a CSV `<lookup_table>`, finds events in the search result that have the `<match_field>`, and then identifies other field-value pairs from the CSV table that correspond to the `input_field` and adds them to the matched events in the form of a JSON object.

The `lookup()` requires a `<lookup_table>`. You can provide this either a CSV lookup file or CSV lookup definition, enclosed within quotation marks. To provide a file, give the full filename of a CSV lookup file that is stored in the global lookups directory (`$/SPLUNK_HOME/etc/system/lookups/`) or in a lookup directory that matches your current app context, such as `$/SPLUNK_HOME/etc/users/<user>/<app>/lookups/`.

If the first quoted string does not end in ".csv", the `eval` processor assumes it is the name of a CSV lookup definition. Specified CSV lookup definitions must be shared globally. CSV lookup definitions cannot be private or shared to a specific app.

Specify a lookup definition if you want the various settings associated with the definition to apply, such as limits on matches, case-sensitive match options, and so on.

A `lookup()` function can use multiple `<input_field>/<match_field>` pairs to identify events, and multiple `<output_field>` values can be applied to those events. Here is an example of valid `lookup()` syntax with multiple inputs, matches, and outputs.

```
... | eval <string>=lookup("<lookup_table>", json_object("<input_field1>", <match_field1>, "<input_field2>", <match_field2>), json_array("<output_field1>", "<output_field2>", "<output_field3>")
```

For more information about uploading CSV lookup files and creating CSV lookup definitions, see Define a CSV lookup in Splunk Web in the *Knowledge Manager Manual*.

The `lookup()` function uses two JSON functions for `eval: json_object` and `json_array`. JSON functions allow the `eval` processor to efficiently group things together. For more information, see [JSON functions](#) in the *Search Reference*.

Examples

These examples show different ways to use the `lookup()` function.

1. Simple example that returns a JSON object with an array

This simple `makeresults` example returns an array that illustrates what `status_description` values are paired in the `http_status.csv` lookup table with a `status_type` of `Successful`.

This search returns: `output={"status_description": ["OK", "Created", "Accepted", "Non-Authoritative Information", "No Content", "Reset Content", "Partial Content"]}`

```
| makeresults | eval type = "Successful" | eval output=lookup("http_status.csv", json_object("status_type", type), json_array("status_description"))
```

2. Example of a search with multiple input and match field pairs

This search employs multiple input and match field pairs to show that an event with `type="Successful"` and `status="200"` matches a `status_description` of `OK` in the `http_status.csv` lookup table.

This search returns: `output={"status_description": "OK"}`

```
| makeresults | eval type = "Successful", status="200" | eval output=lookup("http_status.csv", json_object("status_type", type, "status", status), json_array("status_description"))
```

3. Get counts of HTTP status description and type pairs

This example matches values of a `status` field in a `http_status.csv` lookup file with values of `status` fields in the returned events. It then generates JSON objects as values of a `status_details` field, with the corresponding `status_description` and `status_type` field-value pairs, and adds them to the events. Finally, it provides counts of the JSON objects, broken out by object.

Here is an example of a JSON object returned by this search:

```
status_details=JSON:{ "status_description": "Created", "status_type": "Successful" }

index=_internal | eval output=lookup("http_status.csv", json_object("status", status), json_array("status_description", "status_type")), status_details="JSON:.output | stats count by status_details
```

4. Get counts of the HTTP status description values that have been applied to your events by a HTTP status eval lookup

This example shows how you can nest a `lookup` function inside another `eval` function. In this case it is the `json_extract` JSON function. This extracts `status_description` field-value pairs from the `json_array` objects and applies them to corresponding events. The search then returns a count of events with `status_description` fields, broken out by `status_description` value.

Here is an example of an extracted `status_description` value returned by this search. Compare it to the result returned by the third example: `status_details=Created`

```
index=_internal | eval status_details=json_extract(lookup("http_status.csv", json_object("status", status), json_array("status_description")), "status_description") | stats count by status_details
```

match(<str>, <regex>)

Description

This function returns TRUE if the regular expression `<regex>` finds a match against any substring of the string value `<str>`. Otherwise returns FALSE.

Usage

The `match` function is regular expression based. For example use the backslash (\) character to escape a special character, such as a quotation mark. Use the pipe (|) character to specify an OR condition.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns TRUE if, and only if, `field` matches the basic pattern of an IP address. This examples uses the caret (^) character and the dollar (\$) symbol to perform a full match.

```
... | eval n;if(match(field, "^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}$"), 1, 0)
```

The following example uses the `match` function in an `<eval-expression>`. The `<str>` is a calculated field called `test`. The `<regex>` is the string yes.

```
... | eval matches = if(match(test,"yes"), 1, 0)
```

If the value is stored with quotation marks, you must use the backslash (\) character to escape the embedded quotation marks. For example:

```
| makeresults | eval test="\\"yes\\"" | eval matches = if(match(test, "\\"yes\\\""), 1, 0)
```

null()

Description

This function takes no arguments and returns NULL. The evaluation engine uses NULL to represent "no value". Setting a field value to NULL clears the field value.

Usage

NULL values are field values that are missing in some results but present in another results.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

Suppose you want to calculate the average of the values in a field, but several of the values are zero. If the zeros are placeholders for no value, the zeros will interfere with creating an accurate average. You can use the `null` function to remove the zeros.

See also

- You can use the `fillnull` command to replace NULL values with a specified value.
- You can use the `nullif(X, Y)` function to compare two fields and return NULL if X = Y.

nullif(<field1>, <field2>)

Description

This function compares the values in two fields and returns NULL if the value in `<field1>` is equal to the value in `<field2>`. Otherwise the function returns the value in `<field1>`.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

Using the `makeresults` command, the following search creates a field called `names`. Another field called `ponies` is created based on the `names` field. The `if` function is used to change the name `buttercup` to `mistmane` in the `ponies` field.

```
| makeresults | eval names="buttercup rarity tenderhoof dash" | makemv delim=" " names | mvexpand names | eval ponies = if(names=="buttercup", "mistmane", names)
```

The results look like this:

_time	names	ponies
2022-10-17 14:57:12	buttercup	mistmane
2022-10-17 14:57:12	rarity	rarity
2022-10-17 14:57:12	tenderhoof	tenderhoof
2022-10-17 14:57:12	dash	dash

Using the `nullif` function, you can compare the values in the `names` and `ponies` fields. If the values are different, the value from the first field specified are displayed in the `compare` field. If the values are the same, no value is returned.

```
... eval compare = nullif(names, ponies)
```

The results look like this:

_time	compare	names	ponies
2022-10-17 14:57:12	buttercup	buttercup	mistmane

_time	compare	names	ponies
2022-10-17 14:57:12		rarity	rarity
2022-10-17 14:57:12		tenderhoof	tenderhoof
2022-10-17 14:57:12		dash	dash

searchmatch(<search_str>)

Description

This function returns TRUE if the event matches the search string.

Usage

To use the `searchmatch` function with the `eval` command, you must use the `searchmatch` function inside the `if` function.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `makeresults` command to create some simple results. The `searchmatch` function is used to determine if any of the results match the search string "x=hi y=*".

```
| makeresults 1 | eval _raw = "x=hi y=bye" | eval x="hi" | eval y="bye" | eval test=if(searchmatch("x=hi y=*"), "yes", "no") | table _raw test x y
```

The result of the `if` function is `yes`; the results match the search string specified with the `searchmatch` function.

true()

Description

Use this function to return TRUE.

This function enables you to specify a condition that is obviously true, for example `1==1`. You do not specify a field with this function.

Usage

This function is often used as an argument with other functions.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

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.

The following example shows how to use the `true()` function to provide a default value to the `case` function. If the values in the status field are not 200, or 404, the value used is Other.

```
sourcetype=access_* | eval description=case(status==200, "OK", status==404, "Not found", true(), "Other") |  
table status description
```

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

status	description
200	OK
200	OK
408	Other
200	OK
404	Not found
200	OK
200	OK
406	Other
200	OK

validate(<condition>, <value>,...)

Description

This function takes a list of conditions and values and returns the value that corresponds to the condition that evaluates to FALSE. This function defaults to NULL if all conditions evaluate to TRUE.

This function is the opposite of the `case` function.

Usage

The `<condition>` arguments must be expressions.

The `<value>` arguments must be strings. You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example runs a simple check for valid ports.

```
... | eval n=validate(isint(port), "ERROR: Port is not an integer", port >= 1 AND port <= 65535, "ERROR:  
Port is out of range")
```

Conversion functions

The following list contains the functions that you can use to mask IP addresses and convert numbers to strings and strings to numbers.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

ipmask(<mask>,<ip>)

Description

This function generates a new masked IP address by applying a mask to an IP address through a bitwise AND operation. You can use this function to simplify the isolation of an IPv4 address octet without splitting the IP address.

Usage

The <mask> must be a valid IPv4 address. The <IP> must be a valid IPv4 address or a field name where the field value is a valid IPv4 address.

A valid IPv4 address is a quad-dotted notation of four decimal integers, each ranging from 0 to 255.

For the <mask> argument, you can specify one of the default subnet masks such as 255.255.255.0.

You can use this function with the `eval` command, and as part of eval expressions.

Basic examples

The following example shows how to use the `ipmask` function with the `eval` command:

```
... | eval maskedIP = ipmask("255.255.255.0", "10.20.30.120")
```

The output of this example is 10.20.30.0.

The following example shows how to use the `ipmask` function in the SELECT clause of the `from` command:

```
... | eval maskedIP = ipmask("0.255.0.244", clientip) AS maskedip
```

This search masks every IP address in the `clientip` field and returns the results in an aliased field called `maskedip`.

The following example shows how to use the `ipmask` function in the WHERE clause of the `from` command to filter the events on a specific mask value:

```
... | where ipmask("0.255.0.224", clientip)="10.20.30.120"
```

In this example, the masked value is 0.20.0.96.

printf(<format>,<arguments>)

Description

This function builds a string value, based on a string format and the values specified. You can specify zero or more values. The values can be strings, numbers, computations, or fields.

The SPL `printf` function is similar to the C `sprintf()` function and similar functions in other languages such as Python, Perl, and Ruby.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

format

Description: The `format` is a character string that can include one or more format conversion specifiers. Each conversion specifier can include optional components such as flag characters, width specifications, and precision specifications. The `format` must be enclosed in quotation marks.

Syntax: `"(%[flags][width][.precision]<conversionSpecifier>)..."`

arguments

Description: The `arguments` are optional and can include the width, precision, and the value to format. The value can be a string, number, or field name.

Syntax: `[width][.precision][value]`

Supported conversion specifiers

The following table describes the supported conversion specifiers.

Conversion specifier	Alias	Description	Examples
%a or %A		Floating point number in hexadecimal format	This example returns the value of <code>pi</code> to 3 decimal points, in hexadecimal format. <code>printf("%.3A", pi())</code> which returns <code>0X1.922P+1</code>
%c		Single Unicode code point	This example returns the unicode code point for 65 and the first letter of the string "Foo". <code>printf("%c,%c", 65, "Foo")</code> which returns <code>A,F</code>
%d	%i	Signed decimal integer	This example returns the positive or negative integer values, including any signs specified with those values. <code>printf("%d,%i,%d", -2, +4, 30)</code> which returns <code>-2,4,30</code>
%e or %E		Floating point number, exponential format	This example returns the number 5139 in exponential format with 2 decimal points. <code>printf("%.2e", 5139)</code> which returns <code>5.14e+03</code>
%f or %F		Floating point number	This example returns the value of <code>pi</code> to 2 decimal points. <code>printf("%.2f", pi())</code> which returns <code>3.14</code>
%g or %G		Floating point number. This specifier uses either %e or %f depending on the range of the numbers being formatted.	This example returns the value of <code>pi</code> to 2 decimal points (using the %f specifier) and the number 123 in exponential format with 2 decimal points (using %e specifier). <code>printf("%.2g,%.2g", pi(), 123)</code> which returns <code>3.1,1.2e+02</code>
%o		Unsigned octal number	This example returns the base-8 number for 255.

Conversion specifier	Alias	Description	Examples
			<code>printf("%o", 255)</code> which returns 377
%s	%z	String	This example returns the concatenated string values of "foo" and "bar". <code>printf("%s%z", "foo", "bar")</code> which returns foobar
%u		Unsigned, or non-negative, decimal integer	This example returns the integer value of the number in the argument. <code>printf("%u", 99)</code> which returns 99
%x or %X	%p	Unsigned hexadecimal number (lowercase or uppercase)	This example returns the hexadecimal values that are equivalent to the numbers in the arguments. This example shows both upper and lowercase results when using this specifier. <code>printf("%x, %X, %p", 10, 10, 10)</code> which returns a,A,A
%%		Percent sign	This example returns the string value with a percent sign. <code>printf("100%%")</code> which returns 100%

Flag characters

The following table describes the supported flag characters.

Flag characters	Description	Examples
single quote or apostrophe (')	Adds commas as the thousands separator.	<code>printf("%'d", 12345)</code> which returns 12,345
dash or minus (-)	Left justify. If this flag is not specified, the result keeps its default justification. The <code>printf</code> function supports right justification of results only when it formats that way by default.	<code>printf("%-4d", 1)</code> which returns 1 which is left justified in the output.
zero (0)	Zero pad	This example returns the value in the argument with leading zeros such that the number has 4 digits. <code>printf("%04d", 1)</code> which returns 0001
plus (+)	Always include the sign (+ or -). If this flag is not specified, the conversion displays a sign only for negative values.	<code>printf("%+4d", 1)</code> which returns +1
<space>	Reserve space for the sign. If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a <space> is added as a prefixed to the result. If both the <space> and + flags are specified, the <space> flag is ignored.	<code>printf("% -4d", 1)</code> which returns 1
hash, number, or pound (#)	Use an alternate form. For the %o conversion specifier, the # flag increases the precision to force the first digit of the result to be zero. For %x or %X conversion specifiers, a non-zero result has 0x (or 0X) prefixed to it. For %a, %A, %e, %E, %f, %F, %%g , and G conversion	<code>printf("%#x", 1)</code> which returns 0x1

Flag characters	Description	Examples
	specifiers, the result always contains a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For %g and %G conversion specifiers, trailing zeros are not removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.	

Specifying field width

You can use an asterisk (*) with the `printf` function to return the field width or precision from an argument.

Examples

The following example returns the positive or negative integer values, including any signs specified with those values.

```
printf("%*d", 5, 123) which returns 123
```

The following example returns the floating point number with 1 decimal point.

```
printf("%.1f", 1, 1.23) which returns 1.2
```

The following example returns the value of `pi()` in exponential format with 2 decimal points.

```
printf("%.*e", 9, 2, pi()) which returns 3.14e+00
```

The field width can be expressed using a number or an argument denoted with an asterisk (*) character.

Field width specifier	Description	Examples
number	The minimum number of characters to print. If the value to print is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.	
* (asterisk)	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.	

Specifying precision

Precision	Description
%d, %i, %o, %u, %x or %X	Precision specifies the minimum number of digits to be returned. If the value to be returned is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is returned for the value 0.
%a or %A, %e or %E, %f or %F	This is the number of digits to be returned after the decimal point. The default is 6 .
%g or %G	This is the maximum number of significant digits to be returned.
%s	This is the maximum number of characters to be returned. By default all characters are printed until the ending null character is encountered.
Specifying the period without a precision value	If the period is specified without an explicit value for precision, 0 is assumed.

Precision	Description
Specifying an asterisk for the precision value, for example <code>.*</code>	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Unsupported conversion specifiers

There are a few conversion specifiers from the C `sprintf()` function that are not supported, including:

- `%C`, however `%c` is supported
- `%n`
- `%S`, however `%s` is supported
- `%<num>$` specifier for picking which argument to use

Basic examples

This example creates a new field called `new_field` and creates string values based on the values in `field_one` and `field_two`. The values are formatted with 4 digits before the decimal and 4 digits after the decimal. The `-` specifies to left justify the string values. The `30` specifies the width of the field.

```
... | eval new_field=printf("%04.4f %-30s", field_one, field_two)
```

tonumber(<str>,<base>)

Description

This function converts a string `<str>` to a number. The string can be a field name or a literal value.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `<base>` is optional and used to define the base of the number in `<str>`. The `<base>` can be 2 to 36. The default is 10 to correspond to the decimal system.

If the `tonumber` function cannot parse a field value to a number, for example if the value contains a leading and trailing space, the function returns NULL. Use the `trim` function to remove leading or trailing spaces.

If the `tonumber` function cannot parse a literal string to a number, it returns an error.

Basic examples

The following example converts the string values for the `store_sales` field to numbers.

```
... | eval n=tonumber(store_sales)
```

The following example takes the hexadecimal number and uses a `<base>` of 16 to return the number "164".

```
... | eval n=tonumber("0A4",16)
```

The following example trims any leading or trailing spaces from the values in the `celsius` field before converting it to a number.

```
... | eval temperature=tonumber(trim(celsius))
```

tostring(<value>,<format>)

Description

This function converts a value to a string. If the value is a number, this function reformats it as a string. If the value is a Boolean value, it returns the corresponding string value, "True" or "False".

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `<value>` is required.

When used with the `eval` command, the values might not sort as expected because the values are converted to ASCII. Use the `fieldformat` command with the `tostring` function to format the displayed values. The underlying values are not changed with the `fieldformat` command.

If `<value>` is a number, the `<format>` is optional. Supported formats are `"hex"`, `"commas"`, or `"duration"`.

Examples	Description
<code>tostring(<value>,"hex")</code>	Converts the value to hexadecimal.
<code>tostring(<value>,"commas")</code>	Formats the value with commas. If the number includes decimals, the function rounds to nearest two decimal places.
<code>tostring(<value>,"duration")</code>	Converts a value in seconds to the readable time format HH:MM:SS.

Basic examples

The following example returns "True 0xF 12,345.68".

```
... | eval n=tostring(l==1) + " " + tostring(15, "hex") + " " + tostring(12345.6789, "commas")
```

The following example returns `foo=615` and `foo2=00:10:15`. The 615 seconds is converted into minutes and seconds.

```
... | eval foo=615 | eval foo2 = tostring(foo, "duration")
```

The following example formats the column `totalSales` to display values with a currency symbol and commas. You must use a period between the currency value and the `tostring` function.

```
... | fieldformat totalSales="$".tostring(totalSales, "commas")
```

See also

Commands
 [convert](#)
Functions
 [strftime](#)

Cryptographic functions

The following list contains the functions that you can use to compute the secure hash of string values.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

md5(str)

Description

This function computes and returns the MD5 hash of a string value.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns a new field `n` with a message-digest (MD5) 128-bit hash value for the phrase "Hello World".

```
... | eval n=md5("Hello World")
```

The following example creates a large random string.

```
| makeresults count=32768 | eval message=md5("".random()) | stats values(message) as message | eval message = mvjoin(message, "")
```

- The `makeresults` command creates 32768 results with timestamps.
- The `eval` command creates a new field called `message`:
 - ◆ The `random` function returns a random numeric field value for each of the 32768 results. The `"".` makes the numeric number generated by the `random` function into a string value.
 - ◆ The `md5` function creates a 128-bit hash value from the string value.
 - ◆ The results of the `md5` function are placed into the `message` field created by the `eval` command.
- The `stats` command with the `values` function is used to convert the individual random values into one multivalue result.
- The `eval` command with the `mvjoin` function is used to combine the multivalue entry into a single value.

sha1(str)

Description

This function computes and returns the secure hash of a string value based on the FIPS compliant SHA-1 hash function.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=sha1("Put that in your | and Splunk it.")
```

sha256(str)

Description

This function computes and returns the secure hash of a string value based on the FIPS compliant SHA-256 (SHA-2 family) hash function.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=sha256("Can you SPL?")
```

sha512(str)

Description

This function computes and returns the secure hash of a string value based on the FIPS compliant SHA-512 (SHA-2 family) hash function.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=sha512("You bet your sweet SaaS.")
```

Date and Time functions

The following list contains the functions that you can use to calculate dates and time.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

In addition to the functions listed in this topic, there are also variables and modifiers that you can use in searches.

- [Date and time format variables](#)
- [Time modifiers](#)

now()

Description

This function takes no arguments and returns the time that the search was started.

Usage

The `now()` function is often used with other data and time functions.

The time returned by the `now()` function is represented in UNIX time, or in seconds since Epoch time.

When used in a search, this function returns the UNIX time when the search is run. If you want to return the UNIX time when each result is returned, use the `time()` function instead.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example determines the UNIX time value of the start of yesterday, based on the value of `now()`. This example uses a "snap-to" time modifier to snap to the the start of the day. See [How to specify relative time modifiers](#).

```
... | eval n=relative_time(now(), "-1d@d")
```

Extended example

If you are looking for events that occurred within the last 30 minutes you need to calculate the event hour, event minute, the current hour, and the current minute. You use the `now()` function to calculate the current hour (`curHour`) and current minute (`curMin`). The event timestamp, in the `_time` field, is used to calculate the event hour (`eventHour`) and event minute (`eventMin`). For example:

```
... earliest=-30d | eval eventHour=strftime(_time,"%H") | eval eventMin=strftime(_time,"%M") | eval curHour=strftime(now(),"%H") | eval curMin=strftime(now(),"%M") | where (eventHour=curHour and eventMin > curMin - 30) or (curMin < 30 and eventHour=curHour-1 and eventMin>curMin+30) | bucket _time span=1d | chart count by _time
```

relative_time(<time>,<specifier>)

Description

This function takes a UNIX time as the first argument and a relative time specifier as the second argument and returns the UNIX time value of `<specifier>` applied to `<time>`.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example determines the UNIX time value of the start of yesterday, based on the value of `now()`. This example uses a "snap-to" time modifier to snap to the the start of the day. See [How to specify relative time modifiers](#).

```
... | eval n=relative_time(now(), "-1d@d")
```

The following example specifies an earliest time of 2 hours ago snapped to the hour and a latest time of 1 hour ago snapped to the hour. The offset `-2h` is processed first, followed by the snap-to time `@h`.

```
... | where _time>relative_time(now(), "-2h@h") AND _time<relative_time(now(), "-1h@h")
```

strftime(<time>,<format>)

Description

This function takes a UNIX time value as the first argument and renders the time as a string using the format specified. The UNIX time must be in seconds. Use the first 10 digits of a UNIX time to use the time in seconds.

Usage

If the time is in milliseconds, microseconds, or nanoseconds you must convert the time into seconds. You can use the `pow` function to convert the number.

- To convert from milliseconds to seconds, divide the number by 1000 or 10^3 .
- To convert from microseconds to seconds, divide the number by 10^6 .
- To convert from nanoseconds to seconds, divide the number by 10^9 .

The following search uses the `pow` function to convert from nanoseconds to seconds:

```
| makeresults | eval StartTimestamp="1521467703049000000" | eval starttime=strftime(StartTimestamp/pow(10, 9), "%Y-%m-%dT%H:%M:%S.%Q")
```

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

StartTimeStamp	_time	starttime
1521467703049000000	2018-08-10 09:04:00	2018-03-19T06:55:03.049

In these results the `_time` value is the date and time when the search was run.

For a list and descriptions of format options, see [Date and time format variables](#).

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns the hour and minute from the `_time` field.

```
... | eval hour_min=strftime(_time, "%H:%M")
```

If the `_time` field value is `2022-08-10 11:48:23`, the value returned in the `hour_min` field is `11:48`.

The following example creates a new field called `starttime` in your search results. For the `strftime` values, the `now()` function is used to generate the current UNIX time and date and time variables are used to specify the ISO 8601 timestamp format;

```
... | eval starttime=strftime(now(),"%Y-%m-%dT%H:%M:%S.%Q")
```

The results look something like this:

_starttime
2022-02-11T01:55:00.000

For more information about date and time variables, see [Date and time format variables](#).

Extended example

The following example creates a single result using the `makeresults` command.

```
| makeresults
```

For example:

_time
2022-08-14 14:00:15

The `_time` field is stored in UNIX time, even though it displays in a human readable format. To convert the UNIX time to some other format, you use the `strftime` function with the date and time format variables. The variables must be in quotations marks.

For example, to return the week of the year that an event occurred in, use the `%V` variable.

```
| makeresults | eval week=strftime(_time,"%V")
```

The results show that August 14th occurred in week 33.

_time	week
2022-08-14 14:00:15	33

To return the date and time with subseconds and the time designator (the letter T) that precedes the time components of the format, use the `%Y-%m-%dT%H:%M:%S.%Q` variables. For example:

```
| makeresults | eval mytime=strftime(_time,"%Y-%m-%dT%H:%M:%S.%Q")
```

The results are:

_time	mytime
2022-08-14 14:00:15	2022-08-14T14:00:15.000

strftime(<str>,<format>)

Description

This function takes a time represented by a string and parses the time into a UNIX timestamp format. You use date and time variables to specify the format that matches string. The `strftime` function doesn't work with timestamps that consist of only a month and year. The timestamps must include a day.

For example, if string X is `2022-08-13 11:22:33`, the format Y must be `%Y-%m-%d %H:%M:%S`. The string X date must be January 1, 1971 or later. The `strftime` function takes any date from January 1, 1971 or later, and calculates the UNIX time, in seconds, from January 1, 1970 to the date you provide.

The `_time` field is in UNIX time. In Splunk Web, the `_time` field appears in a human readable format in the UI but is stored in UNIX time. If you attempt to use the `strftime` function on the `_time` field, no action is performed on the values in the field.

Usage

With the `strftime` function, you must specify the time format of the string so that the function can convert the string time into the correct UNIX time. The following table shows some examples:

String time	Matching time format variables
Mon July 23 2022 17:19:01.89	%a %B %d %Y %H:%M:%S.%N
Mon 7/23/2022 17:19:01.89	%a %m/%d/%Y %H.%M:%S.%N
2022/07/23 17:19:01.89	%Y/%m/%d %H:%M:%S.%N
2022-07-23T17:19:01.89	%Y-%m-%dT%H:%M:%S.%N

For a list and descriptions of format options, see [Date and time format variables](#).

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

If the values in the `timeStr` field are hours and minutes, such as `11:59`, the following example returns the time as a timestamp:

```
... | eval n=strptime(timeStr, "%H:%M")
```

Extended example

This example shows the results of using the `strftime` function. The following search does several things:

- The `gentimes` command generates a set of times with 6 hour intervals. This command returns four fields: `starttime`, `starthuman`, `endtime`, and `endhuman`.
- The `fields` command returns only the `starthuman` and `endhuman` fields.
- The `eval` command takes the string time values in the `starthuman` field and returns the UNIX time that corresponds to the string time values.

```
| gentimes start=8/13/18 increment=6h | fields starthuman endhuman | eval startunix=strptime(starthuman,"%a %B %d %H:%M:%S.%N %Y")
```

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

starthuman	endhuman	startunix
Mon Aug 13 00:00:00 2018	Mon Aug 13 05:59:59 2018	1534143600.000000
Mon Aug 13 06:00:00 2018	Mon Aug 13 11:59:59 2018	1534165200.000000
Mon Aug 13 12:00:00 2018	Mon Aug 13 17:59:59 2018	1534186800.000000
Mon Aug 13 18:00:00 2018	Mon Aug 13 23:59:59 2018	1534208400.000000
Tue Aug 14 00:00:00 2018	Tue Aug 14 05:59:59 2018	1534230000.000000
Tue Aug 14 06:00:00 2018	Tue Aug 14 11:59:59 2018	1534251600.000000
Tue Aug 14 12:00:00 2018	Tue Aug 14 17:59:59 2018	1534273200.000000
Tue Aug 14 18:00:00 2018	Tue Aug 14 23:59:59 2018	1534294800.000000

time()

Description

This function returns the wall-clock time, in the UNIX time format, with microsecond resolution.

Usage

The value of the `time()` function will be different for each event, based on when that event was processed by the `eval` command.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example shows the results of using the `time()` function. The following search does several things"

- The `gentimes` command generates a set of times with 6 hour intervals. This command returns four fields: `starttime`, `starthuman`, `endtime`, and `endhuman`.
- The `fields` command returns only the `starttime` and `starthuman` fields.
- The first `eval` command takes the numbers in the `starttime` field and returns them with microseconds included.
- The second `eval` command creates the `testtime` field and returns the UNIX time at the instant the result was processed by the `eval` command.

```
| gentimes start=8/13/18 increment=6h | fields starttime starthuman | eval epoch_time=strptime(starttime,"%s") | eval testtime=time()
```

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

starttime	starthuman	epoch_time	testtime
1534143600	Mon Aug 13 00:00:00 2018	1534143600.000000	1534376565.299298
1534165200	Mon Aug 13 06:00:00 2018	1534165200.000000	1534376565.299300
1534186800	Mon Aug 13 12:00:00 2018	1534186800.000000	1534376565.299302

starttime	starthuman	epoch_time	testtime
1534208400	Mon Aug 13 18:00:00 2018	1534208400.000000	1534376565.299304
1534230000	Tue Aug 14 00:00:00 2018	1534230000.000000	1534376565.299305
1534251600	Tue Aug 14 06:00:00 2018	1534251600.000000	1534376565.299306
1534273200	Tue Aug 14 12:00:00 2018	1534273200.000000	1534376565.299308
1534294800	Tue Aug 14 18:00:00 2018	1534294800.000000	1534376565.299309

Notice the difference in the microseconds between the values in the `epoch_time` and `test_time` fields. You can see that the `test_time` values increase with each result.

Informational functions

The following list contains the functions that you can use to return information about a value.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

isbool(<value>)

Description

This function takes one argument `<value>` and evaluates whether `<value>` is a Boolean data type. The function returns TRUE if `<value>` is Boolean.

Usage

Use this function with other functions that return Boolean data types, such as `cidrmatch` and `mvfind`.

This function cannot be used to determine if field values are "true" or "false" because field values are either string or number data types. Instead, use syntax such as `<fieldname>=true` OR `<fieldname>=false` to determine field values.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

isint(<value>)

Description

This function takes one argument `<value>` and returns TRUE if `<value>` is an integer.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `isint` function with the `if` function. A field, "n", is added to each result with a value of "int" or "not int", depending on the result of the `isint` function. If the value of "field" is a number, the `isint` function returns TRUE and the value adds the value "int" to the "n" field.

```
... | eval n=if(isint(field),"int", "not int")
```

The following example shows how to use the `isint` function with the `where` command.

```
... | where isint(field)
```

isnonnull(<value>)

Description

This function takes one argument <value> and returns TRUE if <value> is not NULL.

Usage

This function is useful for checking for whether or not a field contains a value.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `isnonnull` function with the `if` function. A field, "n", is added to each result with a value of "yes" or "no", depending on the result of the `isnonnull` function. If the value of "field" is a number, the `isnonnull` function returns TRUE and the value adds the value "yes" to the "n" field.

```
... | eval n=if(isnonnull(field),"yes", "no")
```

The following example shows how to use the `isnonnull` function with the `where` command.

```
... | where isnonnull(field)
```

isnull(<value>)

Description

This function takes one argument <value> and returns TRUE if <value> is NULL.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `isnull` function with the `if` function. A field, "n", is added to each result with a value of "yes" or "no", depending on the result of the `isnull` function. If there is no value for "field" in a result, the `isnull` function returns TRUE and adds the value "yes" to the "n" field.

```
... | eval n=if(isnull(field), "yes", "no")
```

The following example shows how to use the `isnull` function with the `where` command.

```
... | where isnull(field)
```

isnum(<value>)

Description

This function takes one argument `<value>` and returns TRUE if `<value>` is a number.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `isnum` function with the `if` function. A field, "n", is added to each result with a value of "yes" or "no", depending on the result of the `isnum` function. If the value of "field" is a number, the `isnum` function returns TRUE and the value adds the value "yes" to the "n" field.

```
... | eval n=if(isnum(field), "yes", "no")
```

The following example shows how to use the `isnum` function with the `where` command.

```
... | where isnum(field)
```

isstr(<value>)

Description

This function takes one argument `<value>` and returns TRUE if `<value>` is a string.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example uses the `isstr` function with the `if` function. A field, "n", is added to each result with a value of "yes" or "no", depending on the result of the `isstr` function. If the value of "field" is a string, the `isstr` function returns

TRUE and the value adds the value "yes" to the "n" field.

```
... | eval n=if(isstr(field), "yes", "no")
```

The following example shows how to use the `isstr` function with the `where` command.

```
... | where isstr(field)
```

typeof(<value>)

Description

This function takes one argument <value> and returns the data type of the argument.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example takes one argument and returns a string representation of its type. This example returns "NumberStringBoolInvalid"

```
... | eval n=typeof(12) + typeof("string") + typeof(1==2) + typeof(badfield)
```

The following example creates a single result using the `makeresults` command.

```
| makeresults
```

For example:

_time
2018-08-14 14:00:15

To determine the data type of the `_time` field, use the `eval` command with the `typeof` function. For example:

```
| makeresults | eval t=typeof(_time)
```

The results are:

_time	t
2018-08-14 14:00:15	Number

JSON functions

The following table describes the functions that are available for you to use to create or manipulate JSON objects:

Description	JSON function
Create a new JSON object from key-value pairs.	<code>json_object</code>

Description	JSON function
Evaluate whether a value can be parsed as JSON. If the value is JSON, the function returns the value. Otherwise, the function returns null.	<code>json</code>
Append elements to the contents of a valid JSON object.	<code>json_append</code>
Create a JSON array using a list of values.	<code>json_array</code>
Map the elements of a JSON array to a multivalued field.	<code>json_array_to_mv</code>
Extend the contents of a valid JSON object with the values of an array.	<code>json_extend</code>
Return either a JSON array or a Splunk software native type value from a field and zero or more paths.	<code>json_extract</code>
Return Splunk software native type values from a piece of JSON by matching literal strings in the event and extracting them as keys.	<code>json_extract_exact</code>
Return the keys from the key-value pairs in a JSON object. The keys are returned as a JSON array.	<code>json_keys</code>
Insert or overwrite values for a JSON node with the values provided and return an updated JSON object.	<code>json_set</code>
Generate or overwrite a JSON object using the key-value pairs specified.	<code>json_set_exact</code>
Evaluate whether a JSON object uses valid JSON syntax and returns either TRUE or FALSE.	<code>json_valid</code>

json_object(<members>)

Creates a new JSON object from members of key-value pairs.

Usage

If you specify a string for a `<key>` or `<value>`, you must enclose the string in double quotation marks. A `<key>` must be a string. A `<value>` can be a string, number, Boolean, null, multivalue field, array, or another JSON object.

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Examples

These examples show different ways to use the `json_object` function to create JSON objects in your events.

1. Create a basic JSON object

The following example creates a basic JSON object `{ "name": "maria" }`.

```
... | eval name = json_object("name", "maria")
```

2. Create a JSON object using a multivalue field

The following example creates a multivalue field called `firstnames` that uses the key `name` and contains the values "maria" and "arun". The JSON object created is `{ "name": ["maria", "arun"] }`.

```
... | eval firstnames = json_object("name", json_array("maria", "arun"))
```

3. Create a JSON object using a JSON array

The following example creates a JSON object that uses a JSON array for the values.

```
... | eval locations = json_object("cities", json_array("London", "Sydney", "Berlin", "Santiago"))
```

The result is the JSON object { "cities": ["London", "Sydney", "Berlin", "Santiago"] }.

4. Create a nested JSON object

The following example creates a nested JSON object that uses other JSON objects and a multivalue or JSON array field called `gamelist`.

```
... | eval gamelist = json_array("Pandemic", "Forbidden Island", "Castle Panic"), games = json_object("category", json_object("boardgames", json_object("cooperative", gamelist)))
```

The result is this JSON object:

```
{
  "games": {
    "category": {
      "boardgames": {
        "cooperative": [ "Pandemic", "Forbidden Island", "Castle Panic" ]
      }
    }
  }
}json(<value>)
```

Evaluates whether a value can be parsed as JSON. If the value is in a valid JSON format, the function returns the value. Otherwise, the function returns null.

Usage

A `<value>` can be any kind of value such as string, number, Boolean, null, or JSON array or object.

Examples

1. Identify a JSON value

This example shows how you can use the `json` function to confirm that a value is JSON. The following search verifies that `{"animal" : "pony"}` is a JSON value by returning its value, `{"animal":"pony"}`.

```
... | eval animals = json_object("animal", "pony"), result = json(animals)
```

The search results look something like this:

_time	animals	result
2023-02-22 14:39:50	{"animal": "pony"}	{"animal": "pony"}

2. Compare multiple results to identify JSON values

The following example shows how to use the `json` function to determine if the values in a field are JSON arrays or objects.

Consider the following search results:

_time	bridges	city
2023-04-26 21:10:45	[{"bridges", {"name": "Tower Bridge"}, {"length": "801"}, {"name": "Millennium Bridge"}, {"length": "1066"}]	London
2023-04-26 21:10:45	[{"bridges", {"name": "Rialto Bridge"}, {"length": "157"}, {"name": "Bridge of Sighs"}, {"length": "36"}, {"name": "Ponte della Paglia"}]	Venice
2023-04-26 21:10:45	Golden Gate Bridge	San Francisco

When you add the `json` evaluation function to the following search, the results in the `bridgesAsJson` field identifies which values in the `bridges` field are JSON values:

```
... | eval bridgesAsJson = json(bridges)
```

When the value is JSON, the value is returned in the `bridgesAsJson` field. When the value is not JSON, the function returns null. The results look like something like this:

_time	bridges	bridgesAsJson	city
2023-04-26 21:10:45	[{"bridges", {"name": "Tower Bridge"}, {"length": "801"}, {"name": "Millennium Bridge"}, {"length": "1066"}]	[{"name": "Tower Bridge", "length": "801"}, {"name": "Millennium Bridge", "length": "1066"}]	London
2023-04-26 21:10:45	[{"name": "Rialto Bridge", "length": "157"}, {"name": "Bridge of Sighs", "length": "36"}, {"name": "Ponte della Paglia"}]	[{"name": "Rialto Bridge", "length": "157"}, {"name": "Bridge of Sighs", "length": "36"}, {"name": "Ponte della Paglia"}]	Venice
2023-04-26 21:10:45	Golden Gate Bridge		San Francisco

json_append(<json>, <path_value_pairs>)

This function appends values to the ends of indicated arrays within a JSON document. This function provides a JSON `eval` function equivalent to the multivalue `mvappend` function.

Usage

The `json_append` function always has at least three function inputs: `<json>` (the name of a valid JSON document such as a JSON object), and at least one `<path>` and `<value>` pair.

If `<json>` does not reference a valid JSON document, such as a JSON object, the function outputs nothing.

The `json_append` function evaluates `<path_value_pairs>` from left to right. When a path-value pair is evaluated, the function updates the `<json>` document. The function then evaluates the next path-value pair against the updated

document.

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Use <path> to designate a JSON document value

Each `<path>` designates an array or value within the `<json>` document. The `json_append` function adds the corresponding `<value>` to the end of the value designated by the `<path>`. The following table explains what `json_append` does depending on what the `<path>` specifies.

If <code><path></code> specifies...	...This is what <code>json_append</code> does with the corresponding <code><value></code>
An array with one or more values.	<code>json_append</code> adds the corresponding <code><value></code> to the end of that array.
An empty array	<code>json_append</code> adds the corresponding <code><value></code> to that array, creating an array with a single value.
A scalar or object value	<code>json_append</code> autowraps the scalar or object value within an array and adds the corresponding <code><value></code> to the end of that array.

The `json_append` function ignores path-value pairs for which the `<path>` does not identify any valid value in the JSON document.

Append arrays as single elements

When the new `<value>` is an array, `json_append` appends the array as a single element. For example, if a `json_array` `<path>` leads to the array `["a", "b", "c"]` and its `<value>` is the array `["d", "e", "f"]`, the result is `["a", "b", "c", ["d", "e", "f"]]`.

Appending arrays as single elements separates `json_append` from `json_extend`, a similar function that flattens arrays and objects into separate elements as it appends them. When `json_extend` takes the example in the preceding paragraph, it returns `["a", "b", "c", "d", "e", "f"]`.

Examples

The following examples show how you can use `json_append` to append values to arrays within a JSON document.

1. Add a string to an array

Say you have an object named `ponies` that contains an array named `ponylist: ["Minty", "Rarity", "Buttercup"]`. This is the search you would run to append "Fluttershy" to `ponylist`.

```
... | eval ponies = json_object("ponylist", json_array("Minty", "Rarity", "Buttercup")), updatePonies = json_append(ponies, "ponylist", "Fluttershy")
```

The output of that `eval` statement is `{"ponylist": ["Minty", "Rarity", "Buttercup", "Fluttershy"]}`.

2. Append a string to a nested object

This example has a `<path>` with the value `Fluttershy.ponySkills`. `Fluttershy.ponySkills` references an array of an object that is nested within `ponyDetails`, the source object. The query uses `json_append` to add a string to the nested object array.

```
... | eval ponyDetails = json_object("Fluttershy", json_object("ponySkills", json_array("running", "jumping"))), ponyDetailsUpdated = json_append(ponyDetails, "Fluttershy.ponySkills", "codebreaking")
```

The output of this `eval` statement is `ponyDetailsUpdated = {"Fluttershy": {"ponySkills": ["running", "jumping", "codebreaking"]}}`

json_array(<values>)

Creates a JSON array using a list of values.

Usage

A `<value>` can be any kind of value such as string, number, or Boolean. You can also use the `json_object` function to specify values.

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Examples

These examples show different ways to use the `json_array` function to create JSON arrays in your events.

1. Create a basic JSON array

The following example creates a simple array `["buttercup", "fluttershy", "rarity"]`.

```
... | eval ponies = json_array("buttercup", "fluttershy", "rarity")
```

2. Create an JSON array from a string and a JSON object

The following example uses a string `dubois` and the `json_object` function for the array values.

```
... | eval surname = json_array("dubois", json_object("name", "patel"))
```

The result is the JSON array `["dubois", {"name": "patel"}]`.

json_array_to_mv(<json_array>, <boolean>)

This function maps the elements of a proper JSON array into a multivalue field.

Usage

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

If the `<json_array>` input to the function is not a valid JSON array, the function outputs nothing.

Use the `<boolean>` input to specify that the `json_array_to_mv` function should preserve bracketing quotes on JSON-formatted strings. The `<boolean>` input defaults to `false()`.

Syntax	Description
--------	-------------

Syntax	Description
<code>json_array_to_mv(<json_array>, false()) or json_array_to_mv(<json_array>)</code>	By default (or when you explicitly set it to <code>false()</code>), the <code>json_array_to_mv</code> function removes bracketing quotes from JSON string data types when it converts an array into a multivalue field.
<code>json_array_to_mv(<json_array>, true())</code>	When set to <code>true()</code> , the <code>json_array_to_mv</code> function preserves bracketing quotes on JSON string data types when it converts an array into a multivalue field.

Example

This example demonstrates usage of the `json_array_to_mv` function to create simple multivalue fields out of JSON data.

The following example creates a simple array: `["Buttercup", "Fluttershy", "Rarity"]`. Then it maps that array into a multivalue field named `my_little_ponies` with the values `Buttercup`, `Fluttershy`, and `Rarity`. The function removes the quote characters when it converts the array elements into field values.

```
... | eval ponies = json_array("Buttercup", "Fluttershy", "Rarity"), my_sweet_ponies =  
json_array_to_mv(ponies)
```

If you change this search so it has `my_sweet_ponies = json_array_to_mv(ponies, true())`, you get an array with the values `"Buttercup"`, `"Fluttershy"`, and `"Rarity"`. Setting the function to `true` causes the function to preserve the quote characters when it converts the array elements into field values.

json_extend(<json>, <path_value_pairs>)

Use `json_extract` when you want to append multiple values at once to an array. `json_extend` flattens arrays into their component values and appends those values to the ends of indicated arrays within a valid JSON document.

Usage

The `json_extend` function always has at least three function inputs: `<json>` (the name of a valid JSON document such as a JSON object), and at least one `<path>` and `<value>` pair. The `<value>` must be an array. When given valid inputs, `json_extend` always outputs an array.

If `<json>` does not reference a valid JSON document, such as a JSON object, the function outputs nothing.

`json_extend` evaluates `<path_value_pairs>` from left to right. When `json_extend` evaluates a path-value pair, it updates the `<json>` document. `json_extend` then evaluates the next path-value pair against the updated document.

You can use `json_extend` with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Use `<path>` to designate a JSON document value

Each `<path>` designates an array or value within the `<json>` document. The `json_extend` function adds the values of the corresponding `<array>` after the last value of the array designated by the `<path>`. The following table explains what `json_extend` does depending on what the `<path>` specifies.

If <code><path></code> specifies...	...This is what <code>json_extend</code> does with the corresponding array values
An array with one or more values.	<code>json_extend</code> adds the corresponding array values to the end of that array.

If <path> specifies...	...This is what json_extend does with the corresponding array values
An empty array	json_extend adds the corresponding array values to that array.
A scalar or object value	json_extend autowraps the scalar or object value within an array and adds the corresponding array values to the end of that array.

json_extend ignores path-value pairs for which the <path> does not identify any valid value in the JSON document.

How json_extend flattens arrays before it appends them

The json_extend function flattens arrays as it appends them to the specified value. "Flattening" refers to the act of breaking the array down into its component values. For example, if a json_extend <path> leads to the array ["a", "b", "c"] and its <value> is the array ["d", "e", "f"], the result is ["a", "b", "c", "d", "e", "f"].

Appending arrays as individual values separates json_extend from json_append, a similar function that appends the <value> as a single element. When json_append takes the example in the preceding paragraph, it returns ["a", "b", "c", ["d", "e", "f"]].

Examples

The following examples show how you can use json_extend to append multiple values at once to arrays within a JSON document.

1. Extend an array with a set of string values

You start with an object named fakeBandsInMovies that contains an array named fakeMovieBandList: ["The Blues Brothers", "Spinal Tap", "Wyld Stallyns"]. This is the search you would run to extend that list with three more names of fake bands from movies.

```
... | eval fakeBandsInMovies = json_object("fakeMovieBandList", json_array("The Blues Brothers", "Spinal Tap", "Wyld Stallyns")), updateBandList = json_extend(fakeBandsInMovies, "fakeMovieBandList", json_array("The Soggy Bottom Boys", "The Weird Sisters", "The Barden Bellas"))
```

The output of this eval statement is {"fakeMovieBandList": ["The Blues Brothers", "Spinal Tap", "Wyld Stallyns", "The Soggy Bottom Boys", "The Weird Sisters", "The Barden Bellas"]}

2. Extend an array with an object

This example has an object named dndChars that contains an array named characterClasses. You want to update this array with an object from a secondary array. Here is a search you could run to achieve that goal.

```
... | eval dndChars = json_object("characterClasses", json_array("wizard", "rogue", "barbarian")), array2 = json_array(json_object("artifact", "deck of many things")), updatedParty = json_extend(dndChars, "characterClasses", array2)
```

The output of this eval statement is {updatedParty = ["wizard", "rogue", "barbarian", {"artifact": "deck of many things"}]}. Note that when json_extend flattens array2, it removes the object from the array. Otherwise the output would be {updatedParty = ["wizard", "rogue", "barbarian", [{"artifact": "deck of many things"}]]}.

json_extract(<json>, <paths>)

This function returns a value from a piece of JSON and zero or more paths. The value is returned in either a JSON array, or a Splunk software native type value.

If a JSON object contains a value with a special character, such as a period, `json_extract` can't access it. Use the `json_extract_exact` function for those situations.

See [json_extract_exact](#).

Usage

What is converted or extracted depends on whether you specify a piece of JSON, or JSON and one or more paths.

Syntax	Description
<code>json_extract(<json>)</code>	Converts a JSON field to the Splunk software native type. For example: <ul style="list-style-type: none">• Converts a JSON string to a string• Converts a JSON Boolean to a Boolean• Converts a JSON null to a null
<code>json_extract(<json>, <path>)</code>	Extracts the value specified by <code><path></code> from <code><json></code> , and converts the value to the native type. This can be a JSON array if the path leads to an array.
<code>json_extract(<json>, <path>, <path>, ...)</code>	Extracts all of the paths from <code><json></code> and returns it as a JSON array.

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Examples

These examples use this JSON object, which is in a field called `cities` in an event:

```
{
  "cities": [
    {
      "name": "London",
      "Bridges": [
        { "name": "Tower Bridge", "length": 801 },
        { "name": "Millennium Bridge", "length": 1066 }
      ]
    },
    {
      "name": "Venice",
      "Bridges": [
        { "name": "Rialto Bridge", "length": 157 },
        { "name": "Bridge of Sighs", "length": 36 },
        { "name": "Ponte della Paglia" }
      ]
    },
    {
      "name": "San Francisco",
      "Bridges": [
        { "name": "Golden Gate Bridge", "length": 8981 },
        { "name": "Bay Bridge", "length": 23556 }
      ]
    }
  ]
}
```

1. Extract the entire JSON object in a field

The following example returns the entire JSON object from the `cities` field. The `cities` field contains only one object. The key is the entire object. This extraction can return any type of value.

```
... | eval extracted_cities = json_extract(cities, "{}")
```

Here are the results of the search:

Field	Results
extract_cities	{"cities": [{"name": "London", "Bridges": [{"name": "Tower Bridge", "length": 801}, {"name": "Millennium Bridge", "length": 1066}], "name": "Venice", "Bridges": [{"name": "Rialto Bridge", "length": 157}, {"name": "Bridge of Sighs", "length": 36}, {"name": "Ponte della Paglia"}], "name": "San Francisco", "Bridges": [{"name": "Golden Gate Bridge", "length": 8981}, {"name": "Bay Bridge", "length": 23556}]}]

2. Extract the first nested JSON object in a field

The following example extracts the information about the city of London from the JSON object. This extraction can return any type of value.

The `{<num>}` indexing demonstrated in this example search only works when the `<path>` maps to a JSON array. In this case the `{0}` maps to the "0" item in the array, which is London. If the example used `{1}` it would select Venice from the array.

```
... | eval London=json_extract(cities, "{0}")
```

Here are the results of the search:

Field	Results
London	{"name": "London", "Bridges": [{"name": "Tower Bridge", "length": 801}, {"name": "Millennium Bridge", "length": 1066}]}

3. Extract the third nested JSON object in a field

The following example extracts the information about the city of San Francisco from the JSON object. This extraction can return any type of value.

```
... | eval San_Francisco=json_extract(cities, "{2}")
```

Here are the results of the search:

Field	Results
San_Francisco	{"name": "San Francisco", "Bridges": [{"name": "Golden Gate Bridge", "length": 8981}, {"name": "Bay Bridge", "length": 23556}]}]

4. Extract a specific key from each nested JSON object in a field

The following example extracts the names of the cities from the JSON object. This extraction can return any type of value.

```
... | eval my_cities=json_extract(cities, "{}.name")
```

Here are the results of the search:

Field	Results
my_cities	["London", "Venice", "San Francisco"]

5. Extract a specific set of key-value pairs from each nested JSON object in a field

The following example extracts the information about each bridge from every city from the JSON object. This extraction can return any type of value.

```
... | eval Bridges=json_extract(cities, "{}.Bridges{})")
```

Here are the results of the search:

Field	Results
Bridges	[{"name": "Tower Bridge", "length": 801}, {"name": "Millennium Bridge", "length": 1066}, {"name": "Rialto Bridge", "length": 157}, {"name": "Bridge of Sighs", "length": 36}, {"name": "Ponte della Paglia"}, {"name": "Golden Gate Bridge", "length": 8981}, {"name": "Bay Bridge", "length": 23556}]

6. Extract a specific value from each nested JSON object in a field

The following example extracts the names of the bridges from all of the cities from the JSON object. This extraction can return any type of value.

```
... | eval Bridge_names=json_extract(cities, "{}.Bridges{}.name")
```

Here are the results of the search:

Field	Results
Bridge_names	["Tower Bridge", "Millennium Bridge", "Rialto Bridge", "Bridge of Sighs", "Ponte della Paglia", "Golden Gate Bridge", "Bay Bridge"]

7. Extract a specific key-value pair from a specific nested JSON object in a field

The following example extracts the name and length of the first bridge from the third city from the JSON object. This extraction can return any type of value.

```
... | eval GG_Bridge=json_extract(cities, "{2}.Bridges{0}")
```

Here are the results of the search:

Field	Results
GG_Bridge	{"name": "Golden Gate Bridge", "length": 8981}

8. Extract a specific value from a specific nested JSON object in a field

The following example extracts the length of the first bridge from the third city from the JSON object. This extraction can return any type of value.

```
... | eval GG_Bridge_length=json_extract(cities, "{2}.Bridges{0}.length")
```

Here are the results of the search:

Field	Results
GG_Bridge_length	8981

json_extract_exact(<json>, <keys>)

Like the `json_extract` function, this function returns a Splunk software native type value from a piece of JSON. The main difference between these functions is that the `json_extract_exact` function does not use paths to locate and extract values, but instead matches literal strings in the event and extracts those strings as keys.

See [json_extract](#).

Usage

The `json_extract_exact` function treats strings for key extraction literally. This means that the function does not support explicitly nested paths. You can set paths with nested `json_array/json_object` function calls.

Syntax	Description
<code>json_extract_exact(<json>)</code>	Converts a JSON field to the Splunk software native type. For example: <ul style="list-style-type: none"> • Converts a JSON string to a string • Converts a JSON Boolean to a Boolean • Converts a JSON null to a null
<code>json_extract_exact(<json>, <string>)</code>	Extracts the key specified by <code><string></code> from <code><json></code> , and converts the key to the Splunk software native type. This can be a JSON array if the path leads to an array.
<code>json_extract_exact(<json>, <string>, <string>, ...)</code>	Extracts all of the strings from <code><json></code> and returns them as a JSON array of keys.

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Example

Suppose you have a JSON event that looks like this: `{"system.splunk.path":"/opt/splunk/"}`

If you want to extract `system.splunk.path` from that event, you can't use the `json_extract` function because of the period characters. Instead, you would use `json_extract_exact`, as shown in the following search:

```
... | eval extracted_path=json_extract_exact(splunk_path, "system.splunk.path")
```

json_keys(<json>)

Returns the keys from the key-value pairs in a JSON object. The keys are returned as a JSON array.

Usage

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

The `json_keys` function cannot be used on JSON arrays.

Examples

1. Return a list of keys from a JSON object

Consider the following JSON object, which is in the `bridges` field:

bridges
{"name": "Clifton Suspension Bridge", "length": 1352, "city": "Bristol", "country": "England"}

This example extracts the keys from the JSON object in the `bridges` field:

```
... | eval bridge_keys = json_keys(bridges)
```

Here are the results of the search:

bridge_keys
["name", "length", "city", "country"]

2. Return a list of keys from multiple JSON objects

Consider the following JSON objects, which are in separate rows in the `bridges` field:

bridges
{"name": "Clifton Suspension Bridge", "length": 1352, "city": "Bristol", "country": "England"}
{"name": "Rialto Bridge", "length": 157, "city": "Venice", "region": "Veneto", "country": "Italy"}
{"name": "Helix Bridge", "length": 918, "city": "Singapore", "country": "Singapore"}
{"name": "Tilikum Crossing", "length": 1700, "city": "Portland", "state": "Oregon", "country": "United States"}

This example extracts the keys from the JSON objects in the `bridges` field:

```
... | eval bridge_keys = json_keys(bridges)
```

Here are the results of the search:

bridge_keys
["name", "length", "city", "country"]
["name", "length", "city", "region", "country"]
["name", "length", "city", "country"]
["name", "length", "city", "state", "country"]

json_set(<json>, <path_value_pairs>)

Inserts or overwrites values for a JSON node with the values provided and returns an updated JSON object.

Similar to the `json_set_exact` function. See [json_set_exact](#)

Usage

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

- If the path contains a list of keys, all of the keys in the chain are created if the keys don't exist.
- If there's a mismatch between the JSON object and the path, the update is skipped and doesn't generate an error. For example, for object `{"a": "b"}`, `json_set(.., "a.c", "d")` produces no results since "a" has a string value and "a.c" implies a nested object.
- If the value already exists and is of a matching non-value type, the `json_set` function overwrites the value by default. A value type match isn't enforced. For example, you can overwrite a number with a string, Boolean, null, and so on.

Examples

These examples use this JSON object, which is in a field called `games` in an event:

```
{  
  "category": {  
    "boardgames": {  
      "cooperative": [  
        {  
          "name": "Pandemic"  
        },  
        {  
          "name": "Forbidden Island"  
        },  
        {  
          "name": "Castle Panic"  
        }  
      ]  
    }  
  }  
}
```

1. Overwrite a value in an existing JSON array

The following example overwrites the value "Castle Panic" in the path `[category.boardgames.cooperative]` in the JSON object. The value is replaced with `"name": "Sherlock Holmes: Consulting Detective"`. The results are placed into a new field called `my_games`.

The position count starts with 0. The third position is 2, which is why the example specifies `{2}` in the path.

```
... | eval my_games = json_set(games, "category.boardgames.cooperative{2}", "name": "Sherlock Holmes:  
Consulting Detective")
```

Here are the results of the search:

Field	Results
my_games	{"category":{"boardgames":{"cooperative":[{"name": "Pandemic", "name": "Forbidden Island", "name": "Sherlock Holmes: Consulting Detective"}]}}}

2. Insert a list of values in an existing JSON object

The following example inserts a list of popular games `["name": "Settlers of Catan", "name": "Terraforming Mars", "name": "Ticket to Ride"]` into the path `[category.boardgames.competitive]` in the JSON object.

Because the key `competitive` doesn't exist in the path, the key is created. The `json_array` function is used to append the value list to the `boardgames` JSON object.

```
... | eval my_games = json_set(games, "category.boardgames.competitive", json_array(json_object("name", "Settlers of Catan"), json_object("name", "Terraforming Mars"), json_object("name", "Ticket to Ride")))
```

Here are the results of the search:

Field	Results
my_games	{"category":{"boardgames":{"cooperative":[{"name": "Pandemic", "name": "Forbidden Island", "name": "Sherlock Holmes: Consulting Detective"}, {"name": "Settlers of Catan", "name": "Terraforming Mars", "name": "Ticket to Ride"}]}}, "competitive": [{}]}

The JSON object now looks like this:

```
{
  "category": {
    "boardgames": {
      "cooperative": [
        {
          "name": "Pandemic"
        },
        {
          "name": "Forbidden Island"
        },
        {
          "name": "Castle Panic"
        }
      ]
    },
    "competitive": [
      {
        "name": "Settlers of Catan"
      },
      {
        "name": "Terraforming Mars"
      },
      {
        "name": "Ticket to Ride"
      }
    ]
  }
}
```

3. Insert a set of key-value pairs in an existing JSON object

The following example inserts a set of key-value pairs that specify if the game is available using a Boolean value. These pairs are inserted into the path `[category.boardgames.competitive]` in the JSON object. The `json_array` function is used to append the key-value pairs list to the `boardgames` JSON object.

```
... | eval my_games = json_set(games, "category.boardgames.competitive{}.available", true())
```

Here are the results of the search:

Field	Results
my_games	{"category":{"boardgames":{"cooperative":[{"name":"Pandemic", "name":"Forbidden Island", "name":"Sherlock Holmes: Consulting Detective"}], "competitive": [{"name":"Settlers of Catan", "available":true, "name":"Terraforming Mars", "available":true}, {"name":"Ticket to Ride", "available":true}]}}}

The JSON object now looks like this:

```
{
  "category": {
    "boardgames": {
      "cooperative": [
        {
          "name": "Pandemic"
        },
        {
          "name": "Forbidden Island"
        },
        {
          "name": "Castle Panic"
        }
      ]
    },
    "competitive": [
      {
        "name": "Settlers of Catan",
        "available": true
      },
      {
        "name": "Terraforming Mars",
        "available": true
      },
      {
        "name": "Ticket to Ride",
        "available": true
      }
    ]
  }
}
```

If the Settlers of Catan game is out of stock, you can overwrite the value for the available key with the value false().

For example:

```
... | eval my_games = json_set(games, "category.boardgames.competitive{0}.available", false())
```

Here are the results of the search:

Field	Results
my_games	{"category":{"boardgames":{"cooperative":[{"name":"Pandemic", "name":"Forbidden Island", "name":"Sherlock Holmes: Consulting Detective"}], "competitive": [{"name":"Settlers of Catan", "available":false, "name":"Terraforming Mars", "available":true}, {"name":"Ticket to Ride", "available":true}]}}}

The JSON object now looks like this:

```
{
  "category": {
    "boardgames": {
      "cooperative": [
        {
          "name": "Pandemic"
        },
        {
          "name": "Forbidden Island"
        },
        {
          "name": "Castle Panic"
        }
      ]
    },
    "competitive": [
      {
        "name": "Settlers of Catan",
        "available": false
      },
      {
        "name": "Terraforming Mars",
        "available": true
      },
      {
        "name": "Ticket to Ride",
        "available": true
      }
    ]
  }
}
```

json_set_exact(<json>, <key_value_pairs>)

Generates or overwrites a JSON object using the key-value pairs that you specify.

Similar to the `json_set` function. See [json_set](#)

Usage

You can use the `json_set_exact` function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

- The `json_set_exact` function interprets the keys as literal strings, including special characters. This function does not interpret strings separated by period characters as keys for nested objects.
- If you supply multiple key-value pairs to `json_set_exact`, the function outputs an array.
- The `json_set_exact` function does not support or expect paths. You can set paths with nested `json_array` or `json_object` function calls.

Example

Suppose you want to have a JSON object that looks like this:

```
{"system.splunk.path":"/opt/splunk"}
```

To generate this object, you can use the `makeresults` command and the `json_set_exact` function as shown in the following search:

```
| makeresults | eval my_object=json_object(), splunk_path=json_set_exact(my_object, "system.splunk.path", "/opt/splunk")
```

You use `json_set_exact` for this instead of `json_set` because the `json_set` function interprets the period characters in `{"system.splunk.path"}` as nested objects. If you use `json_set` in the preceding search you get this JSON object:

```
{"system":{"splunk":{"path":"/opt/splunk"}}}
```

Instead of this object:

```
{"system.splunk.path":"/opt/splunk"}
```

json_valid(<json>)

Evaluates whether a piece of JSON uses valid JSON syntax and returns either TRUE or FALSE.

Usage

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Example

The following example validates a JSON object `{ "names": ["maria", "arun"] }` in the `firstnames` field.

Because fields cannot hold Boolean values, the `if` function is used with the `json_valid` function to place the string value equivalents of the Boolean values into the `isValid` field.

```
... | eval IsValid = if(json_valid(firstnames), "true", "false")
```

See also

Function information

[Evaluation functions](#) quick reference

Related functions

[mv_to_json_array](#) function

Related commands

[tojson](#)

[fromjson](#)

Mathematical functions

The following list contains the functions that you can use to perform mathematical calculations.

- For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).
- For the list of mathematical operators you can use with these functions, see "Operators" in the Usage section of the `eval` command.

abs(<num>)

Description

This function takes a number and returns its absolute value.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example creates a field called `absnum`, whose values are the absolute values of the numeric field `number`.

```
... | eval absnum=abs(number)
```

ceiling(<num>) or ceil(<num>)

Description

This function rounds a number up to the next highest integer.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

You can use the abbreviation `ceil` instead of the full name of the function.

Basic example

The following example returns n=2.

```
... | eval n=ceil(1.9)
```

exact(<expression>)

Description

This function renders the result of a numeric expression with a larger amount of precision in the formatted output.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=exact(3.14 * num)
```

exp(<num>)

Description

This function takes a number and returns the exponential function e^N .

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns $y=e^3$.

```
... | eval y=exp(3)
```

floor(<num>)

Description

This function rounds a number down to the nearest whole integer.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns 1.

```
... | eval n=floor(1.9)
```

ln(<num>)

Description

This function takes a number and returns the natural logarithm.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns the natural logarithm of the values of bytes.

```
... | eval lnBytes=ln(bytes)
```

log(<num>,<base>)

Description

This function takes either one or two numeric arguments and returns the logarithm of the first argument <num> using the second argument <base>. If the second argument <base> is omitted, this function evaluates the logarithm of number with base 10.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval num=log(number,2)
```

pi()

Description

This function takes no arguments and returns the constant *pi* to 11 digits of precision.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example calculates the area of a circle, which is `pi()` multiplied by the radius to the power of 2.

```
... | eval area_circle=pi()*pow(radius,2)
```

pow(<num>,<exp>)

Description

This function takes two numeric arguments <num> and <exp> and returns $<\text{num}>^{<\text{base}>}$, <num> to the power of <base>.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example calculates the area of a circle, which is pi() multiplied by the radius to the power of 2.

```
... | eval area_circle=pi()*pow(radius,2)
```

round(<num>,<precision>)

Description

This function takes one or two numeric arguments <num> and <precision>, returning <num> rounded up to the amount of decimal places specified by <precision>. The default is to round up to an integer.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

You cannot specify a negative number for the decimal places.

Basic examples

The following example returns n=4.

```
... | eval n=round(3.5)
```

The following example returns n=2.56.

```
... | eval n=round(2.555, 2)
```

sigfig(<num>)

Description

This function takes one argument, a number, and rounds that number to the appropriate number of significant figures.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The computation for `sigfig` is based on the type of calculation that generates the number.

- For multiplication and division, the result should have the minimum number of significant figures of all of the operands.
- For addition and subtraction, the result should have the same number of decimal places as the least precise number of all of the operands.

For example, the numbers 123.0 and 4.567 contain different precision with the decimal places. The first number is less precise because it has 1 decimal place. The second number is more precise because it has 3 decimal places.

If the calculation is $123.0 + 4.567 = 127.567$, then the `sigfig` function returns the fewest number of decimal places. In this example only one decimal place is returned. Because the numbers to the right of the last significant figure are greater than 5, the result returned is 127.6

Basic examples

Example 1: The following example shows how the `sigfig` function works. The calculation `1.00*1111` returns the value `n=1111`, but the following search using the `sigfig` function returns `n=1110`.

```
... | eval n=sigfig(1.00*1111)
```

In this example, 1.00 has 3 significant figures and 1111 has 4 significant figures. In this example, the minimum number of significant figures for all operands is 3. Using the `sigfig` function, the final result is rounded to 3 digits, returning `n=1110` and not 1111.

Example 2: There are situations where the results of a calculation can return a different accuracy to the very far right of the decimal point. For example, the following search calculates the average of 100 values:

```
| makeresults count=100 | eval test=3.99 | stats avg(test)
```

The result of this calculation is:

avg(test)
3.9900000000000055

When the count is changed to 10000, the results are different:

```
| makeresults count=10000 | eval test=3.99 | stats avg(test)
```

The result of this calculation is:

avg(test)
3.990000000000215

This occurs because numbers are treated as double-precision floating-point numbers.

To mitigate this issue, you can use the `sigfig` function to specify the number of significant figures you want returned.

However, first you need to make a change to the `stats` command portion of the search. You need to change the name of the field `avg(test)` to remove the parenthesis. For example `stats avg(test) AS test`. The `sigfig` function expects either a number or a field name for X. The `sigfig` function cannot accept a field name that looks like another function, in this case `avg`.

To specify the number of decimal places you want returned, you multiply the field name by 1 and use zeros to specify the number of decimal places. If you want 4 decimal places returned, you would multiply the field name by 1.0000. To return 2 decimal places, multiply by 1.00, as shown in the following example:

```
| makeresults count=10000 | eval test=3.99 | stats avg(test) AS test | eval new_test=sigfig(test*1.00)
```

The result of this calculation is:

test
3.99
sqrt(<num>)

Description

This function takes one numeric argument <num> and returns its square root.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns 3:

```
... | eval n=sqrt(9)
```

sum(<num>,...)

Description

This function takes an arbitrary number of arguments and returns the sum of numerical values as an integer. Each argument must be either a field (single or multi value) or an expression that evaluates to a number. At least one numeric argument is required. When the function is applied to a multivalue field, each numeric value of the field is included in the total. The `eval` command ignores arguments that don't exist in an event or can't be converted to a number.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

Example 1: The following example creates a field called `a` with value 5.0, a field called `b` with value 9, and a field called `x` with value 14 that is the sum of `a` and `b`. A field is not created for `c` and it is not included in the sum because a value was not declared for that argument.

```
... | eval a = 5.0, b = "9", x = sum(a, b, c)
```

Example 2: The following example calculates the sum of three numbers and returns `c=6`.

```
... | eval c=sum(1, 2, 3)
```

However, the following example returns an error because one of the arguments in the function is a string.

```
... | eval c=sum(1, 2, "3")
```

To use a quoted string as a number within the function, you must convert the number to an integer, as shown in the following example that returns c=6.

```
... | eval c=sum(1, 2, tonumber("3"))
```

Example 3: In this example, a field with a value that is a string results in a field called `a` with value 1, and a field called `c` with value 6,

```
... | eval a="1", c=sum(a, 2, 3)
```

Example 4: When an argument is a field, the `eval` command retrieves the value and attempts to treat it as a number, even if it is a string. The following example creates a field called `a` with value `somedata`, and a field called `c` with value 5.

```
... | eval a="somedata", c=sum(a, 2, 3)
```

However, the following example returns an error because the string argument is specified directly within the function.

```
... | eval c=sum("somedata", 2, 3)
```

Multivalue eval functions

The following list contains the functions that you can use on multivalue fields or to return multivalue fields.

You can also use the statistical eval functions, `max` and `min`, on multivalue fields. See [Statistical eval functions](#).

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

commands(<value>)

Description

This function takes a search string, or field that contains a search string, and returns a multivalued field containing a list of the commands used in <value>.

Usage

This function is generally not recommended for use except for analysis of `audit.log` events.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns a multivalued field called `x`, that contains the commands `search`, `stats`, and `sort` which are the commands used in the search string specified.

```
... | eval x=commands("search foo | stats count | sort count")
```

mvappend(<values>)

Description

This function takes one or more arguments and returns a single multivalue result that contains all of the values. The arguments can be strings, multivalue fields or single value fields.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

This example shows how to append two values, `localhost` is a literal string value and `srcip` is a field name.

```
... | eval fullName=mvappend("localhost", srcip)
```

The following example shows how to use nested `mvappend` functions.

- The inner `mvappend` function contains two values: `localhost` is a literal string value and `srcip` is a field name.
- The outer `mvappend` function contains three values: the inner `mvappend` function, `destip` is a field name, and `192.168.1.1` which is a literal IP address.

The results are placed in a new field called `ipaddresses`, which contains the array `["localhost", <values_in_srcip>, <values_in_destip>, "192.168.1.1"]`.

```
... | eval ipaddresses=mvappend(mvappend("localhost", srcip), destip, "192.168.1.1")
```

Note that the previous example generates the same results as the following example, which does not use a nested `mvappend` function:

```
| makeresults | eval ipaddresses=mvappend("localhost", srcip, destip, "192.168.1.1")
```

The results look something like this:

time	ipaddresses
2020-11-19 16:43:31	localhost 192.168.1.1

mvcount(<mv>)

Description

This function takes a field and returns a count of the values in that field for each result. If the field is a multivalue field, returns the number of values in that field. If the field contains a single value, this function returns 1 . If the field has no values, this function returns NULL.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=mvcount(multifield)
```

Extended example

In the following example, the `mvcount()` function returns the number of email addresses in the `To`, `From`, and `CC` fields and saves the addresses in the specified "`_count`" fields.

```
eventtype="sendmail" | eval To_count=mvcount(split(To,"@"))-1 | eval From_count=mvcount(From) | eval  
Cc_count= mvcount(split(Cc,"@"))-1
```

This search takes the values in the `To` field and uses the `split` function to separate the email address on the @ symbol. The `split` function is also used on the `CC` field for the same purpose.

If only a single email address exists in the `From` field, as you would expect, `mvcount(From)` returns 1. If there is no `CC` address, the `CC` field might not exist for the event. In that situation `mvcount(CC)` returns NULL.

mvdedup(<mv>)

Description

This function takes a multivalue field and returns a multivalue field with its duplicate values removed.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval s=mvdedup(mvfield)
```

mvfilter(<predicate>)

Description

This function filters a multivalue field based on an arbitrary Boolean expression. The Boolean expression can reference ONLY ONE field at a time.

Usage

This function will return NULL values of the field as well. If you do not want the NULL values, use one of the following expressions:

- `mvfilter(!isnull(<value>))`
- `mvfilter(isnotnull(<value>))`

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

The following example returns all of the values in field `email` that end in `.net` or `.org`.

```
... | eval n=mvfilter(match(email, "\\.net$") OR match(email, "\\.org$"))
```

mvfind(<mv>,<regex>)

Description

This function tries to find a value in the multivalue field that matches the regular expression. If a match exists, the index of the first matching value is returned (beginning with zero). If no values match, NULL is returned.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

```
... | eval n=mvfind(mymvfield, "err\d+")
```

mvindex(<mv>,<start>,<end>)

Description

This function returns a subset of the multivalue field using the start and end index values.

Usage

The `<mv>` argument must be a multivalue field. The `<start>` and `<end>` indexes must be numbers.

The `<mv>` and `<start>` arguments are required. The `<end>` argument is optional.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Specifying the start and end indexes

- Indexes start at zero. If you have 5 values in the multivalue field, the first value has an index of 0. The second value has an index of 1, and so on.
- If only the `<start>` argument is specified, only that value is included in the results.
- When the `<end>` argument is specified, the range of values from `<start>` to `<end>` are included in the results.
- Both the `<start>` and `<end>` arguments can be negative. An index of `-1` is used to specify the last value in the list.
- If the indexes are out of range or invalid, the result is NULL.

Examples

Consider the following values in a multivalue field called `names`:

Name	alex	celestino	claudia	david	ikraam	nyah	rutherford	wei
index number	0	1	2	3	4	5	6	7

Because indexes start at zero, the following example returns the value `claudia`:

```
... | eval my_names=mvindex(names,2)
```

To return a range of values, specify both a `<start>` and `<end>` value. For example, the following search returns the first 4 values in the field. The start value is `0` and the end value is `3`.

```
... | eval my_names=mvindex(names,0,3)
```

The results look like this:

my_names
alex,celestino,claudia,david

Extended examples

Consider the following values in a multivalue field:

ponies
buttercup, dash, flutter, honey, ivory, minty, pinky, rarity

To return a value from the end of the list of values, the index numbers start with `-1`. The negative symbol indicates that the indexing starts from the last value. For example:

Pony name	buttercup	dash	flutter	honey	ivory	minty	pinky	rarity
index number	-8	-7	-6	-5	-4	-3	-2	-1

To return the last value in the list, you specify `-1`, which indicates to start at the end of the list and return only one value. For example:

```
... | eval my_ponies=mvindex(ponies,-1)
```

The results look like this:

my_ponies
rarity

To return the 3rd value from the end, you would specify the index number `-3`. For example:

```
... | eval my_ponies=mvindex(ponies,-3)
```

The results look like this:

my_ponies
minty

To return a range of values, specify both a `<start>` and `<end>` value. For example, the following search returns the last 3 values in the field. The start value is `-3` and the end value is `-1`.

```
... | eval my_ponies=mvindex(ponies, -3, -1)
```

The results look like this:

my_ponies
minty,pinky,rarity

mvjoin(<mv>,<delim>)

Description

This function takes two arguments, a multivalue field and a string delimiter. The function concatenates the individual values within <mv> using the value of <delim> as a separator.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

You have a multivalue field called "base" that contains the values "1" "2" "3" "4" "5". The values are separated by a space. You want to create a single value field instead, with OR as the delimiter. For example "1 OR 2 OR 3 OR 4 OR 5".

The following search creates the `base` field with the values. The search then creates the `joined` field by using the result of the `mvjoin` function.

```
... | eval base=mvrange(1,6), joined=mvjoin('base', " OR ")
```

The following example joins together the individual values of "myfield" using a semicolon as the delimiter:

```
... | eval n=mvjoin(myfield, ";")
```

mvmap(<mv>,<expression>)

Description

This function iterates over the values of a multivalue field, performs an operation using the <expression> on each value, and returns a multivalue field with the list of results.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example multiplies each value in the `results` field by 10.

```
... | eval n=mvmap(results, results*10)
```

The following example multiplies each value in the `results` field by `threshold`, where `threshold` is a single-valued field.

```
... | eval n=mvmap(results, results*threshold)
```

The following example multiplies the 2nd and 3rd values in the `results` field by `threshold`, where `threshold` is a single-valued field. This example uses the `mvindex` function to identify specific values in the `results` field.

```
... | eval n=mvmap(mvindex(results, 1,2), results*threshold)
```

mvrangle(<start>,<end>,<step>)

Description

This function creates a multivalue field for a range of numbers. This function can contain up to three arguments: a starting number, an ending number (which is excluded from the field), and an optional step increment. If the increment is a timespan such as `7d`, the starting and ending numbers are treated as UNIX time.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The step increment is optional. If the `<step>` increment is a timespan such as `7d`, the starting and ending numbers are treated as UNIX time.

The `<end>` number is not included from the multivalue field that is created.

Basic examples

The following example returns a multivalue field with the values 1, 3, 5, 7, 9.

```
... | eval mv=mvrangle(1,11,2)
```

The following example takes the UNIX timestamp for 1/1/2018 as the start date and the UNIX timestamp for 4/19/2018 as an end date and uses the increment of 7 days.

```
| makeresults | eval mv=mvrangle(1514834731,1524134919,"7d")
```

This example returns a multivalue field with the UNIX timestamps. The results appear on the Statistics tab and look something like this:

_time	mv
2018-04-10 12:31:03	1514834731
	1515439531
	1516044331
	1516649131
	1517253931
	1517858731
	1518463531
	1519068331
	1519673131
	1520277931
	1520879131
	1521483931
	1522088731

_time	mv
	1522693531
	1523298331
	1523903131

mvsort(<mv>)

Description

This function uses a multivalue field and returns a multivalue field with the values sorted lexicographically.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Basic example

```
... | eval s=mvsort(mvfield)
```

mvzip(<mv_left>,<mv_right>,<delim>)

Description

This function combines the values in two multivalue fields. The delimiter is used to specify a delimiting character to join the two values.

Usage

This is similar to the Python `zip` command.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The values are stitched together combining the first value of `<mv_left>` with the first value of field `<mv_right>`, then the second with the second, and so on.

The delimiter is optional, but when specified must be enclosed in quotation marks. The default delimiter is a comma (,).

Basic example

```
... | eval nserver=mvzip(hosts,ports)
```

Extended example

You can nest several `mvzip` functions together to create a single multivalued field `three_fields` from three separate fields. The pipe (|) character is used as the separator between the field values.

```
... | eval three_fields=mvzip(mvzip(field1,field2,"|"),field3,"|")
```

(Thanks to Splunk user cmerriman for this example.)

mv_to_json_array(<field>, <infer_types>)

This function maps the elements of a multivalue field to a JSON array.

Usage

You can use this function with the `eval` and `where` commands, and as part of evaluation expressions with other commands.

Because the elements of JSON arrays can have many data types (such as string, numeric, Boolean, and null), the `mv_to_json_array` function lets you specify how it should map the contents of multivalue fields into JSON arrays. You can have the field values simply written to arrays as string data types, or you can have the function infer different JSON data types.

Use the `<infer_types>` input to specify that the `mv_to_json_array` function should attempt to infer JSON data types when it converts field values into array elements. The `<infer_types>` input defaults to `false`.

Syntax	Description
<code>mv_to_json_array(<field>, false())</code> or <code>mv_to_json_array(<field>)</code>	By default, or when you explicitly set it to <code>false()</code> , the <code>mv_to_json_array</code> function maps all values in the multivalued field to the JSON array as string data types, whether they are numeric, strings, Boolean values, or any other JSON data type. The <code>mv_to_json_array</code> function effectively splits the multivalue field on the comma and writes each quote-enclosed value to the array as an element with the string data type.
<code>mv_to_json_array(<field>, true())</code>	When you set the <code>mv_to_json_array</code> function to <code>true()</code> , the function removes one set of bracketing quote characters from each value it transfers into the JSON array. If the function does not recognize the resulting array element as a proper JSON data type (such as string, numeric, Boolean, or null), the function turns the element into a null data type.

Example

This example shows you how the `mv_to_json_array` function can validate JSON as it generates JSON arrays.

This search creates a multivalue field named `ponies`.

```
... | eval ponies = mvappend("\\"Buttercup\\\"", "\\"Fluttershy\\\"", "\\"Rarity\\\"", "true", "null"),
```

The array that is created from these values depends on the `<infer_types>` input.

Without inferring data types

When `<infer_types>` is set to `false` or omitted, the `mv_to_json_array` function converts the field values into array elements without changing the values.

```
... | eval my_sweet_ponies = mv_to_json_array(ponies, false())
```

The resulting array looks like this:

```
["\"Buttercup\"", "\"Fluttershy\"", "\"Rarity\"", "true", "null"]
```

With inferring data types

When you run this search with `infer_values` set to `true()`, the `mv_to_json_array` function removes the extra quote and backslash escape characters from the field values when the values are converted into array elements.

```
... | eval my_sweet_ponies = mv_to_json_array(ponies, true())
```

The resulting array looks like this:

```
["Buttercup", "Fluttershy", "Rarity", true, null]
```

split(<str>,<delim>)

Description

This function splits the string values on the delimiter and returns the string values as a multivalue field.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Use an empty string ("") to split the original string into one value per character. For example, the following search splits the string into `a`, `b`, `c`, and `d`.

```
| makeresults | eval test="abcd" | eval results=split(test,"")
```

Basic example

To illustrate how the `split` function works, the following search creates an event with a `test` field that contains a list of string values separated by semicolon characters (;).

```
| makeresults | eval test="buttercup;rarity;tenderhoof;dash;mcintosh;fleetfoot;mistmane"
```

The results look like this:

_time	test
2022-09-20 17:39:56	buttercup;rarity;tenderhoof;dash;mcintosh;fleetfoot;mistmane

To split up each of the names in the event into a multivalue field using the semicolon delimiter, you could run a search like this:

```
| makeresults | eval test="buttercup;rarity;tenderhoof;dash;mcintosh;fleetfoot;mistmane" | eval ponies=split(test,";")
```

Now each of the pony names in the `test` event is a field in a multivalue field. The results look something like this:

_time	ponies	test
2022-09-20 18:22:03	buttercup rarity tenderhoof dash mcintosh fleetfoot mistmane	buttercup;rarity;tenderhoof;dash;mcintosh;fleetfoot;mistmane

You can also use a string of contiguous characters in your search like this, which splits the string on "def".

```
|makeresults |eval test="1a2b3c4def567890" |eval results=split(test,"def")
```

The results look something like this.

_time	results	test
2023-01-23 12:18:11	1a2b3c4 567890	1a2b3c4def567890

Extended example

The following search is useful for building equivalents to string functions like Oracle INSTR.

```
| makeresults |eval test="name::value" |eval results=split(test,"::")
```

The results look something like this. The length of the first entry (mvindex=0) is the position of the "::" string, plus or minus one.

_time	results	test
2023-01-23 12:18:11	name value	name::value

See also

See the following multivalue commands:

[makemv](#), [mvcombine](#), [mvexpand](#), [nomv](#)

Statistical eval functions

The following list contains the evaluation functions that you can use to calculate statistics.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

In addition to these functions, there is a comprehensive set of [statistical functions](#) that you can use with the `stats`, `chart`, and related commands.

avg(<values>)

Description

This function takes one or more values and returns the average of numerical values as an integer. Each argument must be either a field (single or multivalue) or an expression that evaluates to a number. At least one numeric argument is required. When the function is applied to a multivalue field, each numeric value of the field is included in the total. The `eval` command ignores arguments that don't exist in an event or can't be converted to a number.

To get the numerical average or mean of the values of two fields, `x` and `y`, note that `avg(x, y)` is equivalent to `sum(x, y) / (mvcount(x) + mvcount(y))`.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

Example 1: The following example creates a field called `a` with value 5.0, a field called `b` with value 9, and a field called `x` with value 7 that is the average of `a` and `b`. A field is not created for `c` and it is not included in the total because a value was not declared for that argument.

```
... | eval a = 5.0, b = "9", x = avg(a, b, c)
```

Example 2: The following example calculates the average of three numbers and returns `c=2`.

```
... | eval c=avg(1, 2, 3)
```

However, the following example returns an error because one of the arguments in the function is a string.

```
... | eval c=avg(1, 2, "3")
```

To use a quoted string as a number within the function, you must convert the number to an integer, as shown in the following example where `c=2`:

```
... | eval c=avg(1, 2, tonumber("3"))
```

Example 3: In this example, a field with a value that is a string results in a field called `a` with value 1, and a field called `c` with value 2,

```
... | eval a="1", c=avg(a, 2, 3)
```

Example 4: When an argument is a field, the `eval` command retrieves the value and attempts to treat it as a number, even if it is a string. The following example creates a field called `a` with value `somedata`, and a field called `c` with value 2.5.

```
... | eval a="somedata", c=avg(a, 2, 3)
```

However, the following example returns an error because the string argument is specified directly within the function.

```
... | eval c=avg("somedata", 2, 3)
```

max(<values>)

Description

This function takes one or more numeric or string values, and returns the maximum. Strings are greater than numbers.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns either "foo" or the value in the `name` field. Splunk searches use lexicographical order, where numbers are sorted before letters. If the value in the `name` field is "baz", then "foo" is returned. If the value in the `name` field is "zaz", then "zaz" is returned.

```
... | eval n=max(1, 3, 6, 7, "foo", name)
```

The following example returns the maximum value in a multivalue field.

This search creates a field called `n` with a single value, which is a series of numbers. The `makemv` command is used to make the single value into multiple values, each of which appears on its own row in the results. Another new field called `maxn` is created which takes the values in `n` and returns the maximum value, 6.

```
| makeresults | eval n = "1 3 5 6 4 2" | makemv n | eval maxn = max(n)
```

The results look like this:

_time	maxn	n
2021-01-29 10:42:37	6	1 3 5 6 4 2

min(<values>)

Description

This function takes one or more numeric or string values, and returns the minimum. Strings are greater than numbers.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns either 3 or the value in the `size` field. Splunk searches use lexicographical order, where numbers are sorted before letters. If the value in the `size` field is 9, then 3 is returned. If the value in the `size` field is 1,

then 1 is returned.

```
... | eval n=min(3, 6, 7, "maria", size)
```

The following example returns the minimum value in a multivalue field.

This search creates a field called `n` with a single value, which is a series of numbers. The `makemv` command is used to make the single value into multiple values, each of which appears on its own row in the results. Another new field called `minn` is created which takes the values in `n` and returns the minimum value, 2.

```
| makeresults | eval n = "3 5 6 4 7 2" | makemv n | eval minn = min(n)
```

The results look like this:

_time	minn	n
2021-01-29 10:42:37	2	3 5 6 4 7 2

random()

Description

This function takes no arguments and returns a pseudo-random integer ranging from zero to $2^{31}-1$.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

The following example returns a random integer, such as 0...2147483647.

```
... | eval n=random()
```

The following example returns a random number within a specified range. In this example, the random number is between 1 and 100,000.

```
... | eval n=(random() % 100000) + 1
```

This example takes a random number and uses the modulo mathematical operator (`%`) to divide the random number by 100000. This ensures that the random number returned is not greater than 100000. The number remaining after the division is increased by 1 to ensure that the number is at least greater than or equal to 1.

Text functions

The following list contains the functions that you can use with string values.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

len(<str>)

Description

This function returns the character length of a string.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

This function is not supported on multivalue fields.

Basic example

Suppose you have a set of results that looks something like this:

_time	names
2020-01-09 16:35:14	buttercup
2020-01-09 16:35:14	rarity
2020-01-09 16:35:14	tenderhoof
2020-01-09 16:35:14	dash
2020-01-09 16:35:14	mistmane

You can determine the length of the values in the `names` field using the `len` function:

```
... | eval length=len(names)
```

The results show a count of the character length of the values in the `names` field:

_time	length	names
2020-01-09 16:35:14	9	buttercup
2020-01-09 16:35:14	6	rarity
2020-01-09 16:35:14	10	tenderhoof
2020-01-09 16:35:14	4	dash
2020-01-09 16:35:14	8	mistmane

lower(<str>)

Description

This function takes one string argument and returns the string in lowercase.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

You can use this function on multivalue fields.

Basic example

The following example returns the value provided by the field `username` in lowercase.

```
... | eval username=lower(username)
```

ltrim(<str>,<trim_chars>)

Description

This function removes characters from the left side of a string.

Usage

The `<str>` argument can be the name of a string field or a string literal.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `<trim_chars>` argument is optional. If not specified, spaces and tabs are removed from the left side of the string.

This function is not supported on multivalue fields.

Basic example

The following example trims the leading spaces and all of the occurrences of the letter Z from the left side of the string. The value that is returned is `x="abcZZ"`.

```
... | eval x=ltrim(" ZZZZabcZZ ", " Z")
```

The following example removes the dollar sign (`$`) from the results for the `NET_COST` field.

```
... | eval cost=ltrim(NET_COST, "$")
```

replace(<str>,<regex>,<replacement>)

Description

This function substitutes the replacement string for every occurrence of the regular expression in the string.

Usage

The <str> argument can be the name of a string field or a string literal.

The <replacement> argument can also reference groups that are matched in the <regex> using perl-compatible regular expressions (PCRE) syntax.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

This function is not supported on multivalue fields.

To replace a backslash (\) character, you must escape the backslash twice. This is because the `replace` function occurs inside an eval expression. The eval expression performs one level of escaping before passing the regular expression to PCRE. Then PCRE performs its own escaping. See SPL and regular expressions.

Basic example

The following example returns the values in the `date` field, with the month and day numbers switched. If the input is 1/14/2023 the return value would be 14/1/2023.

```
... | eval n=replace(date, "^(\\d{1,2})/(\\d{1,2})/", "\\\2\\\\1/")
```

rtrim(<str>,<trim_chars>)

Description

This function removes the trim characters from the right side of the string.

Usage

The <str> argument can be the name of a string field or a string literal.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The <trim_chars> argument is optional. If not specified, spaces and tabs are removed from the right side of the string.

This function is not supported on multivalue fields.

Basic example

The following example trims the leading spaces and all of the occurrences of the letter Z from the right side of the string. The value returned is `zzzzabc`.

```
... | eval n=rtrim(" zzzzabczz ", " z")
```

spath(<value>,<path>)

Description

Use this function to extract information from the structured data formats XML and JSON.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

The `<value>` is an input source field.

The `<path>` is an spath expression for the location path to the value that you want to extract from.

- If `<path>` is a literal string, you need to enclose the string in double quotation marks.
- If `<path>` is a field name, with values that are the location paths, the field name doesn't need quotation marks. Using a field name for `<path>` might result in a multivalue field.

This function is not supported on multivalue fields.

Basic example

The following example returns the values of locDesc elements.

```
... | eval locDesc=spath(_raw, "vendorProductSet.product.desc.locDesc")
```

The following example returns the hashtags from a twitter event.

```
index=twitter | eval output=spath(_raw, "entities.hashtags")
```

substr(<str>,<start>,<length>)

Description

This function returns a substring of a string, beginning at the start index. The length of the substring specifies the number of character to return.

Usage

The `<str>` argument can be the name of a string field or a string literal.

The indexes follow SQLite semantics; they start at 1. Negative indexes can be used to indicate a start from the end of the string.

The `<length>` is optional, and if not specified returns the rest of the string.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

This function is not supported on multivalue fields.

Basic example

The following example concatenates "str" and "ing" together, returning "string":

```
... | eval n=substr("string", 1, 3) + substr("string", -3)
```

trim(<str>,<trim_chars>)

Description

This function removes the trim characters from both sides of the string.

Usage

The <str> argument can be the name of a string field or a string literal.

The <trim_chars> argument is optional. If not specified, spaces and tabs are removed from both sides of the string.

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

This function is not supported on multivalue fields.

Basic example

The following example trims the leading spaces and all of the occurrences of the letter Z from the left and right sides of the string. The value returned is `abc`.

```
... | eval n=trim(" ZZZZabcZZ ", " Z")
```

upper(<str>)

Description

This function returns a string in uppercase.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

You can use this function on multivalue fields.

Basic example

The following example returns the value provided by the field `username` in uppercase.

```
... | eval n=upper(username)
```

urldecode(<url>)

Description

This function takes one URL string argument X and returns the unescaped or decoded URL string.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

This function is not supported on multivalue fields.

Basic example

The following example returns "http://www.splunk.com/download?r=header".

```
... | eval n=urldecode("http%3A%2F%2Fwww.splunk.com%2Fdownload%3Fr%3Dheader")
```

Trig and Hyperbolic functions

The following list contains the functions that you can use to calculate trigonometric and hyperbolic values.

For information about using string and numeric fields in functions, and nesting functions, see [Evaluation functions](#).

acos(X)

Description

This function computes the arc cosine of X, in the interval [0,pi] radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

This example returns 1.5707963267948966.

```
... | eval n=acos(0)
```

The following example calculates 180 divided by *pi* and multiplies the result by the arc cosine of 0. This example returns 90.0000000003.

```
... | eval degrees=acos(0)*180/pi()
```

acosh(X)

Description

This function computes the arc hyperbolic cosine of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 1.3169578969248166.

```
... | eval n=acosh(2)
```

asin(X)

Description

This function computes the arc sine of X, in the interval [-pi/2,+pi/2] radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 1.5707963267948966.

```
... | eval n=asin(1)
```

The following example calculates 180 divided by *pi* and multiplies that by the arc sine of 1.

```
... | eval degrees=asin(1)*180/pi()
```

asinh(X)

Description

This function computes the arc hyperbolic sine of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 0.881373587019543.

```
... | eval n=asinh(1)
```

atan(X)

Description

This function computes the arc tangent of X, in the interval [-pi/2,+pi/2] radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 0.46.

```
... | eval n=atan(0.50)
```

atan2(Y, X)

Description

This function computes the arc tangent of Y, X in the interval [-pi,+pi] radians.

Y is a value that represents the proportion of the y-coordinate. X is the value that represents the proportion of the x-coordinate.

To compute the value, the function takes into account the sign of both arguments to determine the quadrant.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 0.59.

```
... | eval n=atan2(0.50, 0.75)
```

atanh(X)

Description

This function computes the arc hyperbolic tangent of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 0.549.

```
... | eval n=atanh(0.500)
```

cos(X)

Description

This function computes the cosine of an angle of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

This example returns 0.5403023058681398.

```
... | eval n=cos(-1)
```

The following example calculates the cosine of π and returns -1.00000000000.

```
... | eval n=cos(pi())
```

cosh(X)

Description

This function computes the hyperbolic cosine of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns 1.5430806348152437.

```
... | eval n=cosh(1)
```

hypot(X,Y)

Description

This function computes the hypotenuse of a right-angled triangle whose legs are X and Y.

The function returns the square root of the sum of the squares of X and Y, as described in the Pythagorean theorem.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

Creates a field called `n` and returns `n=5`, which is the hypotenuse of a triangle whose legs are 3 and 4.

```
... | eval n=hypot(3,4)
```

sin(X)

Description

This function computes the sine of X.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

This example returns 0.8414709848078965.

```
... | eval n=sin(1)
```

The following example calculates the sine of π divided by 180 and then multiplied by 90.

```
... | eval n=sin(90 * pi()/180)
```

sinh(X)

Description

This function computes the hyperbolic sine of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns `1.1752011936438014`.

```
... | eval n=sinh(1)
```

tan(X)

Description

This function computes the tangent of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic examples

This example returns `1.5574077246549023`

```
... | eval n=tan(1)
```

This example returns `-0.08871575677006045`

```
... | eval n=tan(135)
```

tanh(X)

Description

This function computes the hyperbolic tangent of X radians.

Usage

You can use this function with the `eval`, `fieldformat`, and `where` commands, and as part of eval expressions.

Basic example

This example returns `0.7615941559557649`

```
... | eval n=tanh(1)
```

Statistical and Charting Functions

Statistical and charting functions

You can use the statistical and charting functions with the `chart`, `stats`, and `timechart` commands.

Support for related commands

The functions can also be used with related statistical and charting commands. The following table lists the commands supported by the statistical and charting functions and the related command that can also use these functions.

Command	Supported related commands
chart	<ul style="list-style-type: none"><code>sichart</code>
stats	<ul style="list-style-type: none"><code>eventstats</code><code>streamstats</code><code>geostats</code><code>sistats</code>For the <code>tstats</code> and the <code>mstats</code> commands, see the documentation for each command for a list of the supported functions.
timechart	<ul style="list-style-type: none"><code>sitimechart</code>

Functions that you can use to create sparkline charts are noted in the documentation for each function. Sparkline is a function that applies to only the `chart` and `stats` commands, and allows you to call other functions. For more information, see Add sparklines to search results in the *Search Manual*.

How field values are processed

Most of the statistical and charting functions expect the field values to be numbers. All of the values are processed as numbers, and any non-numeric values are ignored.

The following functions process the field values as literal string values, even though the values are numbers.

<ul style="list-style-type: none"><code>count</code><code>distinct</code><code>earliest</code>	<ul style="list-style-type: none"><code>estdc</code><code>estdc_count</code><code>first</code>	<ul style="list-style-type: none"><code>error</code>	<ul style="list-style-type: none"><code>latest</code><code>last</code><code>list</code>	<ul style="list-style-type: none"><code>max</code><code>min</code><code>mode</code><code>values</code>
--	--	--	---	---

For example, you use the `distinct_count` function and the field contains values such as "1", "1.0", and "01". Each value is considered a distinct string value.

The only exceptions are the `max` and `min` functions. These functions process values as numbers if possible. For example, the values "1", "1.0", and "01" are processed as the same numeric value.

Supported functions and syntax

There are two ways that you can see information about the supported statistical and charting functions:

- [Function list by category](#)
- [Alphabetical list of functions](#)

Function list by category

The following table is a quick reference of the supported statistical and charting functions, organized by category. This table provides a brief description for each function. Use the links in the table to learn more about each function and to see examples.

Type of function	Supported functions and syntax	Description
Aggregate functions	<code>avg(<value>)</code>	Returns the average of the values in the field specified.
	<code>count(<value>)</code>	Returns the number of occurrences where the field that you specify contains any value (is not empty). You can also count the occurrences of a specific value in the field by using the <code>eval</code> command with the <code>count</code> function. For example: <code>count eval(field_name="value")</code> .
	<code>distinct _count(<value>)</code>	Returns the count of distinct values in the field specified.
	<code>estdc(<value>)</code>	Returns the estimated count of the distinct values in the field specified.
	<code>estdc_error(<value>)</code>	Returns the theoretical error of the estimated count of the distinct values in the field specified. The error represents a ratio of the <code>absolute_value(estimate_distinct_count - real_distinct_count) / real_distinct_count</code> .
	<code>exactperc<percentile>(<value>)</code>	Returns a percentile value of the numeric field specified. Provides the exact value, but is very resource expensive for high cardinality fields. An alternative is <code>perc</code> .
	<code>max(<value>)</code>	Returns the maximum value in the field specified. If the field values are non-numeric, the maximum value is found using lexicographical ordering. This function processes field values as numbers if possible, otherwise processes field values as strings.
	<code>mean(<value>)</code>	Returns the arithmetic mean of the values in the field specified.
	<code>median(<value>)</code>	Returns the middle-most value of the values in the field specified.
	<code>min(<value>)</code>	Returns the minimum value in the field specified. If the field values are non-numeric, the minimum value is found using lexicographical ordering.
	<code>mode(<value>)</code>	Returns the most frequent value in the field specified.
	<code>percentile<percentile>(<value>)</code>	Returns the N-th percentile value of all the values in the numeric field specified. Valid field values are integers from 1 to 99. Additional percentile functions are <code>upperperc<percentile>(<value>)</code> and <code>exactperc<percentile>(<value>)</code> .
	<code>range(<value>)</code>	If the field values are numeric, returns the difference between the maximum and minimum values in the field specified.
	<code>stdev(<value>)</code>	Returns the sample standard deviation of the values in the field specified.
	<code>stdevp(<value>)</code>	Returns the population standard deviation of the values in the field specified.
	<code>sum(<value>)</code>	Returns the sum of the values in the field specified.
	<code>sumsq(<value>)</code>	Returns the sum of the squares of the values in the field specified.

	Supported functions and syntax	Description
Event order functions	<code>upperperc<percentile>(<value>)</code>	Returns an approximate percentile value, based on the requested percentile of the numeric field. When there are more than 1000 values, the upperperc function gives the approximate upper bound for the percentile requested. Otherwise the upperperc function returns the same percentile as the perc function.
	<code>var(<value>)</code>	Returns the sample variance of the values in the field specified.
	<code>varp(<value>)</code>	Returns the population variance of the values in the field specified.
	<code>first(<value>)</code>	Returns the first seen value in a field. In general, the first seen value of the field is the most recent instance of this field, relative to the input order of events into the stats command.
	<code>last(<value>)</code>	Returns the last seen value in a field. In general, the last seen value of the field is the oldest instance of this field relative to the input order of events into the stats command.
	<code>list(<value>)</code>	Returns a list of up to 100 values in a field as a multivalue entry. The order of the values reflects the order of input events.
	<code>values(<value>)</code>	Returns the list of all distinct values in a field as a multivalue entry. The order of the values is lexicographical.
	<code>earliest(<value>)</code>	Returns the chronologically earliest (oldest) seen occurrence of a value in a field.
	<code>earliest_time(<value>)</code>	Returns the UNIX time of the earliest (oldest) occurrence of a value of the field. Used in conjunction with the earliest, latest, and latest_time functions to calculate the rate of increase for an accumulating counter.
	<code>latest(<value>)</code>	Returns the chronologically latest (most recent) seen occurrence of a value in a field.
Time functions	<code>latest_time(<value>)</code>	Returns the UNIX time of the latest (most recent) occurrence of a value of the field. Used in conjunction with the earliest, earliest_time, and latest functions to calculate the rate of increase for an accumulating counter.
	<code>per_day(<value>)</code>	Returns the values in a field or eval expression for each day.
	<code>per_hour(<value>)</code>	Returns the values in a field or eval expression for each hour.
	<code>per_minute(<value>)</code>	Returns the values in a field or eval expression for each minute.
	<code>per_second(<value>)</code>	Returns the values in a field or eval expression for each second.
	<code>rate(<value>)</code>	Returns the per-second rate change of the value of the field. Represents $(\text{latest} - \text{earliest}) / (\text{latest_time} - \text{earliest_time})$. Requires the earliest and latest values of the field to be numerical, and the earliest_time and latest_time values to be different.
	<code>rate_avg(<value>)</code>	Returns the average rates for the time series associated with a specified accumulating counter metric.
	<code>rate_sum(<value>)</code>	Returns the summed rates for the time series associated with a specified accumulating counter metric.

Alphabetical list of functions

The following table is a quick reference of the supported statistical and charting functions, organized alphabetically. This table provides a brief description for each function. Use the links in the table to learn more about each function and to see examples.

Supported functions and syntax	Description	Type of function
<code>avg(<value>)</code>	Returns the average of the values in the field specified.	Aggregate functions
<code>count(<value>)</code>	Returns the number of occurrences where the field that you specify contains any value (is not empty). You can also count the occurrences of a specific value in the field by using the <code>eval</code> command with the <code>count</code> function. For example: <code>count eval(field_name="value")</code> .	Aggregate functions
<code>distinct _count(<value>)</code>	Returns the count of distinct values in the field specified.	Aggregate functions
<code>earliest(<value>)</code>	Returns the chronologically earliest (oldest) seen occurrence of a value in the field specified.	Time functions
<code>earliest_time(<value>)</code>	Returns the UNIX time of the earliest (oldest) occurrence of a value in the field specified. Used in conjunction with the <code>earliest</code> , <code>latest</code> , and <code>latest_time</code> functions to calculate the rate of increase for an accumulating counter.	Time functions
<code>estdc(<value>)</code>	Returns the estimated count of the distinct values in the field specified.	Aggregate functions
<code>estdc_error(<value>)</code>	Returns the theoretical error of the estimated count of the distinct values in the field specified. The error represents a ratio of the <code>absolute_value(estimate_distinct_count - real_distinct_count)/real_distinct_count</code> .	Aggregate functions
<code>exactperc<percentile>(<value>)</code>	Returns a percentile value for the numeric field specified. Provides the exact value, but is very resource expensive for high cardinality fields. An alternative is <code>perc</code> .	Aggregate functions
<code>first(<value>)</code>	Returns the first seen value in a field. In general, the first seen value of the field is the most recent instance of this field, relative to the input order of events into the <code>stats</code> command.	Event order functions
<code>last(<value>)</code>	Returns the last seen value in a field. In general, the last seen value of the field is the oldest instance of this field relative to the input order of events into the <code>stats</code> command.	Event order functions
<code>latest(<value>)</code>	Returns the chronologically latest (most recent) seen occurrence of a value in a field.	Time functions
<code>latest_time(<value>)</code>	Returns the UNIX time of the latest (most recent) occurrence of a value of the field. Used in conjunction with the <code>earliest</code> , <code>earliest_time</code> , and <code>latest</code> functions to calculate the rate of increase for an accumulating counter.	Time functions
<code>list(<value>)</code>	Returns a list of up to 100 values in a field as a multivalue entry. The order of the values reflects the order of input events.	Multivalue stats and chart functions
<code>max(<value>)</code>	Returns the maximum value in the field specified. If the field values are non-numeric, the maximum value is found using lexicographical ordering. This function processes field values as numbers if possible, otherwise processes field values as strings.	Aggregate functions
<code>mean(<value>)</code>	Returns the arithmetic mean of the values in the field specified.	Aggregate functions

Supported functions and syntax	Description	Type of function
<code>median(<value>)</code>	Returns the middle-most value of the values in the field specified.	Aggregate functions
<code>min(<value>)</code>	Returns the minimum value in the field specified. If the field values are non-numeric, the minimum value is found using lexicographical ordering.	Aggregate functions
<code>mode(<value>)</code>	Returns the most frequent value in the field specified.	Aggregate functions
<code>perc<percentile>(<value>)</code>	<p>Returns the N-th percentile value of all the values in the numeric field specified. Valid field values are integers from 1 to 99.</p> <p>Additional percentile functions are <code>upperperc</code> and <code>exactperc</code>.</p>	Aggregate functions
<code>per_day(<value>)</code>	Returns the values in a field or eval expression for each day.	Time functions
<code>per_hour(<value>)</code>	Returns the values in a field or eval expression for each hour.	Time functions
<code>per_minute(<value>)</code>	Returns the values in a field or eval expression for each minute.	Time functions
<code>per_second(<value>)</code>	Returns the values in a field or eval expression for each second.	Time functions
<code>range(<value>)</code>	If the field values are numeric, returns the difference between the maximum and minimum values in the field specified.	Aggregate functions
<code>rate(<value>)</code>	Returns the per-second rate change of the value of the field. Represents $(\text{latest} - \text{earliest}) / (\text{latest_time} - \text{earliest_time})$. Requires the earliest and latest values of the field to be numerical, and the <code>earliest_time</code> and <code>latest_time</code> values to be different.	Time functions
<code>rate_avg(<value>)</code>	Returns the average rates for the time series associated with a specified accumulating counter metric.	Time functions
<code>rate_sum(<value>)</code>	Returns the summed rates for the time series associated with a specified accumulating counter metric.	Time functions
<code>stdev(<value>)</code>	Returns the sample standard deviation of the values in the field specified.	Aggregate functions
<code>stdevp(<value>)</code>	Returns the population standard deviation of the values in the field specified.	Aggregate functions
<code>sum(<value>)</code>	Returns the sum of the values in the field specified.	Aggregate functions
<code>sumsq(<value>)</code>	Returns the sum of the squares of the values in the field specified.	Aggregate functions
<code>upperperc<percentile>(<value>)</code>	<p>Returns an approximate percentile value, based on the requested percentile of the numeric field.</p> <p>When there are more than 1000 values, the <code>upperperc</code> function gives the approximate upper bound for the percentile requested. Otherwise the <code>upperperc</code> function returns the same percentile as the <code>perc</code></p>	Aggregate functions

Supported functions and syntax	Description	Type of function
	function.	
values(<value>)	Returns the list of all distinct values in a field as a multivalue entry. The order of the values is lexicographical.	Multivalue stats and chart functions
var(<value>)	Returns the sample variance of the values in the field specified.	Aggregate functions
varp(<value>)	Returns the population variance of the values in the field specified.	Aggregate functions

See also

Commands

[chart](#)
[geostats](#)
[eventstats](#)
[stats](#)
[streamstats](#)
[timechart](#)

Functions

[Evaluation functions](#)

Answers

Have questions? Visit Splunk Answers and search for a specific function or command.

Aggregate functions

Aggregate functions summarize the values from each event to create a single, meaningful value. Common aggregate functions include Average, Count, Minimum, Maximum, Standard Deviation, Sum, and Variance.

Most aggregate functions are used with numeric fields. However, there are some functions that you can use with either alphabetic string fields or numeric fields. The function descriptions indicate which functions you can use with alphabetic strings.

For an overview, see [statistical and charting functions](#).

avg(<value>)

Description

Returns the average of the values of the field specified.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands, and also with [sparkline\(\)](#) charts.

For a list of the related statistical and charting commands that you can use with this function, see [Statistical and charting functions](#).

Basic examples

Example 1

The following example returns the average (mean) "size" for each distinct "host".

```
... | stats avg(size) BY host
```

Example 2

The following example returns the average "thruput" of each "host" for each 5 minute time span.

```
... | bin _time span=5m | stats avg(thruput) BY _time host
```

Example 3

The following example charts the ratio of the average (mean) "size" to the maximum "delay" for each distinct "host" and "user" pair.

```
... | chart eval(avg(size)/max(delay)) AS ratio BY host user
```

Example 4

The following example displays a timechart of the average of cpu_seconds by processor, rounded to 2 decimal points.

```
... | timechart eval(round(avg(cpu_seconds),2)) BY processor
```

Extended examples

Example 1

There are situations where the results of a calculation can return a different accuracy to the very far right of the decimal point. For example, the following search calculates the average of 100 values:

```
| makeresults count=100 | eval test=3.99 | stats avg(test)
```

The result of this calculation is:

avg(test)

avg(test)
3.9900000000000055

When the count is changed to 10000, the results are different:

```
| makeresults count=10000 | eval test=3.99 | stats avg(test)
```

The result of this calculation is:

avg(test)
3.990000000000215

This occurs because numbers are treated as double-precision floating-point numbers.

To mitigate this issue, you can use the `sigfig` function to specify the number of significant figures you want returned.

However, first you need to make a change to the `stats` command portion of the search. You need to change the name of the field `avg(test)` to remove the parenthesis. For example `stats avg(test) AS test`. The `sigfig` function expects either a number or a field name. The `sigfig` function cannot accept a field name that looks like another function, in this case `avg`.

To specify the number of decimal places you want returned, you multiply the field name by 1 and use zeros to specify the number of decimal places. If you want 4 decimal places returned, you would multiply the field name by 1.0000. To return 2 decimal places, multiply by 1.00, as shown in the following example:

```
| makeresults count=10000 | eval test=3.99 | stats avg(test) AS test | eval new_test=sigfig(test*1.00)
```

The result of this calculation is:

test
3.99

Example 2

Chart the average number of events in a transaction, based on transaction duration.

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

1. Run the following search to create a chart to show the average number of events in a transaction based on the duration of the transaction.

```
sourcetype=access_* status=200 action=purchase | transaction clientip maxspan=30m | chart avg(eventcount) by duration span=log2
```

The `transaction` command adds two fields to the results `duration` and `eventcount`. The `eventcount` field tracks the number of events in a single transaction.

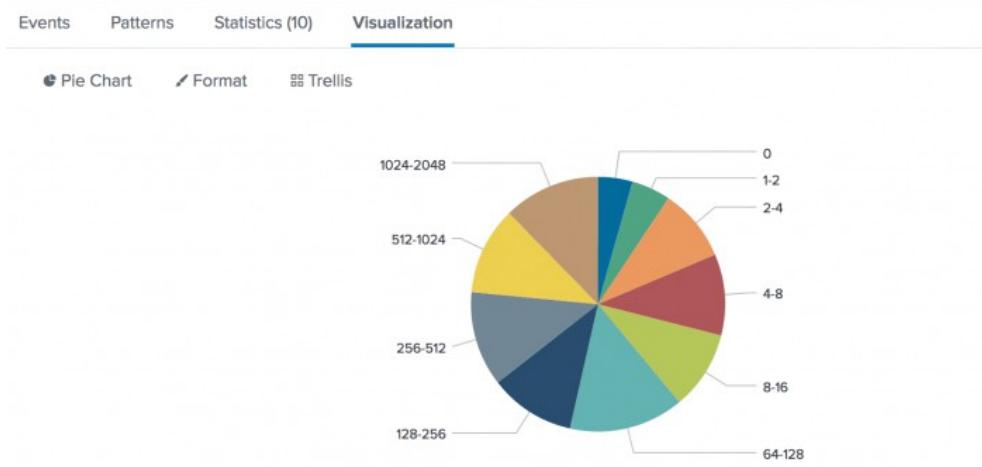
In this search, the transactions are piped into the `chart` command. The `avg()` function is used to calculate the average number of events for each duration. Because the duration is in seconds and you expect there to be many

values, the search uses the `span` argument to bucket the duration into bins using logarithm with a base of 2.

2. Use the field format option to enable number formatting.

A screenshot of the Splunk interface showing a histogram of event counts. The x-axis represents duration bins (0, 1-2, 2-4, 4-8, 8-16, 64-128, 128-256, 256-512, 512-1024, 1024-2048) and the y-axis represents event count. A context menu is open over the 'avg(eventcount)' field, specifically over the 'Number Formatting' section. The 'Disabled' option is selected, while 'Enabled' is available as a choice. The histogram shows a distribution where most events fall into the 0-1024-2048 bins.

3. Click the **Visualization** tab and change the display to a pie chart.



Each wedge of the pie chart represents a duration for the event transactions. You can hover over a wedge to see the average values.

count(<value>) or c(<value>)

Description

Returns the number of occurrences of the field specified. To indicate a specific field value to match, format <value> as eval(field="value"). Processes field values as strings. To use this function, you can specify `count (<value>)`, or the abbreviation `c(<value>)`.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Basic examples

The following example returns the count of events where the `status` field has the value "404".

```
... | stats count(eval(status="404")) AS count_status BY sourcetype
```

This example uses an eval expression with the `count` function. See [Using eval expressions in stats functions](#).

The following example separates search results into 10 bins and returns the count of raw events for each bin.

```
... | bin size bins=10 | stats count(_raw) BY size
```

The following example generates a sparkline chart to count the events that use the `_raw` field.

```
... sparkline(count)
```

The following example generates a sparkline chart to count the events that have the `user` field.

```
... sparkline(count(user))
```

The following example uses the `timechart` command to count the events where the `action` field contains the value `purchase`.

```
sourcetype=access_* | timechart count(eval(action="purchase")) BY productName usenull=f useother=f
```

Extended examples

Count the number of earthquakes that occurred for each magnitude range

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

- Run the following search to calculate the number of earthquakes that occurred in each magnitude range. This data set is comprised of events over a 30-day period.

```
source=all_month.csv | chart count AS "Number of Earthquakes" BY mag span=1 | rename mag AS "Magnitude Range"
```

- This search uses `span=1` to define each of the ranges for the magnitude field, `mag`.
- The [rename command](#) is then used to rename the field to "Magnitude Range".

The results look something like this:

Magnitude Range	Number of Earthquakes
-1-0	18
0-1	2088
1-2	3005
2-3	1026
3-4	194
4-5	452
5-6	109
6-7	11
7-8	3

Count the number of different page requests for each Web server

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 **All time** when you run the search.

- Run the following search to use the `chart` command to determine the number of different page requests, GET and POST, that occurred for each Web server.

```
sourcetype=access_* | chart count(eval(method="GET")) AS GET, count(eval(method="POST")) AS POST BY host
```

This example uses `eval` expressions to specify the different field values for the `stats` command to count. The first clause uses the `count()` function to count the Web access events that contain the `method` field value `GET`. Then, using the `AS` keyword, the field that represents these results is renamed `GET`.

The second clause does the same for `POST` events. The counts of both types of events are then separated by the web server, using the `BY` clause with the `host` field.

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

host	GET	POST
www1	8431	5197
www2	8097	4815
www3	8338	465

2. Click the **Visualization** tab. If necessary, format the results as a column chart. This chart displays the total count of events for each event type, GET or POST, based on the `host` value.



distinct_count(<value>) or dc(<value>)

Description

Returns the count of distinct values of the field specified. This function processes field values as strings. To use this function, you can specify `distinct_count(<value>)`, or the abbreviation `dc(<value>)`.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Basic examples

The following example removes duplicate results with the same "host" value and returns the total count of the remaining results.

```
... | stats dc(host)
```

The following example generates sparklines for the distinct count of devices and renames the field, "numdevices".

```
...sparkline(dc(device)) AS numdevices
```

The following example counts the distinct sources for each sourcetype, and buckets the count for each five minute spans.

```
...sparkline(dc(source), 5m) BY sourcetype
```

Extended example

This example uses the sample data from the Search Tutorial. 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.

1. Run the following search to count the number of different customers who purchased something from the Buttercup Games online store yesterday. The search organizes the count by the type of product (accessories, t-shirts, and type of games) that customers purchased.

```
sourcetype=access_* action=purchase | stats dc(clientip) BY categoryId
```

- ◆ This example first searches for purchase events, `action=purchase`.
- ◆ These results are piped into the `stats` command and the `dc()` function counts the number of different users who make purchases.
- ◆ The `BY` clause is used to break up this number based on the different category of products, the `categoryId`.

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

categoryId	dc(clientip)
ACCESSORIES	37
ARCADE	58
NULL	8
SHOOTER	31
SIMULATION	34
SPORTS	13
STRATEGY	74
TEE	38

estdc(<value>)

Description

Returns the estimated count of the distinct values of the field specified. This function processes field values as strings. The string values 1.0 and 1 are considered distinct values and counted separately.

Usage

You can use this function with the `chart`, `stats`, `timechart`, and `tstats` commands.

By default, if the actual number of distinct values returned by a search is below 1000, the Splunk software does not estimate the distinct value count for the search. It uses the actual distinct value count instead. This threshold is set by the `approx_dc_threshold` setting in `limits.conf`.

Basic examples

The following example removes duplicate results with the same "host" value and returns the estimated total count of the remaining results.

```
... | stats estdc(host)
```

The results look something like this:

estdc(host)
6

The following example generates sparklines for the estimated distinct count of the `devices` field and renames the results field, "numdevices".

```
...sparkline(estdc(device)) AS numdevices
```

The following example estimates the distinct count for the sources for each sourcetype. The results are displayed for each five minute span in sparkline charts.

```
...sparkline(estdc(source),5m) BY sourcetype
```

estdc_error(<value>)

Description

Returns the theoretical error of the estimated count of the distinct values of the field specified. The error represents a ratio of the `absolute_value(estimate_distinct_count - real_distinct_count)/real_distinct_count`. This function processes field values as strings.

Usage

You can use this function with the `chart`, `stats`, and `timechart` commands.

Basic examples

The following example determines the error ratio for the estimated distinct count of the "host" values.

```
... | stats estdc_error(host)
```

exactperc<percentile>(<value>)

Description

Returns a percentile value of the numeric field specified.

Usage

You can use this function with the `chart`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

The `exactperc` function provides the exact value, but is very resource expensive for high cardinality fields. The `exactperc` function can consume a large amount of memory in the search head, which might impact how long it takes for a search to complete.

Examples

See the `perc<percentile>(<value>)` function.

max(<value>)

Description

Returns the maximum value of the field specified. If the values in the field are non-numeric, the maximum value is found using lexicographical ordering.

Processes field values as numbers if possible, otherwise processes field values as strings.

Usage

You can use this function with the `chart`, `mstats`, `stats`, and `timechart` commands, and also with `sparkline()` charts.

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- If the items are all numeric, they're sorted in numerical order based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted as 10, 100, 70, 9.
- if the items are mixed, they're sorted in numeric and then lexicographical order with all numbers sorted before non-numeric items. For example, the items 1, c, a, 2, 100, b, 4, 9 are sorted as 1, 2, 4, 9, 100, a, b, c.
- if all items are non-numeric, they're sorted in lexicographical order.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Basic examples

This example returns the maximum value of the `size` field.

```
... | stats max(size)
```

Extended example

Calculate aggregate statistics for the magnitudes of earthquakes in an area

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

1. Search for earthquakes in and around California. Calculate the number of earthquakes that were recorded. Use statistical functions to calculate the minimum, maximum, range (the difference between the min and max), and

average magnitudes of the recent earthquakes. List the values by magnitude type.

```
source=all_month.csv place=*California* | stats count, max(mag), min(mag), range(mag), avg(mag) BY magType
```

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

magType	count	max(mag)	min(mag)	range(mag)	avg(mag)
H	123	2.8	0.0	2.8	0.549593
MbLg	1	0	0	0	0.000000
Md	1565	3.2	0.1	3.1	1.056486
Me	2	2.0	1.6	.04	1.800000
MI	1202	4.3	-0.4	4.7	1.226622
Mw	6	4.9	3.0	1.9	3.650000
ml	10	1.56	0.19	1.37	0.934000

mean(<value>)

Description

Returns the arithmetic mean of the field specified.

The `mean` values should be exactly the same as the values calculated using the `avg()` function.

Usage

You can use this function with the `chart`, `mstats`, `stats`, and `timechart` commands, and also with `sparkline()` charts.

Basic examples

The following example returns the mean of "kbps" values:

```
... | stats mean(kbps)
```

Extended example

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

1. Run the following search to find the mean, standard deviation, and variance of the magnitudes of recent quakes by magnitude type.

```
source=usgs place=*California* | stats count mean(mag), stdev(mag), var(mag) BY magType
```

The results look something like this:

magType	count	mean(mag)	std(mag)	var(mag)
H	123	0.549593	0.356985	0.127438
MbLg	1	0.000000	0.000000	0.000000
Md	1565	1.056486	0.580042	0.336449
Me	2	1.800000	0.346410	0.120000
Ml	1202	1.226622	0.629664	0.396476
Mw	6	3.650000	0.716240	0.513000
ml	10	0.934000	0.560401	0.314049

median(<value>)

Description

Returns the middle-most value of the field specified.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), and [timechart](#) commands.

If you have an even number of events, by default the median calculation is approximated to the higher of the two values.

This function is, by its nature, nondeterministic. This means that subsequent runs of a search using this function over identical data can contain slight variances in their results.

If you require results that are more exact and consistent you can use `exactperc50()` instead. However, the `exactperc<percentile>(<value>)` function is very resource expensive for high cardinality fields. See [perc<percentile>\(<value>\)](#).

Basic examples

Consider the following list of values, which counts the number of different customers who purchased something from the Buttercup Games online store yesterday. The values are organized by the type of product (accessories, t-shirts, and type of games) that customers purchased.

categoryId	count
ACCESSORIES	37
ARCADE	58
NULL	8
SIMULATION	34
SPORTS	13
STRATEGY	74
TEE	38

When the list is sorted the median, or middle-most value, is 37.

categoryId	count
NULL	8
SPORTS	13
SIMULATION	34
ACCESSORIES	37
TEE	38
ARCADE	58
STRATEGY	74

min(<value>)

Description

Returns the minimum value of the field specified. If the values of X are non-numeric, the minimum value is found using lexicographical ordering.

This function processes field values as numbers if possible, otherwise processes field values as strings.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), and [timechart](#) commands.

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- If the items are all numeric, they're sorted in numerical order based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted as 10, 100, 70, 9.
- If the items are mixed, they're sorted in numeric and then lexicographical order with all numbers sorted before non-numeric items. For example, the items 1, c, a, 2, 100, b, 4, 9 are sorted as 1, 2, 4, 9, 100, a, b, c.
- If all items are non-numeric, they're sorted in lexicographical order.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Basic examples

The following example returns the minimum size and maximum size of the HotBucketRoller component in the _internal index.

```
index=_internal component=HotBucketRoller | stats min(size), max(size)
```

The following example returns a list of processors and calculates the minimum cpu_seconds and the maximum cpu_seconds.

```
index=_internal | chart min(cpu_seconds), max(cpu_seconds) BY processor
```

Extended example

See the [Extended example for the `max\(\)` function](#). That example includes the `min()` function.

mode(<value>)

Description

Returns the most frequent value of the field specified.

Processes field values as strings.

Usage

You can use this function with the `chart`, `stats`, and `timechart` commands.

Basic examples

The `mode` returns the most frequent value. Consider the following data:

firstname	surname	age
Claudia	Garcia	32
David	Mayer	45
Alex	Garcia	29
Wei	Zhang	45
Javier	Garcia	37

When you search for the mode in the `age` field, the value `45` is returned.

```
... | stats mode(age)
```

You can also use mode with fields that contain string values. When you search for the mode in the `surname` field, the value `Garcia` is returned.

```
... | stats mode(surname)
```

Here's another set of sample data:

_time	host	sourcetype
04-06-2020 17:06:23.000 PM	www1	access_combined
04-06-2020 10:34:19.000 AM	www1	access_combined
04-03-2020 13:52:18.000 PM	www2	access_combined
04-02-2020 07:39:59.000 AM	www3	access_combined
04-01-2020 19:35:58.000 PM	www1	access_combined

If you run a search that looks for the mode in the `host` field, the value `www1` is returned because it is the most common value in the `host` field. For example:

```
... |stats mode(host)
```

The results look something like this:

mode(host)
www1

perc<percentile>(<value>)

Description

The percentile functions return the Nth percentile value of the numeric field <value>. You can think of this as an estimate of where the top percentile starts. For example, a 95th percentile says that 95% of the values in field Y are below the estimate and 5% of the values in field <value> are above the estimate.

Valid percentile values are floating point numbers between 0 and 100, such as 99.95.

There are three different percentile functions that you can use:

Function	Description
perc<percentile>(<value>) or the abbreviation p<percentile>(<value>)	Use the <code>perc</code> function to calculate an approximate threshold, such that of the values in field Y, X percent fall below the threshold. The <code>perc</code> function returns a single number that represents the lower end of the approximate values for the percentile requested.
upperperc<percentile>(<value>)	When there are more than 1000 values, the <code>upperperc</code> function gives the approximate upper bound for the percentile requested. Otherwise the <code>upperperc</code> function returns the same percentile as the <code>perc</code> function.
exactperc<percentile>(<value>)	The <code>exactperc</code> function provides the exact value, but is very resource expensive for high cardinality fields. The <code>exactperc</code> function can consume a large amount of memory, which might impact how long it takes for a search to complete.

The percentile functions process field values as strings.

The `perc` and `upperperc` functions are, by their nature, nondeterministic, which means that subsequent runs of searches using these functions over identical data can contain slight variances in their results.

If you require exact and consistent results, you can use `exactperc<X>(Y)` instead.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands.

Differences between Splunk and Excel percentile algorithms

If there are less than 1000 distinct values, the Splunk percentile functions use the nearest rank algorithm. See http://en.wikipedia.org/wiki/Percentile#Nearest_rank. Excel uses the NIST interpolated algorithm, which basically means you can get a value for a percentile that does not exist in the actual data, which is not possible for the nearest rank approach.

Splunk algorithm with more than 1000 distinct values

If there are more than 1000 distinct values for the field, the percentiles are approximated using a custom radix-tree digest-based algorithm. This algorithm is much faster and uses much less memory, a constant amount, than an exact computation, which uses memory in linear relation to the number of distinct values. By default this approach limits the approximation error to < 1% of rank error. That means if you ask for 95th percentile, the number you get back is between the 94th and 96th percentile.

You always get the exact percentiles even for more than 1000 distinct values by using the `exactperc` function compared to the `perc`.

Basic examples

Consider this list of values `Y = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}`.

The following example returns 5.5.

```
... | stats perc50(Y)
```

The following example returns 9.55.

```
... | stats perc95(Y)
```

Extended example

Consider the following set of data, which shows the number of visitors for each hour a store is open:

hour	visitors
0800	0
0900	212
1000	367
1100	489
1200	624
1300	609
1400	492
1500	513
1600	376
1700	337

This data resides in the `visitor_count` index. You can use the `streamstats` command to create a cumulative total for the visitors.

```
index=visitor_count | streamstats sum(visitors) as 'visitors total'
```

The results from this search look like this:

hour	visitors	visitors total
------	----------	----------------

0800	0	0
0900	212	212
1000	367	579
1100	489	1068
1200	624	1692
1300	609	2301
1400	492	2793
1500	513	3306
1600	376	3673
1700	337	4010

Let's add the `stats` command with the `perc` function to determine the 50th and 95th percentiles.

```
index=visitor_count | streamstats sum(visitors) as 'visitors total' | stats perc50('visitors total') perc95('visitors total')
```

The results from this search look like this:

perc50(visitors total)	perc95(visitors total)
1996.5	3858.35

The `perc50` estimates the 50th percentile, when 50% of the visitors had arrived. You can see from the data that the 50th percentile was reached between visitor number 1996 and 1997, which was sometime between 1200 and 1300 hours. The `perc95` estimates the 95th percentile, when 95% of the visitors had arrived. The 95th percentile was reached with visitor 3858, which occurred between 1600 and 1700 hours.

range(<value>)

Description

Returns the difference between the max and min values of the field specified. The values in the field must be numeric.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Basic example

This example uses events that list the numeric sales for each product and quarter, for example:

products	quarter	sales	quota
ProductA	QTR1	1200	1000
ProductB	QTR1	1400	1550
ProductC	QTR1	1650	1275
ProductA	QTR2	1425	1300

products	quarter	sales	quota
ProductB	QTR2	1175	1425
ProductC	QTR2	1550	1450
ProductA	QTR3	1300	1400
ProductB	QTR3	1250	1125
ProductC	QTR3	1375	1475
ProductA	QTR4	1550	1300
ProductB	QTR4	1700	1225
ProductC	QTR4	1625	1350

It is easiest to understand the `range` if you also determine the `min` and `max` values. To determine the range of sales by product, run this search:

```
source="addtotalsData.csv" | chart sum(sales) min(sales) max(sales) range(sales) BY products
```

The results look something like this:

quarter	sum(sales)	min(sales)	max(sales)	range(sales)
QTR1	4250	1200	1650	450
QTR2	4150	1175	1550	375
QTR3	3925	1250	1375	125
QTR4	4875	1550	1700	150

The `range(sales)` is the `max(sales)` minus the `min(sales)`.

Extended example

See the [Extended example for the `max\(\)` function](#). That example includes the `range()` function.

stdev(<value>)

Description

Returns the sample standard deviation of the field specified.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Basic examples

This example returns the standard deviation of wildcarded fields "`*delay`" which can apply to both, "delay" and "xdelay".

```
... | stats stdev(*delay)
```

Extended example

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

1. Run the following search to find the mean, standard deviation, and variance of the magnitudes of recent quakes by magnitude type.

```
source=usgs place=*California* | stats count mean(mag), stdev(mag), var(mag) BY magType
```

The results look something like this:

magType	count	mean(mag)	std(mag)	var(mag)
H	123	0.549593	0.356985	0.127438
MbLg	1	0.000000	0.000000	0.000000
Md	1565	1.056486	0.580042	0.336449
Me	2	1.800000	0.346410	0.120000
MI	1202	1.226622	0.629664	0.396476
Mw	6	3.650000	0.716240	0.513000
ml	10	0.934000	0.560401	0.314049

stdevp(<value>)

Description

Returns the population standard deviation of the field specified.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands, and also with sparkline() charts.

Basic examples

Extended example

sum(<value>)

Description

Returns the sum of the values of the field specified.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands, and also with `sparkline()` charts.

Basic examples

You can create totals for any numeric field. For example:

```
...| stats sum(bytes)
```

The results look something like this:

sum(bytes)
21502

You can rename the column using the AS keyword:

```
...| stats sum(bytes) AS "total bytes"
```

The results look something like this:

total bytes
21502

You can organize the results using a BY clause:

```
...| stats sum(bytes) AS "total bytes" by date_hour
```

The results look something like this:

date_hour	total bytes
07	6509
11	3726
15	6569
23	4698

sumsq(<value>)

Description

Returns the sum of the squares of the values of the field specified.

The sum of the squares is used to evaluate the variance of a dataset from the dataset mean. A large sum of the squares indicates a large variance, which tells you that individual values fluctuate widely from the mean.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands, and also with `sparkline()` charts.

Basic examples

The following table contains the temperatures taken every day at 8 AM for a week.

You calculate the mean of these temperatures and get 48.9 degrees. To calculate the deviation from the mean for each day, take the temperature and subtract the mean. If you square each number, you get results like this:

day	temp	mean	deviation	square of temperatures
sunday	65	48.9	16.1	260.6
monday	42	48.9	-6.9	47.0
tuesday	40	48.9	-8.9	78.4
wednesday	31	48.9	-17.9	318.9
thursday	47	48.9	-1.9	3.4
friday	53	48.9	4.1	17.2
saturday	64	48.9	15.1	229.3

Take the total of the squares, 954.9, and divide by 6 which is the number of days minus 1. This gets you the sum of squares for this series of temperatures. The standard deviation is the square root of the sum of the squares. The larger the standard deviation the larger the fluctuation in temperatures during the week.

You can calculate the mean, sum of the squares, and standard deviation with a few statistical functions:

```
...|stats mean(temp), sumsq(temp), stdev(temp)
```

This search returns these results:

mean(temp)	sumsq(temp)	stdev(temp)
48.857142857142854	17664	12.615183595289349

upperperc<percentile>(<value>)

Description

Returns an approximate percentile value, based on the requested percentile of the numeric field.

When there are more than 1000 values, the `upperperc` function gives the approximate upper bound for the percentile requested. Otherwise the `upperperc` function returns the same percentile as the `perc` function.

See the [percentile<percentile>\(<value>\)](#) function.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands, and also with `sparkline()` charts.

Examples

See the `perc` function.

var(<value>)

Description

Returns the sample variance of the field specified.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Example

See the [Extended example for the mean\(\) function](#). That example includes the `var()` function.

varp(<value>)

Description

Returns the population variance of the field specified.

Usage

You can use this function with the `chart`, `mstats`, `stats`, `timechart`, and `tstats` commands, and also with `sparkline()` charts.

Basic examples

Event order functions

Use the event order functions to return values from fields based on the order in which the event is processed, which is not necessarily chronological or timestamp order.

The following table lists the timestamps from a set of events returned from a search. This table identifies which event is returned when you use the `first` and `last` event order functions, and compares them with the `earliest` and `latest` functions, which you can read more about at [Time functions](#).

<code>_time</code>	Event order function	Description
2020-04-28 00:15:05	first	This event is the first event in the search results. But this event is not chronologically the earliest event.
2020-05-01 00:15:04		
2020-04-30 00:15:02		

<code>_time</code>	Event order function	Description
2020-04-28 00:15:01		
2020-05-01 00:15:05	latest	This event is chronologically the latest event in the search results.
2020-04-27 00:15:01	earliest last	This event is both the chronologically earliest event and the last event in the search results.

See [Overview of statistical and charting functions](#).

first(<value>)

Description

Returns the first seen value in a field. The first seen value of the field is the most recent instance of this field, based on the order in which the events are seen by the `stats` command. The order in which the events are seen is not necessarily chronological order.

Usage

You can use this function with the `chart`, `stats`, and `timechart` commands.

- To locate the first value based on time order, use the `earliest` function instead.
- This function works best when the search includes the `sort` command immediately before the statistics or charting command.
- This function processes field values as strings.

Basic example

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

You run the following search to locate invalid user login attempts against a specific sshd (Secure Shell Daemon). You use the `table` command to see the values in the `_time`, `source`, and `_raw` fields.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw
```

The results look something like this:

<code>_time</code>	<code>source</code>	<code>_raw</code>
2020-04-28 00:15:05	tutorialdata.zip:./mailsv/secure.log	Mon Apr 28 2020 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2
2020-05-01 00:15:04	tutorialdata.zip:./www2/secure.log	Thu May 01 2020 00:15:04 www2 sshd[5258]: Failed password for invalid user brian from 130.253.37.97 port 4284 ssh2
2020-04-30 00:15:02	tutorialdata.zip:./www3/secure.log	Wed Apr 30 2020 00:15:02 www3 sshd[5258]: Failed password for invalid user operator from 222.169.224.226 port 1711 ssh2

<code>_time</code>	<code>source</code>	<code>_raw</code>
2020-04-28 00:15:01	tutorialdata.zip:./www1/secure.log	Mon Apr 28 2020 00:15:01 www1 sshd[5258]: Failed password for invalid user rightscale from 87.194.216.51 port 3361 ssh2
2020-05-01 00:15:05	tutorialdata.zip:./mailsv/secure.log	Thu May 01 2020 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user testuser from 194.8.74.23 port 3626 ssh2
2020-04-27 00:15:01	tutorialdata.zip:./www1/secure.log	Sun Apr 27 2020 00:15:01 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

You extend the search using the `first` function.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw | stats first(_raw)
```

The search returns the value for `_raw` field with the timestamp 2020-04-28 00:15:05, which is the first event in the original list of values returned.

<code>first(_raw)</code>
Mon Apr 28 2020 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2

Extended example

The Basic example uses the `_raw` field to show how the `first` function works. That's useful because the `_raw` field contains a timestamp. However, you can use the `first` function on any field.

Let's start by creating some results. You can use the `makeresults` command to create a series of results to test your search syntax. Include the `streamstats` command to count your results:

```
| makeresults count=5 | streamstats count
```

The results look like this:

<code>_time</code>	<code>count</code>
2020-05-09 14:35:58	1
2020-05-09 14:35:58	2
2020-05-09 14:35:58	3
2020-05-09 14:35:58	4
2020-05-09 14:35:58	5

With the `count` field, you can create different dates in the `_time` field, using the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600)
```

Use 3600, the number of seconds in an hour, to create a series of hours. The calculation multiplies the value in the `count` field by the number of seconds in an hour. The result is subtracted from the original `_time` field to get new dates equivalent to 1 hours ago, 2 hours ago, and so forth.

The results look like this:

_time	count
2020-05-09 13:45:24	1
2020-05-09 12:45:24	2
2020-05-09 11:45:24	3
2020-05-09 10:45:24	4
2020-05-09 09:45:24	5

The hours in the results begin with the 1 hour earlier than the original date, 2020-05-09 at 14:24. The minutes and seconds are slightly different because the date is refreshed each time you run the search.

Use the `eval` command to add a field to your search with values in descending order:

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600) | eval field1=20-count
```

The results look like this:

_time	count	field1
2020-05-09 14:45:24	1	19
2020-05-09 13:45:24	2	18
2020-05-09 12:45:24	3	17
2020-05-09 11:45:24	4	16
2020-05-09 10:45:24	5	15

As you can see from the results, the first result contains the highest number in `field1`. This shows the order in which the results were processed. The first result was processed first ($20-1=19$) followed by the remaining results in order.

When you add the `first` function to the search, the only value returned is the value in the field you specify:

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600) | eval field1=20-count | stats first(field1)
```

The results look like this:

first(field1)
19

last(<value>)

Description

Returns the last seen value in a field. The last seen value of the field is the oldest instance of this field, based on the order in which the events are seen by the `stats` command. The order in which the events are seen is not necessarily chronological order.

Usage

You can use this function with the `chart`, `stats`, and `timechart` commands.

- To locate the last value based on time order, use the `latest` function instead.
- This function works best when the search includes the `sort` command immediately before the statistics or charting command.
- This function processes field values as strings.

Basic example

The following example returns the first "log_level" value for each distinct "sourcetype".

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

You run the following search to locate invalid user login attempts against a specific sshd (Secure Shell Daemon). You use the `table` command to see the values in the `_time`, `source`, and `_raw` fields.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw
```

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

<code>_time</code>	<code>source</code>	<code>_raw</code>
2020-04-28 00:15:05	tutorialdata.zip:./mailsv/secure.log	Mon Apr 28 2020 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2
2020-05-01 00:15:04	tutorialdata.zip:./www2/secure.log	Thu May 01 2020 00:15:04 www2 sshd[5258]: Failed password for invalid user brian from 130.253.37.97 port 4284 ssh2
2020-04-30 00:15:02	tutorialdata.zip:./www3/secure.log	Wed Apr 30 2020 00:15:02 www3 sshd[5258]: Failed password for invalid user operator from 222.169.224.226 port 1711 ssh2
2020-04-28 00:15:01	tutorialdata.zip:./www1/secure.log	Mon Apr 28 2020 00:15:01 www1 sshd[5258]: Failed password for invalid user rightscale from 87.194.216.51 port 3361 ssh2
2020-05-01 00:15:05	tutorialdata.zip:./mailsv/secure.log	Thu May 01 2020 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user testuser from 194.8.74.23 port 3626 ssh2
2020-04-27 00:15:01	tutorialdata.zip:./www1/secure.log	Sun Apr 27 2020 00:15:01 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

You extend the search using the `last` function.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw | stats last(_raw)
```

The search returns the event with the `_time` value 2020-04-27 00:15:01, which is the last event in the list of events. However it is not the last chronological event.

<code>_time</code>	<code>source</code>	<code>_raw</code>
2020-04-27 00:15:01	tutorialdata.zip:./www1/secure.log	Sun Apr 27 2020 00:15:01 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

_time	source	_raw
-------	--------	------

Extended example

The Basic example uses the `_raw` field to show how the `last` function works. That's useful because the `_raw` field contains a timestamp. However, you can use the `last` function on any field.

Let's start by creating some results. You can use the `makeresults` command to create a series of results to test your search syntax. Include the `streamstats` command to count your results:

```
| makeresults count=5 | streamstats count
```

The results look like this:

_time	count
2020-05-09 14:35:58	1
2020-05-09 14:35:58	2
2020-05-09 14:35:58	3
2020-05-09 14:35:58	4
2020-05-09 14:35:58	5

With the `count` field, you can create different dates in the `_time` field, using the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*86400)
```

Use 86400, the number of seconds in a day, to create a series of days. The calculation multiplies the value in the `count` field by the number of seconds in a day. The result is subtracted from the original `_time` field to get new dates equivalent to 1 day ago, 2 days ago, and so forth.

The results look like this:

_time	count
2020-05-08 14:45:24	1
2020-05-07 14:45:24	2
2020-05-06 14:45:24	3
2020-05-05 14:45:24	4
2020-05-04 14:45:24	5

The dates in the results begin with the 1 day earlier than the original date, 2020-05-09 at 14:45:24. The minutes and seconds are slightly different because the date is refreshed each time you run the search.

Use the `eval` command to add a field to your search with values in descending order:

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*86400) | eval field1=20-count
```

The results look like this:

_time	count	field1
-------	-------	--------

2020-05-08 14:45:24	1	19
2020-05-07 14:45:24	2	18
2020-05-06 14:45:24	3	17
2020-05-05 14:45:24	4	16
2020-05-04 14:45:24	5	15

As you can see from the results, the last result contains the lowest number in `field1`. This shows the order in which the results were processed. The fifth result was processed last ($20-5=15$) after all of the other results.

When you add the `last` function to the search, the only value returned is the value in the field you specify:

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*86400) | eval field1=20-count | stats last(field1)
```

The results look like this:

lastfield1)
15

See also

Commands

[eval](#)

[makeresults](#)

Multivalue stats and chart functions

list(<value>)

Description

The `list` function returns a multivalue entry from the values in a field. The order of the values reflects the order of the events.

Usage

You can use this function with the `chart`, `stats`, and `timechart` commands.

- If more than 100 values are in a field, only the first 100 are returned.
- This function processes field values as strings.

Basic example

To illustrate what the `list` function does, let's start by generating a few simple results.

1. Use the `makeresults` and `streamstats` commands to generate a set of results that are simply timestamps and a count of the results which are used as row numbers.

```
| makeresults count=1000 | streamstats count AS rowNum
```

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

_time	rowNumber
2018-04-02 20:27:11	1
2018-04-02 20:27:11	2
2018-04-02 20:27:11	3
2018-04-02 20:27:11	4
2018-04-02 20:27:11	5

Notice that each result appears on a separate row.

2. Add the `stats` command with the `list` function to the search. The numbers are returned in ascending order in a single, multivalue result.

```
| makeresults count=1000 | streamstats count AS rowNum | stats list(rowNum) AS numbers
```

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

numbers
1
2
3
4
5

Notice that it is a single result. There are no alternating row background colors.

3. Compare this result with the results returned by the `values` function.

values(<values>)

Description

The `values` function returns a list of the distinct values in a field as a multivalue entry. The order of the values is lexicographical.

Usage

You can use the `values(x)` function with the `chart`, `stats`, `timechart`, and `tstats` commands.

- By default there is no limit to the number of values returned. Users with the appropriate permissions can specify a limit in the `limits.conf` file. You specify the limit in the `[stats | sistats]` stanza using the `maxvalues` setting.
- This function processes field values as strings.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Basic example

To illustrate what the `values` function does, let's start by generating a few simple results.

1. Use the `makeresults` and `streamstats` commands to generate a set of results that are simply timestamps and a count of the results, which are used as row numbers.

```
| makeresults count=1000 | streamstats count AS rowNumber
```

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

_time	rowNumber
2018-04-02 20:27:11	1
2018-04-02 20:27:11	2
2018-04-02 20:27:11	3
2018-04-02 20:27:11	4
2018-04-02 20:27:11	5

Notice that each result appears on a separate row.

2. Add the `stats` command with the `values` function to the search. The results are returned in lexicographical order.

```
| makeresults count=1000 | streamstats count AS rowNumber | stats values(rowNumber) AS numbers
```

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

numbers
1
10
100
1000
101
102
103
104
105
106
107
108
109
11
110

Notice that it is a single result. There are no alternating row background colors.

3. Compare these results with the results returned by the `list` function.

Time functions

earliest(<value>)

Description

Returns the chronologically earliest seen occurrence of a value in a field.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands.

This function processes field values as strings.

Basic example

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

You run the following search to locate invalid user login attempts against a sshd (Secure Shell Daemon). You use the `table` command to see the values in the `_time`, `source`, and `_raw` fields.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw
```

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

<code>_time</code>	<code>source</code>	<code>_raw</code>
2022-05-02 00:15:05	tutorialdata.zip:./mailsv/secure.log	Mon May 02 2022 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2
2022-05-01 00:16:17	tutorialdata.zip:./www2/secure.log	Sun May 01 2022 00:16:17 www2 sshd[5258]: Failed password for invalid user brian from 130.253.37.97 port 4284 ssh2
2022-05-01 00:11:25	tutorialdata.zip:./www3/secure.log	Sun May 01 2022 00:11:25 www3 sshd[5258]: Failed password for invalid user operator from 222.169.224.226 port 1711 ssh2
2022-04-30 00:19:01	tutorialdata.zip:./www1/secure.log	Sat Apr 30 2022 00:19:01 www1 sshd[5258]: Failed password for invalid user rightscale from 87.194.216.51 port 3361 ssh2
2022-04-30 00:13:45	tutorialdata.zip:./mailsv/secure.log	Sat Apr 01 2022 00:13:45 mailsv1 sshd[5258]: Failed password for invalid user testuser from 194.8.74.23 port 3626 ssh2
2022-04-29 00:23:28	tutorialdata.zip:./www1/secure.log	Fri Apr 29 2022 00:23:28 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

You extend the search using the `earliest` function.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw | stats earliest(_raw)
```

The search returns the event with the `_time` value 2022-04-29 00:23:28, which is the event with the oldest timestamp.

<code>_time</code>	<code>source</code>	<code>_raw</code>
2022-04-29 00:23:28	tutorialdata.zip:./www1/secure.log	Fri Apr 29 2022 00:23:28 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

2022-04-29 00:23:28	tutorialdata.zip:./www1/secure.log	Fri Apr 29 2022 00:23:28 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2
------------------------	------------------------------------	--

earliest_time(<value>)

Description

Returns the UNIX time of the chronologically earliest-seen occurrence of a given field value.

Usage

You can use this function with the [mstats](#), [stats](#), and [tstats](#) commands.

This function processes field values as strings.

If you have metrics data, you can use `earliest_time` function in conjunction with the `earliest`, `latest`, and `latest_time` functions to calculate the rate of increase for a counter. Alternatively you can use the `rate` counter to do the same thing.

Basic example

The following search runs against metric data. It is designed to return the earliest UNIX time values on every minute for each `metric_name` that begins with `deploy`.

```
| mstats earliest_time(_value) where index=_metrics metric_name=deploy* BY metric_name span=1m
```

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

_time	metric_name	earliest_time(_value)
2018-11-11 18:14:00	deploy-connections.nCurrent	1541988860.000000
2018-11-11 18:14:00	deploy-connections.nStarted	1541988860.000000
2018-11-11 18:14:00	deploy-server.volumeCompletedKB	1541988860.000000
2018-11-11 18:15:00	deploy-connections.nCurrent	1541988922.000000
2018-11-11 18:15:00	deploy-connections.nStarted	1541988922.000000
2018-11-11 18:15:00	deploy-server.volumeCompletedKB	1541988922.000000

latest(<value>)

Description

Returns the chronologically latest seen occurrence of a value in a field.

Usage

You can use this function with the [chart](#), [mstats](#), [stats](#), [timechart](#), and [tstats](#) commands.

This function processes field values as strings.

Basic example

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

You run the following search to locate invalid user login attempts against a specific sshd (Secure Shell Daemon). You use the `table` command to see the values in the `_time`, `source`, and `_raw` fields.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw
```

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

<code>_time</code>	<code>source</code>	<code>_raw</code>
2022-05-02 00:15:05	tutorialdata.zip:/mailsv/secure.log	Mon May 02 2022 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2
2022-05-01 00:16:17	tutorialdata.zip:/www2/secure.log	Sun May 01 2022 00:16:17 www2 sshd[5258]: Failed password for invalid user brian from 130.253.37.97 port 4284 ssh2
2022-05-01 00:11:25	tutorialdata.zip:/www3/secure.log	Sun May 01 2022 00:11:25 www3 sshd[5258]: Failed password for invalid user operator from 222.169.224.226 port 1711 ssh2
2022-04-30 00:19:01	tutorialdata.zip:/www1/secure.log	Sat Apr 30 2022 00:19:01 www1 sshd[5258]: Failed password for invalid user rightscale from 87.194.216.51 port 3361 ssh2
2022-04-30 00:13:45	tutorialdata.zip:/mailsv/secure.log	Sat Apr 01 2022 00:13:45 mailsv1 sshd[5258]: Failed password for invalid user testuser from 194.8.74.23 port 3626 ssh2
2022-04-29 00:23:28	tutorialdata.zip:/www1/secure.log	Fri Apr 29 2022 00:23:28 www1 sshd[5258]: Failed password for invalid user redmine from 91.208.184.24 port 3587 ssh2

You extend the search using the `latest` function.

```
sourcetype=secure invalid user "sshd[5258]" | table _time source _raw | stats latest(_raw)
```

The search returns the event with the `_time` value 2022-05-02 00:15:05, which is the event with the most recent timestamp.

<code>_time</code>	<code>source</code>	<code>_raw</code>
2022-05-02 00:15:05	tutorialdata.zip:/mailsv/secure.log	Mon May 02 2022 00:15:05 mailsv1 sshd[5258]: Failed password for invalid user tomcat from 67.170.226.218 port 1490 ssh2

latest_time(<value>)

Description

Returns the UNIX time of the chronologically latest-seen occurrence of a given field value.

Usage

You can use this function with the `mstats`, `stats`, and `tstats` commands.

This function processes field values as strings.

If you have metrics data, you can use `latest_time` function in conjunction with `earliest`, `latest`, and `earliest_time` functions to calculate the rate of increase for a counter. Alternatively, you can use the `rate` function counter to do the same thing.

Basic example

The following search runs against metric data. It is designed to return the latest UNIX time values in the past 60 minutes for metrics with names that begin with `queue..`

```
| mstats latest_time(_value) where index=_metrics metric_name=queue.* BY metric_name span=1m
```

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

_time	metric_name	earliest_time(_value)
2018-11-13 14:43:00	queue.current_size	1542149039.000000
2018-11-13 14:43:00	queue.current_size_kb	1542149039.000000
2018-11-13 14:43:00	queue.largest_size	1542149039.000000
2018-11-13 14:43:00	queue.max_size_kb	1542149039.000000
2018-11-13 14:43:00	queue.smallest_size	1542149039.000000
2018-11-13 14:44:00	queue.current_size	1542149070.000000
2018-11-13 14:44:00	queue.current_size_kb	1542149070.000000
2018-11-13 14:44:00	queue.largest_size	1542149070.000000
2018-11-13 14:44:00	queue.max_size_kb	1542149070.000000
2018-11-13 14:44:00	queue.smallest_size	1542149070.000000

per_day(<value>)

Description

Returns the values in a field or eval expression for each day.

Usage

You can use this function with the `timechart` command.

Basic examples

The following example returns the values for the field `total` for each day.

```
... | timechart per_day(total)
```

The following example returns the results of the eval expression `eval(method="GET")) AS Views`.

```
... | timechart per_day(eval(method="GET")) AS Views
```

Extended example

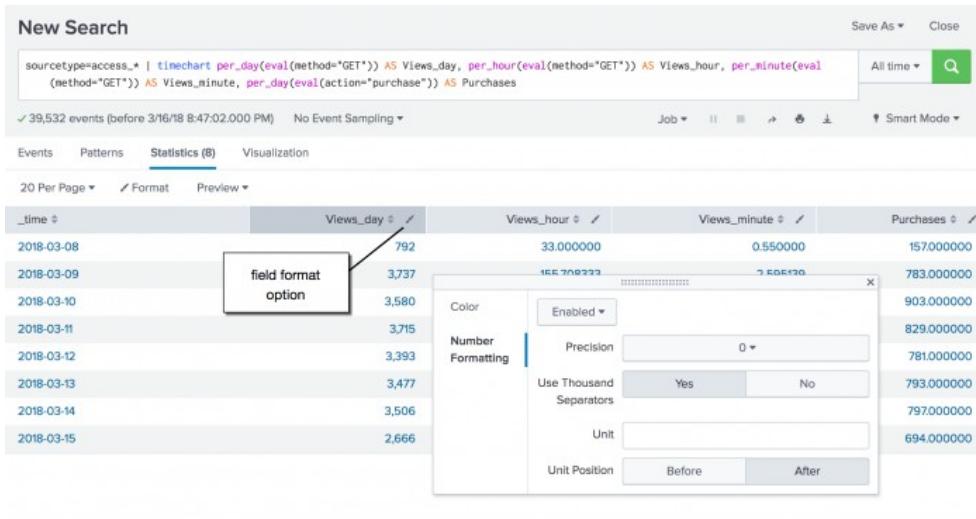
This example uses the sample dataset from the Search Tutorial but should work with any format of Apache Web access log. Download the data set from [this topic in the Search Tutorial](#) and follow the instructions to upload it to your Splunk deployment.

This search uses the `per_day()` function and eval expressions to determine how many times the web pages were viewed and how many times items were purchased. The results appear on the Statistics tab.

```
sourcetype=access_* | timechart per_day(eval(method="GET")) AS Views_day, per_day(eval(action="purchase")) AS Purchases
```

To determine the number of Views and Purchases for each hour, minute, or second you can add the other time functions to the search. For example:

```
sourcetype=access_* | timechart per_day(eval(method="GET")) AS Views_day, per_hour(eval(method="GET")) AS Views_hour, per_minute(eval(method="GET")) AS Views_minute, per_day(eval(action="purchase")) AS Purchases
```



Use the field format option to change the number formatting for the field values.

per_hour(<value>)

Description

Returns the values in a field or eval expression for each hour.

Usage

You can use this function with the `timechart` command.

Basic examples

The following example returns the values for the field `total` for each hour.

```
... | timechart per_hour(total)
```

The following example returns the the results of the eval expression `eval(method="POST")) AS Views`.

```
... | timechart per_hour(eval(method="POST")) AS Views
```

per_minute(<value>)

Description

Returns the values in a field or eval expression for each minute.

Usage

You can use this function with the `timechart` command.

Basic examples

The following example returns the values for the field `total` for each minute.

```
... | timechart per_minute(total)
```

The following example returns the results of the eval expression `eval(method="GET")) AS Views`.

```
... | timechart per_minute(eval(method="GET")) AS Views
```

per_second(<value>)

Description

Returns the values in a field or eval expression for each second.

Usage

You can use this function with the `timechart` command.

Basic examples

The following example returns the values for the field `kb` for each second.

```
... | timechart per_second(kb)
```

rate(<value>)

Description

Returns the per-second rate change of the value in a field. The `rate` function represents the following formula:

$$(\text{latest}(<\text{value}>) - \text{earliest}(<\text{value}>)) / \text{latest_time}(<\text{value}>) - \text{earliest_time}(<\text{value}>))$$

The `rate` function also handles the largest value reset if there is at least one reset.

Usage

You can use this function with the `mstats`, `stats`, and `tstats` commands.

- Provides the per-second rate change for an accumulating **counter metric**. Accumulating counter metrics report the total counter value since the last counter reset. See Investigate counter metrics in *Metrics*
- Requires the `earliest` and `latest` values of the field to be numerical, and the `earliest_time` and `latest_time` values to be different.
- Requires at least two metric data points in the search time range.
- Should be used to provide rate information about single, rather than multiple, counters.

Basic example

The following search runs against metric data. It provides the hourly hit rate for a metric that provides measurements of incoming web traffic. It uses the `processor` filter to ensure that it is not reporting on multiple metric series (`name` and `processor` combinations).

```
| mstats rate(traffic.incoming) as rate_hits where index=_metrics name=indexerpipe processor=index_thruput span=1h
```

The resulting chart shows you that the counter hit rate for the `traffic.incoming` metric spiked at 1 pm, 4 pm, and 11 am, but otherwise remained stable.



rate_avg(<value>)

Description

Computes the per **metric time series** rates for an accumulating **counter metric**. Returns the averages of those rates.

For a detailed explanation of metric time series, see Perform statistical calculations on metric time series in *Metrics*.

Usage

You can use this function with the `mstats` command.

- To ensure accurate results, Splunk software uses the latest value of a metric measurement from the previous timespan as the starting basis for a rate computation.
- When you calculate the average rates for accumulating counter metrics, the cleanest way to do it is to split the counter metric rate calculations out by metric time series and then compute the average rate across all of the

metric time series.

- Unlike `rate`, the `rate_avg` function can calculate rates even when there is only a single metric data point per time series per timespan. It can pull in data across timespans to calculate rates when necessary.
- The `rate_avg` function does not support `prestats=true`. It needs the final list of dimensions to split by.

Basic example

In your `_metrics` index, you have data for the metric `spl.intr.resource_usage.PerProcess.data.elapsed`. This is an accumulating counter metric. It contains a number of metric time series.

The following example search uses the `rate_avg` function to calculate the `rate(X)` for each `spl.mlog.thruput.thruput.total_k_processed` time series in the time range. Then it gets the average rate across all of the time series. Lastly, it splits the results by time, so they can be plotted on a chart.

```
| mstats rate_avg(spl.mlog.thruput.thruput.total_k_processed) where index=_metrics span=1h
```

rate_sum(<value>)

Description

Computes the per **metric time series** rates for an accumulating **counter metric**. Returns the aggregate of those rates.

For a detailed explanation of metric time series, see Perform statistical calculations on metric time series in *Metrics*.

Usage

You can use this function with the `mstats` command.

- To ensure accurate results, Splunk software uses the latest value of a metric measurement from the previous timespan as the starting basis for a rate computation.
- When you calculate the aggregated rates for accumulating counter metrics, the cleanest way to do it is to split the counter metric rate calculations out by metric time series and then compute the aggregate rate across all of the metric time series.
- Unlike `rate`, the `rate_sum` function can calculate rates even when there is only a single metric data point per time series per timespan. It can pull in data across timespans to calculate rates when necessary.
- The `rate_sum` function does not support `prestats=true`. It needs the final list of dimensions to split by.

Basic example

In your `_metrics` index, you have data for the metric `spl.intr.resource_usage.PerProcess.data.elapsed`. This is an accumulating counter metric. It contains a number of metric time series.

The following example search uses the `rate_sum` function to calculate the `rate(X)` for each `spl.mlog.thruput.thruput.total_k_processed` time series in the time range. Then it gets the aggregate rate across all of the time series. Lastly, it splits the results by time, so they can be plotted on a chart.

```
| mstats rate_sum(spl.mlog.thruput.thruput.total_k_processed) where index=_metrics span=1h
```

Time Format Variables and Modifiers

Date and time format variables

This topic lists the variables that you can use to define time formats in the evaluation functions, [strftime\(\)](#) and [strptime\(\)](#). You can also use these variables to describe timestamps in event data.

Additionally, you can use the `relative_time()` and `now()` time functions as arguments.

For more information about working with dates and time, see [Time modifiers for search](#) and [About searching with time](#) in the *Search Manual*.

Refer to the list of tz database time zones for all permissible time zone values. For more information about how the Splunk software determines a time zone and the tz database, see [Specify time zones for timestamps](#) in *Getting Data In*.

Subsecond time variables such as %N and %Q can be used in metrics searches of metrics indexes that are enabled for millisecond timestamp resolution.

For more information about enabling metrics indexes to index metric data points with millisecond timestamp precision:

- For Splunk Cloud Platform, see [Manage Splunk Cloud Platform indexes](#) in the *Splunk Cloud Platform Admin Manual*.
- For Splunk Enterprise, see [Create custom indexes](#) in *Managing indexers and clusters of indexers*.

Date and time variables

Variable	Description
%C	The date and time in the current locale's format as defined by the server's operating system. For example, Thu Jul 18 09:30:00 2019 for US English on Linux.
%+	The date and time with time zone in the current locale's format as defined by the server's operating system. For example, Thu Jul 18 09:30:00 PDT 2019 for US English on Linux.

Time variables

Variable	Description
%Ez	Splunk-specific, timezone in minutes.
%f	Microseconds as a decimal number.
%H	Hour (24-hour clock) as a decimal number. Hours are represented by the values 00 to 23. Leading zeros are accepted but not required.
%I	Uppercase "i". Hour (12-hour clock) with the hours represented by the values 01 to 12. Leading zeros are accepted but not required. Use with %p to specify AM or PM for the 12-hour clock.
%k	Like %H, the hour (24-hour clock) as a decimal number. Leading zeros are replaced by a space, for example 0 to 23.
%M	Minute as a decimal number. Minutes are represented by the values 00 to 59. Leading zeros are accepted but not required.

Variable	Description
%N	The number of subsecond digits. The default is %9N. You can specify %3N = milliseconds, %6N = microseconds, %9N = nanoseconds.
%p	AM or PM. Use with %l to specify the 12-hour clock for AM or PM. Do not use with %H.
%Q	The subsecond component of a UTC timestamp. The default is milliseconds, %3Q. Valid values are: <ul style="list-style-type: none"> • %3Q = milliseconds, with values of 000-999 • %6Q = microseconds, with values of 000000-999999 • %9Q = nanoseconds, with values of 000000000-999999999
%S	Second as a decimal number, for example 00 to 59.
%s	The UNIX Epoch Time timestamp, or the number of seconds since the Epoch: 1970-01-01 00:00:00 +0000 (UTC). For example the UNIX epoch time 1484993700 is equal to Tue Jan 21 10:15:00 2020.
%T	The time in 24-hour notation (%H:%M:%S). For example 23:59:59.
%X	The time in the format for the current locale. For US English the format for 9:30 AM is 9:30:00.
%Z	The timezone abbreviation. For example EST for US Eastern Standard Time.
%z	The timezone offset from UTC, in hour and minute: +hhmm or -hhmm. For example, for 5 hours before UTC the value is -0500 which is US Eastern Standard Time. Examples: <ul style="list-style-type: none"> • Use %z to specify hour and minute, for example -0500 • Use %:z to specify hour and minute separated by a colon, for example -05:00 • Use %::z to specify hour minute and second separated with colons, for example -05:00:00 • Use %:::z to specify hour only, for example -05
%%	A literal "%" character.

Date variables

Variable	Description
%F	Equivalent to %Y-%m-%d (the ISO 8601 date format).
%x	The date in the format of the current locale. For example, 7/13/2019 for US English.

Specifying days and weeks

Variable	Description
%A	Full weekday name. (Sunday, ..., Saturday)
%a	Abbreviated weekday name. (Sun, ... ,Sat)
%d	Day of the month as a decimal number, includes a leading zero. (01 to 31)
%e	Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space. (1 to 31)
%j	Day of year as a decimal number, includes a leading zero. (001 to 366)
%V (or %U)	Week of the year. The %V variable starts the count at 1, which is the most common start number. The %U variable starts the count at 0.
%w	Weekday as a decimal number. (0 = Sunday, ..., 6 = Saturday)

Specifying months

Variable	Description
%b	Abbreviated month name. (Jan, Feb, etc.)
%B	Full month name. (January, February, etc.)
%m	Month as a decimal number. (01 to 12). Leading zeros are accepted but not required.

Specifying year

Variable	Description
%y	Year as a decimal number, without the century. (00 to 99). Leading zeros are accepted but not required.
%Y	Year as a decimal number with century. For example, 2020.

Examples

Converting UNIX timestamps into dates

The following table shows the results of several date format variables, using the `strftime` function. These examples show the results when you use the `strftime` function with the date Fri Apr 29 2022 23:45:22 GMT-0700 (Pacific Daylight Time).

Date format string	Result
%Y-%m-%d	2022-04-29
%y-%m-%d	22-04-29
%b %d, %Y	Apr 29, 2022
%B %d, %Y	April 29, 2022
%a %b %d, %Y	Fri Apr 29, 2022
%d %b '%y = %Y-%m-%d	29 Apr '22 = 2022-04-29

Converting UNIX timestamps into dates and times

The following table shows the results of several date time format variables, using the `strftime` function. These examples show the results when you use the `strftime` function with the date Fri Apr 29 2022 23:45:22 GMT-0700 (Pacific Daylight Time).

Date and Time format string	Result
%Y-%m-%dT%H:%M:%S.%Q	2022-04-29T23:45:22.000
%Y-%m-%dT %H:%M:%S.%Z	2022-04-29T 23:45:22.PDT
%Y-%m-%dT %H:%M:%S.%QZ	2022-04-29T 23:45:22.000Z
%Y-%m-%dT%H:%M:%S.%QZ	2022-04-29T23:45:22.000Z
%Y-%m-%dT%H:%M:%S	2022-04-29T23:45:22
%Y-%m-%dT%T	2022-04-29T23:45:22
%m-%d-%Y %I:%M:%S %p	04-29-2022 11:45:22 PM
%b %d, %Y %I:%M:%S %p	Apr 29, 2022 11:45:22 PM

Date and Time format string	Result
%m-%d-%Y %H:%M:%S.%Q	04-29-2022 23:45:22.000
%m-%d-%Y %H:%M:%S.%Q %z	04-29-2022 23:45:22.000 -0700
%d/%b/%Y:%H:%M:%S.%f %z	29/Apr/2022:23:45:22.000000 -0700

Converting timestamps into UNIX

The following table shows the results of using several date time format variables to timestampls into UNIX time using the `strptime` function.

For example, this search returns the UNIX time 1671126322.000000.

```
... | eval mytime=strptime("2022-12-15T09:45:22", "%Y-%m-%dT%H:%M:%S")
```

Timestamps	Date and Time format string	UNIX time
2022-9-25T09:45:22.000	%Y-%m-%dT%H:%M:%S.%Q	1664124322.000000
2022-12-15 09:45:22	%Y-%m-%d %H:%M:%S	1671126322.000000

The following table shows the results of searches that use time variables:

Sample search	Result
host="www1" eval WeekNo = strftime(_time, "%V")	Creates a field called <code>WeekNo</code> and returns the values for the week numbers that correspond to the dates in the <code>_time</code> field.
... eval mytime=strftime(_time, "%Y-%m-%dT%H:%M:%S.%Q")	Creates a field called <code>mytime</code> and returns the converted timestamp values in the <code>_time</code> field. The values are stored in UNIX format and converted using the format specified, which is the ISO 8601 format. For example: 2021-04-13T14:00:15.000.
... eval start=strptime(Sent, "%H:%M:%S.%N"), end=strptime(Received, "%H:%M:%S.%N") eval difference=end-start table end, start, difference	Takes the values in the Sent and Received fields and converts them into a standard time using the <code>strptime</code> function. Then calculates the difference between the start and end times. The results are displayed in a table. You can use the <code>round</code> function to round the difference to a specific number of decimal places. For example ... eval difference=round(end-start, 2).

Time modifiers

Use time modifiers to customize the time range of a search or change the format of the timestamps in the search results.

Searching the `_time` field

When an event is processed by Splunk software, its timestamp is saved as the default field `_time`. This timestamp, which is the time when the event occurred, is saved in UNIX time notation. Searching with relative time modifiers, `earliest` or `latest`, finds every event with a timestamp beginning, ending, or between the specified timestamps.

For example, when you search for `earliest=@d`, the search finds every event with a `_time` value since midnight. This example uses `@d`, which is a date format variable. See [Date and time format variables](#).

Time modifiers and the Time Range Picker

When you use a time modifier in the SPL syntax, that time overrides the time specified in the Time Range Picker.

For example, suppose your search uses `yesterday` in the Time Range Picker. You add the time modifier `earliest=-2d` to your search syntax. The search uses the time specified in the time modifier and ignores the time in the Time Range Picker. Because the search does not specify the `latest` time modifier, the default value `now` is used for `latest`.

For more information, see [Specify time modifiers in your search](#) in the *Search Manual*.

Time ranges and subsearches

Time ranges selected from the Time Range Picker apply to the base search and to subsearches.

However, time ranges specified directly in the base search do not apply to subsearches. Likewise, a time range specified directly in a subsearch applies only to that subsearch. The time range does not apply to the base search or any other subsearch.

For example, if the Time Range Picker is set to **Last 7 days** and a subsearch contains `earliest=2d@d`, then the earliest time modifier applies only to the subsearch and **Last 7 days** applies to the base search.

Searching based on index time

You also have the option of searching for events based on when they were indexed. The UNIX time is saved in the `_indextime` field. Similar to `earliest` and `latest` for the `_time` field, you can use the relative time modifiers `_index_earliest` and `_index_latest` to search for events based on `_indextime`. For example, if you wanted to search for events indexed in the previous hour, use: `_index_earliest=-h@h _index_latest=@h`.

When using index-time based modifiers such as `_index_earliest` and `_index_latest`, your search must also have an event-time window which will retrieve the events. In other words, chunks of events might be ruled out based on the non index-time window as well as the index-time window. To be certain of retrieving every event based on index-time, you must run your search using All Time.

List of time modifiers

Use the `earliest` and `latest` modifiers to specify custom and relative time ranges. You can specify an exact time such as `earliest="10/5/2016:20:00:00"`, or a relative time such as `earliest=-h` or `latest=@w6`.

When specifying relative time, you can use the `now` modifier to refer to the current time.

Modifier	Syntax	Description
earliest	<code>earliest=[+ -]<time_integer><time_unit>@<time_unit></code>	Specify the earliest <code>_time</code> for the time range of your search. Use <code>earliest=1</code> to specify the UNIX epoch time 1, which is UTC January 1, 1970 at 12:00:01 AM.

Modifier	Syntax	Description
		Use <code>earliest=0</code> to specify the earliest event in your data.
<code>_index_earliest</code>	<code>_index_earliest=[+ -]<time_integer><time_unit>@<time_unit></code>	Specify the earliest <code>_indextime</code> for the time range of your search.
<code>_index_latest</code>	<code>_index_latest=[+ -]<time_integer><time_unit>@<time_unit></code>	Specify the latest <code>_indextime</code> for the time range of your search.
<code>latest</code>	<code>latest=[+ -]<time_integer><time_unit>@<time_unit></code>	Specify the latest time for the <code>_time</code> range of your search.
<code>now</code>	<code>now() or now</code>	Refers to the current time. If set to earliest, <code>now()</code> is the start of the search.
<code>time</code>	<code>time()</code>	In real-time searches, <code>time()</code> is the current machine time.

For more information about customizing your search window, see [Specify real-time time range windows in your search](#) in the *Search Manual*.

How to specify relative time modifiers

You can define the relative time in your search with a string of characters that indicate time amount (integer and unit). You can also specify a "snap to" time unit, which is specified with the @ symbol followed by a time unit.

The syntax for using time modifiers is `[+|-]<time_integer><time_unit>@<time_unit>`

The steps to specify a relative time modifier are:

1. Indicate the time offset from the current time.
2. Define the time amount, which is a number and a unit.
3. Specify a "snap to" time unit. The time unit indicates the nearest or latest time to which your time amount rounds down.

When a relative time modifier is processed, the offset is processed first, followed by the snap-to time. For example, if the relative time modifier is `-2h@h` the offset `-2h` is processed first. Then the snap-to time `@h` is processed.

Indicate the time offset

Begin your time offset with a plus (+) or minus (-) to indicate the offset from the current time.

Define the time amount

Define your time amount with a number and a unit. The supported time units are listed in the following table:

Time unit	Valid unit abbreviations
subseconds	microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)
second	s, sec, secs, second, seconds
minute	m, min, mins, minute, minutes
hour	h, hr, hrs, hour, hours

Time unit	Valid unit abbreviations
day	d, day, days
week	w, week, weeks
month	mon, month, months
quarter	q, qtr, qtrs, quarter, quarters
year	y, yr, yrs, year, years

For example, to start your search an hour ago, use either of the following time modifiers.

`earliest=-h`

or

`earliest=-60m`

When specifying single time amounts, the number one is implied. An 's' is the same as '1s', 'm' is the same as '1m', 'h' is the same as '1h', and so forth.

Subsecond timescales such as ms can be used in metrics searches only when they are searching over metrics indexes that are enabled for millisecond timestamp resolution.

For more information about enabling metrics indexes to index metric data points with millisecond timestamp precision:

- For Splunk Cloud Platform, see Manage Splunk Cloud Platform indexes in the *Splunk Cloud Platform Admin Manual*.
- For Splunk Enterprise, see Create custom indexes in *Managing indexers and clusters of indexers*.

Specify a snap to time unit

You can specify a snap to time unit. The time unit indicates the nearest or latest time to which your time amount rounds down. Separate the time amount from the "snap to" time unit with an "@" character.

- You can use any of time units listed previously. For example:
 - ◆ @w, @week, and @w0 for Sunday
 - ◆ @month for the beginning of the month
 - ◆ @q, @qtr, or @quarter for the beginning of the most recent quarter (Jan 1, Apr 1, Jul 1, or Oct 1).
- You can specify a day of the week: w0 (Sunday), w1, w2, w3, w4, w5 and w6 (Saturday). For Sunday, you can specify w0 or w7.
- You can also specify **offsets from the snap-to-time** or "chain" together the time modifiers for more specific relative time definitions. For example, `@d-2h` snaps to the beginning of today (12 AM or midnight), and then applies the time offset of -2h. This results in a time of 10 PM yesterday.
 - ◆ The Splunk platform always applies the offset before it applies the snap. In other words, the left-hand side of the @ symbol is applied before the right-hand side.
- When snapping to the nearest or latest time, Splunk software always **snaps backwards** or rounds down to the latest time not after the specified time. For example, if it is 11:59:00 and you "snap to" hours, you will snap to 11:00 not 12:00.
- If you do not specify a time offset before the "snap to" amount, Splunk software interprets the time as "current time snapped to" the specified amount. For example, if it is currently 11:59 PM on Friday and you use `@w6` to "snap to Saturday", the resulting time is the *previous* Saturday at 12:01 A.M.

Examples

1. Run a search over all time

If you want to search events from the start of UNIX time, use `earliest=1`.

When `earliest=1` and `latest=now()` are used, the search runs over all time.

```
...earliest=1 latest=now()
```

Specifying `latest=now()` does not return future events.

To return future events, specify `latest=<a_big_number>`. Future events are events that contain timestamps later than the current time `now()`.

2. Search the events from the beginning of the current week

```
...earliest=@w0
```

3. Search the events from the last full business week

```
...earliest=-5d@w1 latest=@w6
```

4. Search with an exact date as a boundary

With a boundary such as from November 15 at 8 PM to November 22 at 8 PM, use the timeformat `%m/%d/%Y:%H:%M:%S`.

```
...earliest="11/15/2022:20:00:00" latest="11/22/2022:20:00:00"
```

5. Specify multiple time windows using a fixed date time format

You can specify multiple time windows using the timeformat `%m/%d/%Y:%H:%M:%S`. For example to find events from 5-6 PM or 7-8 PM on specific dates, use the following syntax.

```
... (earliest="9/23/2022:17:00:00" latest="9/23/2022:18:00:00") OR (earliest="9/23/2022:19:00:00"  
latest="9/23/2022:20:00:00")
```

6. Specify multiple time windows using a relative time format

You can specify multiple time windows using the time modifiers and snap-to with a relative time. For example to find events for the last 24 hours but omit the events from Midnight to 1:00 A.M., use the following syntax:

```
... ((earliest=-24h latest<@d) OR (earliest>=@d+1h))
```

Other time modifiers

The following search time modifiers are still valid, but might be removed and their function no longer supported in a future release.

Modifier	Syntax	Description
----------	--------	-------------

Modifier	Syntax	Description
daysago	daysago=<int>	Search events within the last integer number of days.
enddaysago	enddaysago=<int>	Set an end time for an integer number of days before Now.
endhoursago	endhoursago=<int>	Set an end time for an integer number of hours before Now.
endminutesago	endminutesago=<int>	Set an end time for an integer number of minutes before Now.
endmonthsago	endmonthsago=<int>	Set an end time for an integer number of months before Now.
endtime	endtime=<string>	Search for events before the specified time (exclusive of the specified time). Use timeformat to specify how the timestamp is formatted.
endtimeu	endtimeu=<int>	Search for events before the specific UNIX time.
hoursago	hoursago=<int>	Search events within the last integer number of hours.
minutesago	minutesago=<int>	Search events within the last integer number of minutes.
monthsago	monthsago=<int>	Search events within the last integer number of months.
searchtimespandays	searchtimespandays=<int>	Search within a specified range of days, expressed as an integer.
searchtimespanhours	searchtimespanhours=<int>	Search within a specified range of hours, expressed as an integer.
searchtimespanminutes	searchtimespanminutes=<int>	Search within a specified range of minutes, expressed as an integer.
searchtimespanmonths	searchtimespanmonths=<int>	Search within a specified range of months, expressed as an integer.
startdaysago	startdaysago=<int>	Search the specified number of days before the present time.
starthoursago	starthoursago=<int>	Search the specified number of hours before the present time.
startminutesago	startminutesago=<int>	Search the specified number of minutes before the present time.
startmonthsago	startmonthsago=<int>	Search the specified number of months before the present time.
starttime	starttime=<timestamp>	Search from the specified date and time to the present, inclusive of the specified time.
starttimeu	starttimeu=<int>	Search for events starting from the specific UNIX time.
timeformat	timeformat=<string>	Set the timeformat for the starttime and endtime modifiers. By default: timeformat=%m/%d/%Y:%H:%M:%S

See also

Functions

[Date and time functions](#) used with evaluation commands

[Time functions](#) used with statistical and charting commands

Related information

Specify time modifiers in your search in the *Search Manual*

Search Commands

abstract

Description

Produces an abstract, a summary or brief representation, of the text of the search results. The original text is replaced by the summary.

The abstract is produced by a scoring mechanism. Events that are larger than the selected `maxlines`, those with more textual terms and more terms on adjacent lines, are preferred over events with fewer terms. If a line has a search term, its neighboring lines also partially match, and might be returned to provide context. When there are gaps between the selected lines, lines are prefixed with an ellipsis (...).

If the text of an event has fewer lines or an equal number of lines as `maxlines`, no change occurs.

Syntax

The required syntax is in **bold**.

```
abstract
[maxterms=<int>]
[maxlines=<int>]
```

Optional arguments

`maxterms`

Syntax: `maxterms=<int>`
Description: The maximum number of terms to match. Accepted values are 1 to 1000.
Default: 1000

`maxlines`

Syntax: `maxlines=<int>`
Description: The maximum number of lines to match. Accepted values are 1 to 500.
Default: 10

Examples

Specify the number of lines to return

Show a summary of up to 5 lines for each search result.

```
... | abstract maxlines=5
```

Specify the number of terms to return

Consider the following events:

Time	Event
1/4/23 6:22:16.000 PM	91.205.189.15 - - [04/Jan/2023:18:22:16] "GET /oldlink?itemId=EST-14&JSESSIONID=SD6SL7FF7ADFF53113 HTTP 1.1" 200 1665 "http://www.buttercupgames.com/oldlink?itemId=EST-14" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 159
1/3/23 11:08:57.000 PM	194.146.236.22 - - [03/Jan/2023:23:08:57] "POST /cart.do?action=addtocart&itemId=EST-15&productId=WC-SH-T02&JSESSIONID=SD4SL1FF2ADFF47548 HTTP 1.1" 200 1493 "http://www.buttercupgames.com/product.screen?productId=WC-SH-T02" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5" 848

If you specify `maxterms=20` the results look like this:

Time	Event
1/4/23 6:22:16.000 PM	91.205.189.15 - - [04/Jan/2023:18:22:16] "GET /oldlink?itemId=EST-14&JSESSIONID=SD6SL7FF7ADFF53113 HTTP 1.1" 200 1665 "http://www.buttercupgames.com/oldlink?itemId=EST-14" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 159
1/3/23 11:08:57.000 PM	194.146.236.22 - - [03/Jan/2023:23:08:57] "POST /cart.do?action=addtocart&itemId=EST-15&productId=WC-SH-T02&JSESSIONID=SD4SL1FF2ADFF47548 HTTP 1.1" 200 1493 "http://www.buttercupgames.com/product.screen?productId=WC-SH-T02" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5" 848

The "terms" are identified as shown in the following table:

Number	Event 1 term	Event 2 term
1	91	194
2	.	.
3	205	146
4	.	.
5	189	236
6	.	.
7	15	22
8		
9	-	-
10		
11	-	-
12		
13	[[
14	04	03
15	/	/
16	Jan	Jan
17	/	/
18	2023	2023

Number	Event 1 term	Event 2 term
19	:	:
20	18	23

See also

[highlight](#)

accum

Description

For each event where `field` is a number, the `accum` command calculates a running total or sum of the numbers. The accumulated sum can be returned to either the same field, or a `newfield` that you specify.

Syntax

`accum <field> [AS <newfield>]`

Required arguments

`field`

Syntax: `<string>`

Description: The name of the field that you want to calculate the accumulated sum for. The field must contain numeric values.

Optional arguments

`newfield`

Syntax: `<string>`

Description: The name of a new field where you want the results placed.

Basic example

1. Create a running total of a field

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 **All time** when you run the search.

The following search looks for events from web access log files that were successful views of strategy games. A count of the events by each product ID is returned.

```
sourcetype=access_* status=200 categoryId=STRATEGY | chart count AS views by productId
```

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

productId	views
DB-SG-G01	1796

productId	views
DC-SG-G02	1642
FS-SG-G03	1482
PZ-SG-G05	1300

You can use the `accum` command to generate a running total of the views and display the running total in a new field called "TotalViews".

```
sourcetype=access_* status=200 categoryId=STRATEGY | chart count AS views by productId | accum views as TotalViews
```

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

productId	views	TotalViews
DB-SG-G01	1796	1796
DC-SG-G02	1642	3438
FS-SG-G03	1482	4920
PZ-SG-G05	1300	6220

See also

[autoregress](#), [delta](#), [streamstats](#), [trendline](#)

addcoltotals

Description

The `addcoltotals` command appends a new result to the end of the search result set. The result contains the sum of each numeric field or you can specify which fields to summarize. Results are displayed on the Statistics tab. If the `labelfield` argument is specified, a column is added to the statistical results table with the name specified.

Syntax

```
addcoltotals [labelfield=<field>] [label=<string>] [<wc-field-list>]
```

Optional arguments

<wc-field-list>

Syntax: <field> ...

Description: A space delimited list of valid field names. The `addcoltotals` command calculates the sum only for the fields in the list you specify. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Default: Calculates the sum for all of the fields.

labelfield

Syntax: labelfield=<fieldname>

Description: Specify a field name to add to the result set.

Default: none

label

Syntax: label=<string>

Description: Used with the `labelfield` argument to add a label in the summary event. If the `labelfield` argument is absent, the `label` argument has no effect.

Default: Total

Basic examples

1. Compute the sums of all the fields

Compute the sums of all the fields, and put the sums in a summary event called "change_name".

```
... | addcoltotals labelfield=change_name label=ALL
```

2. Add a column total for two specific fields

Add a column total for two specific fields in a table.

```
sourcetype=access_* | table userId bytes avgTime duration | addcoltotals bytes duration
```

3. Create the totals for a field that match a field name pattern

Filter fields for two name-patterns, and get totals for one of them.

```
... | fields user*, *size | addcoltotals *size
```

4. Specify a field name for the column totals

Augment a chart with a total of the values present.

```
index=_internal source="metrics.log" group=pipeline | stats avg(cpu_seconds) by processor | addcoltotals labelfield=processor
```

Extended example

1. Generate a total for a column

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 **All time** when you run the search.

The following search looks for events from web access log files that were successful views of strategy games. A count of the events by each product ID is returned.

```
sourcetype=access_* status=200 categoryId=STRATEGY | chart count AS views by productId
```

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

productId	views
-----------	-------

productId	views
DB-SG-G01	1796
DC-SG-G02	1642
FS-SG-G03	1482
PZ-SG-G05	1300

You can use the `addcoltotals` command to generate a total of the views and display the total at the bottom of the column.

```
sourcetype=access_* status=200 categoryId=STRATEGY | chart count AS views by productId | addcoltotals
```

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

productId	views
DB-SG-G01	1796
DC-SG-G02	1642
FS-SG-G03	1482
PZ-SG-G05	1300
	6220

You can use `add` to the results that labels the total.

```
sourcetype=access_* status=200 categoryId=STRATEGY | chart count AS views by productId | addcoltotals  
labelfield="Total views"
```

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

productId	views	Total views
DB-SG-G01	1796	
DC-SG-G02	1642	
FS-SG-G03	1482	
PZ-SG-G05	1300	
	6220	Total

See also

Commands

[addtotals](#)
[stats](#)

addinfo

Description

Adds fields to each event that contain global, common information about the search. This command is primarily an internally-used component of Summary Indexing.

Syntax

addinfo

The following fields are added to each event when you use the `addinfo` command.

Field	Description
<code>info_min_time</code>	The earliest time boundary for the search.
<code>info_max_time</code>	The latest time boundary for the search.
<code>info_sid</code>	The ID of the search that generated the event.
<code>info_search_time</code>	The time when the search was run.

Usage

The `addinfo` command is a distributable streaming command. See [Command types](#).

Examples

1. Add information to each event

Add information about the search to each event.

```
... | addinfo
```

2. Determine which heartbeats are later than expected

You can use this example to track heartbeats from hosts, forwarders, `tcpin_connections` on indexers, or any number of system components. This example uses hosts.

You have a list of host names in a lookup file called `expected_hosts`. You want to search for heartbeats from your hosts that are after an expected time range. You use the `addinfo` command to add information to each event that will help you evaluate the time range.

```
... | stats latest(_time) AS latest_time BY host | addinfo | eval latest_age = info_max_time - latest_time | fields - info_* | inputlookup append=t expected_hosts | fillnull value=9999 latest_age | dedup host | where latest_age > 42
```

Use the `stats` command to calculate the latest heartbeat by host. The `addinfo` command adds information to each result. This search uses `info_max_time`, which is the latest time boundary for the search. The `eval` command is used to create a field called `latest_age` and calculate the age of the heartbeats relative to end of the time range. This allows for a time range of `-11m@m to -m@m`. This is the previous 11 minutes, starting at the beginning of the minute, to the previous 1 minute, starting at the beginning of the minute. The search does not work if you specify `latest=null / all time` because `info_max_time` would be set to `+infinity`.

Using the lookup file, `expected_hosts`, append the list of hosts to the results. Using this list you can determine which hosts are not sending a heartbeat in the expected time range. For any hosts that have a null value in the `latest_age` field, fill the field with the value 9999. Remove any duplicated host events with the `dedup` command. Use the `where` command to filter the results and return any heartbeats older than 42 seconds.

In this example, you could use the `tstats` command, instead of the `stats` command, to improve the performance of the search.

See also

[search](#)

addtotals

Description

The `addtotals` command computes the arithmetic sum of all numeric fields for each search result. The results appear in the Statistics tab.

You can specify a list of fields that you want the sum for, instead of calculating every numeric field. The sum is placed in a new field.

If `col=true`, the `addtotals` command computes the column totals, which adds a new result at the end that represents the sum of each field. `labelfield`, if specified, is a field that will be added to this summary event with the value set by the 'label' option. Alternately, instead of using the `addtotals col=true` command, you can use the `addcoltotals` command to calculate a summary event.

Syntax

```
addtotals [row=<bool>] [col=<bool>] [labelfield=<field>] [label=<string>] [fieldname=<field>] [<field-list>]
```

Required arguments

None.

Optional arguments

`field-list`

Syntax: `<field> ...`

Description: One or more numeric fields, delimited with a space. Only the fields specified in the `<field-list>` are summed. If a `<field-list>` is not specified, all numeric fields are included in the sum.

Usage: You can use wildcards in the field names. For example, if the field names are `count1`, `count2`, and `count3` you can specify `count*` to indicate all fields that begin with 'count'.

Default: All numeric fields are included in the sum.

`row`

Syntax: `row=<bool>`

Description: Specifies whether to calculate the sum of the `<field-list>` for each event. This is similar to calculating a total for each row in a table. The sum is placed in a new field. The default name of the field is `Total`. If you want to specify a different name for the field, use the `fieldname` argument.

Usage: Because the default is `row=true`, specify the `row` argument only when you do not want the event totals to appear `row=false`.

Default: `true`

col

Syntax: col=<bool>

Description: Specifies whether to add a new event, referred to as a summary event, at the bottom of the list of events. The summary event displays the sum of each field in the events, similar to calculating column totals in a table.

Default: false

fieldname

Syntax: fieldname=<field>

Description: Used to specify the name of the field that contains the calculated sum of the *field-list* for each event. The `fieldname` argument is valid only when `row=true`.

Default: Total

labelfield

Syntax: labelfield=<field>

Description: Used to specify a field for the summary event label. The `labelfield` argument is valid only when `col=true`.

* To use an existing field in your result set, specify the field name for the `labelfield` argument. For example if the field name is `IP`, specify `labelfield=IP`.

* If there is no field in your result set that matches the `labelfield`, a new field is added using the `labelfield` value.

Default: none

label

Syntax: label=<string>

Description: Used to specify a row label for the summary event.

* If the `labelfield` argument is an existing field in your result set, the `label` value appears in that row in the display.

* If the `labelfield` argument creates a new field, the `label` appears in the new field in the summary event row.

Default: Total

Usage

The `addtotals` command is a distributable streaming command, except when used to calculate column totals. When used to calculate column totals, the `addtotals` command is a transforming command. See [Command types](#).

Examples

1: Calculate the sum of the numeric fields of each event

This example uses events that list the numeric sales for each product and quarter, for example:

products	quarter	sales	quota
ProductA	QTR1	1200	1000
ProductB	QTR1	1400	1550
ProductC	QTR1	1650	1275
ProductA	QTR2	1425	1300
ProductB	QTR2	1175	1425

products	quarter	sales	quota
ProductC	QTR2	1550	1450
ProductA	QTR3	1300	1400
ProductB	QTR3	1250	1125
ProductC	QTR3	1375	1475
ProductA	QTR4	1550	1300
ProductB	QTR4	1700	1225
ProductC	QTR4	1625	1350

Use the chart command to summarize data

To summarize the data by product for each quarter, run this search:

```
source="addtotalsData.csv" | chart sum(sales) BY products quarter
```

In this example, there are two fields specified in the BY clause with the `chart` command.

- The `products` field is referred to as the <row-split> field.
- The `quarter` field is referred to as the <column-split> field.

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

products	QTR1	QTR2	QTR3	QTR4
ProductA	1200	1425	1300	1550
ProductB	1400	1175	1250	1700
ProductC	1650	1550	1375	1625

To add a column that generates totals for each row, run this search:

```
source="addtotalsData.csv" | chart sum(sales) BY products quarter | addtotals
```

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

products	QTR1	QTR2	QTR3	QTR4	Total
ProductA	1200	1425	1300	1550	5475
ProductB	1400	1175	1250	1700	5525
ProductC	1650	1550	1375	1625	6200

Use the stats command to calculate totals

If all you need are the totals for each product, a simpler solution is to use the `stats` command:

```
source="addtotalsData.csv" | stats sum(sales) BY products
```

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

products	sum(sales)
ProductA	5475

products	sum(sales)
ProductA	5475
ProductB	5525
ProductC	6200

2. Specify a name for the field that contains the sums for each event

Instead of accepting the default name added by the `addtotals` command, you can specify a name for the field.

```
... | addtotals fieldname=sum
```

3. Use wildcards to specify the names of the fields to sum

Calculate the sums for the fields that begin with `amount` or that contain the text `size` in the field name. Save the sums in the field called `TotalAmount`.

```
... | addtotals fieldname=TotalAmount amount* *size*
```

4. Calculate the sum for a specific field

In this example, the row calculations are turned off and the column calculations are turned on. The total for only a single field, `sum(quota)`, is calculated.

```
source="addtotalsData.csv" | stats sum(quota) by quarter| addtotals row=f col=t labelfield=quarter sum(quota)
```

- The `labelfield` argument specifies in which field the label for the total appears. The default label is **Total**.

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

quarter	sum(quota)
QTR1	3825
QTR2	4175
QTR3	4000
QTR4	3875
Total	15875

5. Calculate the field totals and add custom labels to the totals

Calculate the sum for each quarter and product, and calculate a grand total.

```
source="addtotalsData.csv" | chart sum(sales) by products quarter| addtotals col=t labelfield=products
label="Quarterly Totals" fieldname="Product Totals"
```

- The `labelfield` argument specifies in which field the label for the total appears, which in this example is **products**.
- The `label` argument is used to specify the label **Quarterly Totals** for the `labelfield`, instead of using the default label **Total**.
- The `fieldname` argument is used to specify the label **Product Totals** for the row totals.

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

products	QTR1	QTR2	QTR3	QTR4	Product Totals
ProductA	1200	1425	1300	1550	5475
ProductB	1400	1175	1250	1700	5525
ProductC	1650	1550	1375	1625	6200
Quarterly Totals	4250	4150	3925	4875	17200

See also

[stats](#)

analyzefields

Description

Using `<field>` as a discrete random variable, this command analyzes all numerical fields to determine the ability for each of those fields to predict the value of the `classfield`. It determines the stability of the relationship between values in the target `classfield` and numeric values in other fields.

As a reporting command, `analyzefields` consumes all input results and generates one row for each numeric field in the output results. The values in that row indicate the performance of the `analyzefields` command at predicting the value of a `classfield`. For each event, if the conditional distribution of the numeric field with the highest z-probability based on matches the actual class, the event is counted as accurate. The highest z-probability is based on the `classfield`.

Syntax

`analyzefields classfield=<field>`

You can use the abbreviation `af` for the `analyzefields` command.

The `analyzefields` command returns a table with five columns.

Field	Description
<code>field</code>	The name of a numeric field from the input search results.
<code>count</code>	The number of occurrences of the field in the search results.
<code>coocur</code>	The co-occurrence of the field. In the results where <code>classfield</code> is present, this is the ratio of results in which <code>field</code> is also present. The <code>coocur</code> is 1 if the <code>field</code> exists in every event that has a <code>classfield</code> .
<code>acc</code>	The accuracy in predicting the value of the <code>classfield</code> , using the value of the field. This is the ratio of the number of accurate predictions to the total number of events with that field. This argument is valid only for numerical fields.
<code>balacc</code>	The balanced accuracy is the non-weighted average of the accuracies in predicted each value of the <code>classfield</code> . This is only valid for numerical fields.

Required arguments

classfield

Syntax: classfield=<field>

Description: For best results, `classfield` should have two distinct values, although multiclass analysis is possible.

Examples

Example 1:

Analyze the numerical fields to predict the value of "is_activated".

```
... | analyzefields classfield=is_activated
```

See also

[anomalousvalue](#)

anomalies

Description

Use the `anomalies` command to look for events or field values that are unusual or unexpected.

The `anomalies` command assigns an unexpectedness score to each event and places that score in a new field named `unexpectedness`. Whether the event is considered anomalous or not depends on a `threshold` value. The `threshold` value is compared to the unexpectedness score. The event is considered unexpected or anomalous if the unexpectedness score is greater than the `threshold` value.

After you use the `anomalies` command in a search, look at the **Interesting Fields** list in the Search & Reporting window. Select the `unexpectedness` field to see information about the values in your events.

The unexpectedness score of an event is calculated based on the similarity of that event (X) to a set of previous events (P).

The formula for unexpectedness is:

$$\text{unexpectedness} = \frac{[s(P \text{ and } X) - s(P)]}{[s(P) + s(X)]}$$

In this formula, `s()` is a metric of how similar or uniform the data is. This formula provides a measure of how much adding X affects the similarity of the set of events. The formula also normalizes the results for the differing event sizes.

Syntax

The required syntax is in **bold**.

anomalies
[threshold=<num>]

[labelonly=<bool>
[normalize=<bool>
[maxvalues=<num>
[field=<field>
[denylist=<filename>
[denylistthreshold=<num>
[by-clause]

Optional arguments

threshold

Syntax: threshold=<num>

Description: A number to represent the upper limit of expected or normal events. If unexpectedness calculated for an event is greater than this threshold limit, the event is considered unexpected or anomalous.

Default: 0.01

labelonly

Syntax: labelonly=<bool>

Description: Specifies if you want the output result set to include all events or only the events that are above the threshold value. The `unexpectedness` field is appended to all events. If `labelonly=true`, no events are removed. If `labelonly=false`, events that have a unexpectedness score less than the threshold are removed from the output result set.

Default: false

normalize

Syntax: normalize=<bool>

Description: Specifies whether or not to normalize numeric text in the fields. All characters in the field from 0 to 9 are considered identical for purposes of the algorithm. The placement and quantity of the numbers remains significant. When a field contains numeric data that should not be normalized but treated as categories, set `normalize=false`.

Default: true

maxvalues

Syntax: maxvalues=<num>

Description: Specifies the size of the sliding set of previous events to include when determining the unexpectedness of a field value. By default the calculation uses the previous 100 events for the comparison. If the current event number is 1000, the calculation uses the values in events 900 to 999 in the calculation. If the current event number is 1500, the calculation uses the values in events 1400 to 1499 in the calculation. You can specify a number between 10 and 10000. Increasing the value of `maxvalues` increases the total CPU cost per event linearly. Large values have very long search runtimes.

Default: 100

field

Syntax: field=<field>

Description: The field to analyze when determining the unexpectedness of an event.

Default: `_raw`

denylist

Syntax: denylist=<filename>

Description: The name of a CSV file that contains a list of events that are expected and should be ignored. Any incoming event that is similar to an event in the denylist is treated as not anomalous, or expected, and given an unexpectedness score of 0.0. The CSV file must be located in the `$SPLUNK_HOME/var/run/splunk/csv` directory on

the search head. If you have Splunk Cloud Platform and want to configure a denylist file, file a Support ticket.

denylistthreshold

Syntax: denylistthreshold=<num>

Description: Specifies a similarity score threshold for matching incoming events to denylisted events. If the incoming event has a similarity score above the `denylistthreshold`, the event is marked as unexpected.

Default: 0.05

by-clause

Syntax: by <fieldlist>

Description: Use to specify a list of fields to segregate the results for anomaly detection. For each combination of values for the specified fields, the events with those values are treated entirely separately.

Examples

1. Specify a denylist file of the events to ignore

The following example shows the interesting events, ignoring any events in the denylist 'boringevents'. Sort the event list in descending order, with highest value in the unexpectedness field listed first.

```
... | anomalies denylist=boringevents | sort -unexpectedness
```

2. Find anomalies in transactions

This example uses transactions to find regions of time that look unusual.

```
... | transaction maxpause=2s | anomalies
```

3. Identify anomalies by source

Look for anomalies in each source separately. A pattern in one source does not affect that it is anomalous in another source.

```
... | anomalies by source
```

4. Specify a threshold when identifying anomalies

This example shows how to tune a search for anomalies using the `threshold` value. Start with a search that uses the default `threshold` value.

```
index=_internal | anomalies BY group | search group=*
```

This search looks at events in the `_internal` index and calculates an `unexpectedness` score for sets of events that have the same `group` value.

- The sliding set of events that are used to calculate the `unexpectedness` score for each unique `group` value includes only the events that have the same `group` value.
- The `search` command is used to show events that only include the `group` field.

The `unexpectedness` and `group` fields appear in the list of **Interesting fields**. Click on the field name and then click **Yes** to move the field to the **Selected fields** list. The fields are moved and also appear in the search results. Your results should look something like the following image.

List		Format	20 Per Page	< Prev	1	2	3	4	5	6	7	8	... Next >
« Hide Fields	All Fields	i	Time	Event									
SELECTED FIELDS				> 3/16/18 03-16-2018 21:24:30.181 -0700 INFO Metrics - group=pipeline, name=indexerpipe, processor=indexer, cpu_seconds=0.033983, executes=187, cumulative_hits=339832, write_cpu_seconds=0, service_cpu_seconds=0.033983, hot_btks_rolled=0, rsearch_filter_seconds=0									
# group 17				group = pipeline host = docs-unix-17f source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.016616									
# host 1													
# source 1													
# sourcetype 1													
# unexpectedness 100+													
INTERESTING FIELDS				> 3/16/18 03-16-2018 21:23:59.179 -0700 INFO Metrics - group=per_sourcetype_thruput, series="splunk_resource_usage", kbps=0.4586322857224797, eps=1.096836217867417, kb=14.216796875, ev=34, avg_age=0.38235294117647056, max_age=3									
# component 1				group = per_sourcetype_thruput host = docs-unix-17f source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.010139									
# cpu_seconds 32													
# cumulative_hits 100+													
# date_hour 12													
# date_mday 1													
# date_minute 60													
# date_month 1													
# date_second 60													
# date_wday 1													
# date_year 1													
# date_zone 1													
# ev 100+													
# executes 100+													
# index 1													
# kb 100+													
# linecount 1													
# log_level 1													
# message 100+													

The key-value pairs in the first event include `group=pipeline, name=indexerpipe, processor=indexer, cpu_seconds=0.022`, and so forth.

With the default `threshold`, which is 0.01, you can see that some of these events might be very similar. The next search increases the `threshold` a little:

```
index=_internal | anomalies threshold=0.03 by group | search group=*
```

List		Format	20 Per Page	< Prev	1	2	3	4	5	6	7	Next >	
« Hide Fields	All Fields	i	Time	Event									
SELECTED FIELDS				> 3/16/18 03-16-2018 22:54:24.180 -0700 INFO Metrics - group=pipeline, name=fchangemanager, processor=fsch angemanager, cpu_seconds=0, executes=1, cumulative_hits=71									
# group 9				group = pipeline host = docs-unix-17f source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.031949									
# host 1													
# source 1													
# sourcetype 1													
# unexpectedness 91													
INTERESTING FIELDS				> 3/16/18 03-16-2018 22:44:04.181 -0700 INFO Metrics - group=pipeline, name=fchangemanager, processor=fsch angemanager, cpu_seconds=0, executes=1, cumulative_hits=70									
# component 1				group = pipeline host = docs-unix-17f source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.030794									
# cpu_seconds 1													
# cumulative_hits 85													
# date_hour 13													
# date_mday 1													
# date_minute 45													
# date_month 1													
# date_second 55													
# date_wday 1													
# date_year 1													
# date_zone 1													
# executes 8													
# index 1													
# linecount 1													
# log_level 1													
# message 100+													

With the higher `threshold` value, the timestamps and key-value pairs show more distinction between each of the events.

Also, you might not want to hide the events that are not anomalous. Instead, you can add another field to your events that tells you whether or not the event is interesting to you. One way to do this is with the `eval` command:

```
index=_internal | anomalies threshold=0.03 labelonly=true by group | search group=* | eval threshold=0.03 | eval score;if(unexpectedness>=threshold, "anomalous", "boring")
```

This search uses `labelonly=true` so that the boring events are still retained in the results list. The `eval` command is used to define a field named `threshold` and set it to the threshold value. This has to be done explicitly because the `threshold` attribute of the `anomalies` command is not a field.

The second `eval` command is used to define another new field, `score`, that is either "anomalous" or "boring" based on how the `unexpectedness` compares to the `threshold` value. The following image shows a snapshot of the results.

List ▾		Format	20 Per Page ▾	< Prev	1	2	3	4	5	6	7	8	... Next >
◀ Hide Fields	>All Fields	i	Time	Event									
SELECTED FIELDS				> 3/16/18 03-16-2018 23:23:20.181 -0700 INFO Metrics - group=thruput, name=thruput, instantaneous_kbps=1.9519686558473517, instantaneous_eps=6.741749759538862, average_kbps=1.8345233045061788, total_k_processed=85828, kb=60.5126953125, ev=209, load_average=0.67									
a group 26				group = thruput host = docs-unix-17f score = anomalous									
a host 1				source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.062092									
a score 2													
a source 2													
a sourcetype 1													
# unexpectedness 100+													
INTERESTING FIELDS				> 3/16/18 03-16-2018 23:23:20.181 -0700 INFO Metrics - group=syslog_output, instantaneous_kbps=0, instantaneous_eps=0, average_kbps=0, total_k_processed=0, kb=0, ev=0									
a component 3				group = thruput host = docs-unix-17f score = anomalous									
# cpu_seconds 67				source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.105263									
# cumulative_hits 100+													
# current_size 36				> 3/16/18 03-16-2018 23:23:20.181 -0700 INFO Metrics - group=index_thruput, name=index_thruput, instantaneous_kbps=1.9661758640195958, instantaneous_eps=6.03209247384854, average_kbps=1.8346315199506906, total_k_processed=85825, kb=60.953125, ev=187									
# date_hour 14				group = thruput host = docs-unix-17f score = anomalous									
# date_mday 1				source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.943662									
# date_minute 60													
a date_month 1				> 3/16/18 03-16-2018 23:23:20.181 -0700 INFO Metrics - group=queue, name=typingqueue, max_size_kb=500, current_size_kb=0, current_size=0, largest_size=20, smallest_size=0									
# date_second 60				group = queue host = docs-unix-17f score = boring									
a date_wday 1				source = /opt/splunk-nightlight/var/log/splunk/metrics.log sourcetype = splunkd unexpectedness = 0.019022									
# date_year 1													
# date_zone 1													
# executes 100+													
a index 1													
# largest_size 100+													
## linecount 1													

See also

[anomalousvalue](#), [cluster](#), [kmeans](#), [outlier](#)

anomalousvalue

Description

The `anomalousvalue` command computes an anomaly score for each field of each event, relative to the values of this field across other events. For numerical fields, it identifies or summarizes the values in the data that are anomalous either by frequency of occurrence or number of standard deviations from the mean.

For fields that are determined to be anomalous, a new field is added with the following scheme. If the field is numeric, such as `size`, the new field will be `Anomaly_Score_Num(size)`. If the field is non-numeric, such as `name`, the new field will be `Anomaly_Score_Cat(name)`.

Syntax

anomalousvalue <av-options>... [action] [pthresh] [field-list]

Required arguments

None.

Optional arguments

<av-options>

Syntax: minsupcount=<int> | maxanofreq=<float> | minsupfreq=<float> | minnormfreq=<float>

Description: Specify one or more option to control which fields are considered for discriminating anomalies.

Descriptions for the av-option arguments

maxanofreq

Syntax: maxanofreq=<float>

Description: Maximum anomalous frequency is expressed as a floating point number between 0 and 1. Omits a field from consideration if the field is too frequently anomalous. If the ratio of anomalous occurrences of the field to the total number of occurrences of the field is greater than the `maxanofreq` value, then the field is removed from consideration.

Default: 0.05

minnormfreq

Syntax: minnormfreq=<float>

Description: Minimum normal frequency is expressed as a floating point number between 0 and 1. Omits a field from consideration if the field is not anomalous frequently enough. If the ratio of anomalous occurrences of the field to the total number of occurrences of the field is smaller than p , then the field is removed from consideration.

Default: 0.01

minsupcount

Syntax: minsupcount=<int>

Description: Minimum supported count must be a positive integer. Drops a field that has a small number of occurrences in the input result set. If the field appears fewer than N times in the input events, the field is removed from consideration.

Default: 100

minsupfreq

Syntax: minsupfreq=<float>

Description: Minimum supported frequency is expressed as a floating point number between 0 and 1. Drops a field that has a low frequency of occurrence. The `minsupfreq` argument checks the ratio of occurrences of the field to the total number of events. If this ratio is smaller than p the field is removed from consideration.

Default: 0.05

action

Syntax: action=annotate | filter | summary

Description: Specify whether to return the anomaly score (annotate), filter out events that are not anomalous values (filter), or return a summary of anomaly statistics (summary).

Default: filter

Descriptions for the action arguments

annotate

Syntax: action=annotate

Description: The `annotate` action adds new fields to the events containing anomalous values. The fields that are added are `Anomaly_Score_Cat(field)`, `Anomaly_Score_Num(field)`, or both.

filter

Syntax: action=filter

Description: The `filter` action returns events with anomalous values. Events without anomalous values are removed. The events that are returned are annotated, as described for `action=annotate`.

summary

Syntax: action=summary

Description: The `summary` action returns a table summarizing the anomaly statistics for each field generated. The table includes how many events contained this field, the fraction of events that were anomalous, what type of test (categorical or numerical) were performed, and so on.

Output field	Description
fieldname	The name of the field.
count	The number of times the field appears.
distinct_count	The number of unique values of the field.
mean	The calculated mean of the field values.
catAnoFreq%	The anomalous frequency of the categorical field.
catNormFreq%	The normal frequency of the categorical field.
numAnoFreq%	The anomalous frequency of the numerical field.
stdev	The standard deviation of the field value.
supportFreq%	The support frequency of the field.
useCat	Use categorical anomaly detection. Categorical anomaly detection looks for rare values.
useNum	Use numerical anomaly detection. Numerical anomaly detection looks for values that are far from the mean value. This anomaly detection is Gaussian distribution based.
isNum	Whether or not the field is numerical.

field-list

Syntax: <field> ...

Description: The List of fields to consider.

Default: If no field list is provided, all fields are considered.

pthresh

Syntax: pthresh=<num>

Description: Probability threshold (as a decimal) that has to be met for a value to be considered anomalous.

Default: 0.01.

Usage

By default, a maximum of 50,000 results are returned. This maximum is controlled by the `maxresultrows` setting in the `[anomalousvalue]` stanza in the `limits.conf` file. Increasing this limit can result in more memory usage.

Only users with file system access, such as system administrators, can edit the configuration files. Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make the changes in the local directory.

See How to edit a configuration file.

Basic examples

1. Return only uncommon values from the search results

```
... | anomalousvalue
```

This is the same as running the following search:

```
... | anomalousvalue action=filter pthresh=0.01
```

2. Return uncommon values from the host "reports"

```
host="reports" | anomalousvalue action=filter pthresh=0.02
```

Extended example

1. Return a summary of the anomaly statistics for each numeric field

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

Search for anomalous values in the earthquake data.

```
source="all_month.csv" | anomalousvalue action=summary pthresh=0.02 | search isNum=YES
```

Events (9,496)	Patterns	Statistics (21)	Visualization								
100 Per Page ▾		✓ Format	Preview ▾								
✓ catAnoFreq% ▽	✓ catNormFreq% ▽	✓ count ▽	✓ distinct_count ▽	✓ fieldname ▽	✓ isNum ▽	✓ mean ▽	✓ numAnoFreq% ▽	✓ stdev ▽	✓ supportFreq% ▽	✓ useCat ▽	✓ useNum ▽
0.0000	4.1667	9496	24	date_hour	YES	11.392060	0.0000	6.809368	100.0000	YES	YES
0.0000	3.2258	9496	31	date_mday	YES	15.885531	0.0000	8.225140	100.0000	YES	YES
0.0000	1.6667	9496	60	date_minute	YES	29.335088	0.0000	17.357693	100.0000	YES	YES
0.0000	1.6667	9496	60	date_second	YES	29.305708	0.0000	17.267549	100.0000	YES	YES
0.0000	100.0000	9496	1	date_year	YES	2018.000000	0.0000	0.000000	100.0000	NO	NO
0.0000	100.0000	9496	1	date_zone	YES	0.000000	0.0000	0.000000	100.0000	NO	NO
0.0000	0.0322	9496	3105	depth	YES	23.668763	2.8749	47.105182	100.0000	NO	YES
0.0000	0.1385	9454	722	depthError	YES	3.471316	0.0105	42.808936	99.5577	NO	YES
0.0000	0.0205	6441	4876	dmin	YES	0.561560	2.5379	1.671079	67.8286	NO	YES
0.0000	0.0859	6471	1164	gap	YES	123.678824	3.2645	68.523067	68.1445	NO	YES
0.0000	0.1942	5626	515	horizontalError	YES	1.997239	4.0227	3.325279	59.2460	NO	YES
0.0000	0.0118	9496	8467	latitude	YES	41.872144	4.2650	17.960626	100.0000	NO	YES
0.0000	100.0000	9496	1	linecount	YES	1.000000	0.0000	0.000000	100.0000	NO	NO
0.0000	0.0116	9496	8623	longitude	YES	-117.209617	4.9916	59.937839	100.0000	NO	YES
0.0000	0.2326	9495	430	mag	YES	1.548413	7.3294	1.180903	99.9895	NO	NO

The numeric results are returned with multiple decimals. Use the field formatting icon, which looks like a pencil, to enable number formatting and specify the decimal precision to display.

Events (9,496)	Patterns	Statistics (21)	Visualization
100 Per Page ▾		✓ Format	Preview ▾
0.0000	4.17	9496	24 date_hour YES 11.392060 0.0000 6.809368
0.0000	3.23		
0.0000	1.67		
0.0000	1.67		
0.0000	100.00		
0.0000	100.00		
0.0000	0.03		
0.0000	0.14		
0.0000	0.02		
0.0000	0.09	6471	1164 gap YES 123.678824 3.2645 68.523067
0.0000	0.19	5626	515 horizontalError YES 1.997239 4.0227 3.325279
0.0000	0.01	9496	8467 latitude YES 41.872144 4.2650 17.960626

See also

[analyzefields](#), [anomalies](#), [cluster](#), [kmeans](#), [outlier](#)

anomalydetection

Description

A transforming command that identifies anomalous events by computing a probability for each event and then detecting unusually small probabilities. The probability is defined as the product of the frequencies of each individual field value in the event.

- For categorical fields, the frequency of a value X is the number of times X occurs divided by the total number of events.
- For numerical fields, we first build a histogram for all the values, then compute the frequency of a value X as the size of the bin that contains X divided by the number of events.

The `anomalydetection` command includes the capabilities of the existing `anomalousvalue` and `outlier` commands and offers a histogram-based approach for detecting anomalies.

Syntax

`anomalydetection [<method-option>] [<action-option>] [<pthresh-option>] [<cutoff-option>] [<field-list>]`

Optional arguments

<method-option>

Syntax: `method = histogram | zscore | iqr`

Description: Select the method of anomaly detection. When `method=zscore`, performs like the `anomalousvalue` command. When `method=iqr`, performs like the `outlier` command. See [Usage](#).

Default: `method=histogram`

<action-option>

Syntax for method=histogram or method=zscore: `action = filter | annotate | summary`

Syntax for method=iqr: `action = remove | transform`

Description: The actions and defaults depend on the method that you specify. See the detailed descriptions for the actions for each method below.

<pthresh-option>

Syntax: `pthresh=<num>`

Description: Used with `method=histogram` or `method=zscore`. Sets the probability threshold, as a decimal number, that has to be met for an event to be deemed anomalous.

Default: For `method=histogram`, the command calculates `pthresh` for each data set during analysis. For `method=zscore`, the default is 0.01. If you try to use this when `method=iqr`, it returns an invalid argument error.

<cutoff-option>

Syntax: `cutoff=<bool>`

Description: Sets the upper bound threshold on the number of anomalies. This option applies to only the histogram method. If `cutoff=false`, the algorithm uses the formula `threshold = 1st-quartile - 1.5 * IQR`

without modification. If `cutoff=true`, the algorithm modifies the formula in order to come up with a smaller number of anomalies.

Default: true

<field-list>

Syntax: <string> <string> ...

Description: A list of field names.

Histogram actions

<action-option>

Syntax: action=annotate | filter | summary

Description: Specifies whether to return all events with additional fields (annotate), to filter out events with anomalous values (filter), or to return a summary of anomaly statistics (summary).

Default: filter

When `action=filter`, the command returns anomalous events and filters out other events. Each returned event contains four new fields. When `action=annotate`, the command returns all the original events with the same four new fields added when `action=filter`.

Field	Description
<code>log_event_prob</code>	The natural logarithm of the event probability.
<code>probable_cause</code>	The name of the field that best explains why the event is anomalous. No one field causes anomaly by itself, but often some field value occurs too rarely to make the event probability small.
<code>probable_cause_freq</code>	The frequency of the value in the <code>probable_cause</code> field.
<code>max_freq</code>	Maximum frequency for all field values in the event.

When `action=summary`, the command returns a single event containing six fields.

Output field	Description
<code>num_anomalies</code>	The number of anomalous events.
<code>thresh</code>	The event probability threshold that separates anomalous events.
<code>max_logprob</code>	The maximum of all <code>log(event_prob)</code> .
<code>min_logprob</code>	The minimum of all <code>log(event_prob)</code> .
<code>1st_quartile</code>	The first quartile of all <code>log(event_prob)</code> .
<code>3rd_quartile</code>	The third quartile of all <code>log(event_prob)</code> .

Zscore actions

<action-option>

Syntax: action=annotate | filter | summary

Description: Specifies whether to return the anomaly score (annotate), filter out events with anomalous values (filter), or a summary of anomaly statistics (summary).

Default: filter

When `action=filter`, the command returns events with anomalous values while other events are dropped. The kept events are annotated, like the `annotate` action.

When `action=annotate`, the command adds new fields, `Anomaly_Score_Cat(field)` and `Anomaly_Score_Num(field)`, to the events that contain anomalous values.

When `action=summary`, the command returns a table that summarizes the anomaly statistics for each field generated. The table includes how many events contained this field, the fraction of events that were anomalous, what type of test (categorical or numerical) were performed, and so on.

IQR actions

<action-option>

Syntax: `action=remove | transform`

Description: Specifies what to do with outliers. The `remove` action removes the event containing the outlying numerical value. The `transform` action transforms the event by truncating the outlying value to the threshold for outliers. If `mark=true`, the `transform` action prefixes the value with "000".

Abbreviations: The abbreviation for `remove` is `rm`. The abbreviation for `transform` is `tf`.

Default: `action=transform`

Usage

The `anomalydetection` command is a streaming command. See [Command types](#).

The `zscores` method

When you specify `method=zscores`, the `anomalydetection` command performs like the `anomalousvalue` command. You can specify the syntax components of the `anomalousvalue` command when you use the `anomalydetection` command with `method=zscores`. See the [anomalousvalue](#) command.

The `iqr` method

When you specify `method=iqr`, the `anomalydetection` command performs like the `outlier` command. You can specify the syntax components of the `outlier` command when you specify `method=iqr` with the `anomalydetection` command. For example, you can specify the outlier options <action>, <mark>, <param>, and <uselower>. See the [outlier](#) command.

Examples

Example 1: Return only anomalous events

These two searches return the same results. The arguments specified in the second search are the default values.

```
... | anomalydetection  
... | anomalydetection method=histogram action=filter
```

Example 2: Return a short summary of how many anomalous events are there

Return a short summary of how many anomalous events are there and some other statistics such as the threshold value used to detect them.

```
... | anomalydetection action=summary
```

Example 3: Return events with anomalous values

This example specifies `method=zscore` to return anomalous values. The search uses the `filter` action to filter out events that do not have anomalous values. Events must meet the probability threshold `pthresh` before being considered an anomalous value.

```
... | anomalydetection method=zscore action=filter pthresh=0.05
```

Example 4: Return outliers

This example uses the outlier options from the `outlier` command. The abbreviation `tf` is used for the transform action in this example.

```
... | anomalydetection method=iqr action=tf param=4 uselower=true mark=true
```

See also

[analyzefields](#), [anomalies](#), [anomalousvalue](#), [cluster](#), [kmeans](#), [outlier](#)

append

Description

Appends the results of a **subsearch** to the current results. The `append` command runs only over historical data and does not produce correct results if used in a real-time search.

For more information about when to use the `append` command, see the flowchart in the topic *About event grouping and correlation* in the *Search Manual*.

If you are familiar with SQL but new to SPL, see [Splunk SPL for SQL users](#).

Syntax

```
append [<subsearch-options>...] <subsearch>
```

Required arguments

`subsearch`

Syntax: `[subsearch]`

Description: A secondary search where you specify the source of the events that you want to append. The subsearch must be enclosed in square brackets. See *About subsearches* in the *Search Manual*.

Optional arguments

`subsearch-options`

Syntax: `extendtimerange=<boolean> | maxtime=<int> | maxout=<int> | timeout=<int>`

Description: Controls how the subsearch is processed.

Subsearch options

extendtimerange

Syntax: extendtimerange=<boolean>

Description: Specifies whether to include the subsearch time range in the time range for the entire search. Use the `extendtimerange` argument when the time range in the subsearch extends beyond the time range for the main search. Use this argument when a **transforming command**, such as `chart`, `timechart`, or `stats`, follows the `append` command in the search and the search uses time based bins.

Default: false

maxtime

Syntax: maxtime=<int>

Description: The maximum time, in seconds, to spend on the subsearch before automatically finalizing.

Default: 60

maxout

Syntax: maxout=<int>

Description: The maximum number of result rows to output from the subsearch.

Default: 50000

timeout

Syntax: timeout=<int>

Description: The maximum time, in seconds, to wait for subsearch to fully finish.

Default: 60

Usage

The `append` command is a transforming command. See [Command types](#).

Examples

1: Use the append command to add column totals.

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

Count the number of earthquakes that occurred in and around California yesterday and then calculate the total number of earthquakes.

```
source=usgs place=*California* | stats count by magType | append [search index=usgs_* source=usgs place=*California* | stats count]
```

This example uses a subsearch to count all the earthquakes in the California regions (`place="*California"`), then uses the main search to count the number of earthquakes based on the magnitude type of the search.

You cannot use the `stats` command to simultaneously count the total number of events and the number of events for a specified field. The subsearch is used to count the total number of earthquakes that occurred. This count is added to the results of the previous search with the `append` command.

Because both searches share the `count` field, the results of the subsearch are listed as the last row in the count column.

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

magType	count
H	123
MbLg	1
Md	1565
Me	2
MI	1202
Mw	6
ml	10
	2909

This search demonstrates how to use the `append` command in a way that is similar to using the `addcoltotals` command to add the column totals.

2. Count the number of different customers who purchased items. Append the top purchaser for each type of product.

This example uses the sample data from the Search Tutorial. 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.

Count the number of different customers who purchased something from the Buttercup Games online store yesterday, and break this count down by the type of product (accessories, t-shirts, and type of games) they purchased. Also, list the top purchaser for each type of product and how much that person bought of that product.

```
sourcetype=access_* action=purchase | stats dc(clientip) BY categoryId | append [search sourcetype=access_* action=purchase | top 1 clientip BY categoryId] | table categoryId, dc(clientip), clientip, count
```

This example first searches for purchase events (`action=purchase`). These results are piped into the `stats` command and the `dc()`, or `distinct_count()` function is used to count the number of different users who make purchases. The `BY` clause is used to break up this number based on the different category of products (`categoryId`).

This example contains a subsearch as an argument for the `append` command.

```
...[search sourcetype=access_* action=purchase | top 1 clientip BY categoryId]
```

The subsearch is used to search for purchase events and count the top purchaser (based on `clientip`) for each category of products. These results are added to the results of the previous search using the `append` command.

Here, the `table` command is used to display only the category of products (`categoryId`), the distinct count of users who bought each type of product (`dc(clientip)`), the actual user who bought the most of a product type (`clientip`), and the number of each product that user bought (`count`).

categoryId	dc(clientip)	clientip	count
ACCESSORIES		37	
ARCADE		58	
NULL		8	
SHOOTER		31	
SIMULATION		34	
SPORTS		13	
STRATEGY		74	
TEE		38	
ACCESSORIES	91.208.184.24		3
ARCADE	211.166.11.101		4
NULL	87.194.216.51		2
SHOOTER	87.194.216.51		2
SIMULATION	211.166.11.101		2
SPORTS	95.163.78.227		1
STRATEGY	76.169.7.252		3
TEE	87.194.216.51		2

You can see that the `append` command just tacks on the results of the subsearch to the end of the previous search, even though the results share the same field values. It does not let you manipulate or reformat the output.

3. Use the `append` command to determine the number of unique IP addresses that accessed the Web server.

Use the `append` command, along with the `stats`, `count`, and `top` commands to determine the number of unique IP addresses that accessed the Web server. Find the user who accessed the Web server the most for each type of page request.

This example uses the sample data from the Search Tutorial. 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.

Count the number of different IP addresses that accessed the Web server and also find the user who accessed the Web server the most for each type of page request (`method`).

```
sourcetype=access_* | stats dc(clientip), count by method | append [search sourcetype=access_* | top 1 clientip by method]
```

The Web access events are piped into the `stats` command and the `dc()` or `distinct_count()` function is used to count the number of different users who accessed the site. The `count()` function is used to count the total number of times the site was accessed. These numbers are separated by the page request (`method`).

The subsearch is used to find the top user for each type of page request (`method`). The `append` command is used to add the result of the subsearch to the bottom of the table.

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

method	dc(clientip)	count	clientip	percent
GET	173	2666		
POST	168	1727		
GET		83	87.194.216.51	3.113278

method	dc(clientip)	count	clientip	percent
POST		64	87.194.216.51	3.705848

The first two rows are the results of the first search. The last two rows are the results of the subsearch. Both result sets share the `method` and `count` fields.

4. Specify the maximum time for the subsearch to run and the maximum number of result rows from the subsearch

Use the `append` command, to determine the number of unique IP addresses that accessed the Web server. Find the user who accessed the Web server the most for each type of page request.

This example uses the sample dataset from the Search Tutorial but should work with any format of Apache web access log. Download the data set from [this topic in the Search Tutorial](#) and follow the instructions to upload it to your Splunk deployment. Use the time range **Yesterday** when you run this search.

Count the number of different IP addresses that accessed the Web server and also find the user who accessed the Web server the most for each type of page request (`method`). Limit the subsearch to 30 seconds and the maximum number of subsearch results to 1000.

```
sourcetype=access_* | stats dc(clientip), count by method | append maxtime=30 maxout=1000 [search sourcetype=access_* | top 1 clientip by method]
```

5. Use the `extendtimerange` argument

Use the `extendtimerange` argument to ensure that the time range used for the search includes both the time range of the main search and the time range of the subsearch.

```
index=_internal earliest=11/20/2017:00:00:00 latest=11/30/2017:00:00:00 |append extendtimerange=true [search index=_audit earliest=11/1/2017:00:00:00 latest=11/25/2017:00:00:00] |timechart span=1d count
```

The time range used for the search is from 11/1/2017:00:00:00, the earliest time in the subsearch, to 11/30/2017:00:00:00, the latest time in the main search.

See also

[appendcols](#), [appendpipe](#), [join](#), [set](#)

appendcols

Description

Appends the fields of the **subsearch** results with the input search results. All fields of the subsearch are combined into the current results, with the exception of **internal fields**. For example, the first subsearch result is merged with the first main result, the second subsearch result is merged with the second main result, and so on.

Syntax

```
appendcols [override= <bool> | <subsearch-options>...] <subsearch>
```

Required arguments

subsearch

Description: A secondary search added to the main search. See how subsearches work in the *Search Manual*.

Optional arguments

override

Syntax: override=<bool>

Description: If the `override` argument is false, and if a field is present in both a subsearch result and the main result, the main result is used. If `override=true`, the subsearch result value is used.

Default: `override=false`

subsearch-options

Syntax: maxtime=<int> | maxout=<int> | timeout=<int>

Description: These options control how the subsearch is executed.

Subsearch options

maxtime

Syntax: maxtime=<int>

Description: The maximum time, in units of seconds, to spend on the subsearch before automatically finalizing.

Default: 60

maxout

Syntax: maxout=<int>

Description: The maximum number of result rows to output from the subsearch.

Default: 50000

timeout

Syntax: timeout=<int>

Description: The maximum time, in units of seconds, to wait for subsearch to fully finish.

Default: 60

Usage

The `appendcols` command must be placed in a search string after a **transforming command** such as `stats`, `chart`, or `timechart`. The `appendcols` command can't be used before a transforming command because it must append to an existing set of table-formatted results, such as those generated by a transforming command. See [Command types](#).

Note that the subsearch argument to the `appendcols` command doesn't have to contain a transforming command.

Examples

Example 1:

Search for "404" events and append the fields in each event to the previous search results.

```
index=_internal | table host | appendcols [ search 404]
```

This is a valid search string because `appendcols` comes after the transforming command `table` and adds columns to an

existing table of results.

Example 2:

This search uses `appendcols` to count the number of times a certain field occurs on a specific server and uses that value to calculate other fields.

```
specific.server | stats dc(userID) as totalUsers | appendcols [ search specific.server AND "text" | addinfo | where _time >= info_min_time AND _time <=info_max_time | stats count(<field>) as variableA ] | eval variableB = exact(variableA/totalUsers)
```

- First, this search uses `stats` to count the number of individual users on a specific server and names that variable "totalUsers".
- Then, this search uses `appendcols` to search the server and count how many times a certain field occurs on that specific server. This count is renamed "VariableA". The `addinfo` command adds the `info_min_time` and `info_max_time` fields to the search results. The `where` command is used to constrain the subsearch within time range of those fields.
- The `eval` command is used to define a "variableB".

The result is a table with the fields `totalUsers`, `variableA`, and `variableB`.

See also

[append](#), [appendpipe](#), [join](#), [set](#)

appendpipe

Description

Appends the result of the subpipeline to the search results. Unlike a subsearch, the subpipeline is not run first. The subpipeline is run when the search reaches the `appendpipe` command. The `appendpipe` command is used to append the output of transforming commands, such as `chart`, `timechart`, `stats`, and `top`.

Syntax

```
appendpipe [run_in_preview=<bool>] [<subpipeline>]
```

Optional Arguments

`run_in_preview`

Syntax: `run_in_preview=<bool>`

Description: Specifies whether or not display the impact of the `appendpipe` command in the preview. When set to FALSE, the search runs and the preview shows the results as if the `appendpipe` command is not part of the search. However, when the search finishes, the results include the impact of the `appendpipe` command.

Default: True

`subpipeline`

Syntax: `<subpipeline>`

Description: A list of commands that are applied to the search results from the commands that occur in the search before the `appendpipe` command.

Usage

The `appendpipe` command can be useful because it provides a summary, total, or otherwise descriptive row of the entire dataset when you are constructing a table or chart. This command is also useful when you need the original results for additional calculations.

Examples

Example 1:

Append subtotals for each action across all users.

```
index=_audit | stats count by action user | appendpipe [stats sum(count) as count by action | eval user = "TOTAL - ALL USERS"] | sort action
```

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

action	user	count
accelerate_search	admin	209
accelerate_search	buttercup	345
accelerate_search	can-delete	6
accelerate_search	TOTAL - ALL USERS	560
add	n/a	1
add	TOTAL - ALL USERS	1
change_authentication	admin	50
change_authentication	buttercup	9
change_authentication	can-delete	24
change_authentication	TOTAL - ALL USERS	83

See also

[append](#), [appendcols](#), [join](#), [set](#)

arules

Description

The arules command looks for associative relationships between field values. The command returns a table with the following columns: Given fields, Implied fields, Strength, Given fields support, and Implied fields support. The given and implied field values are the values of the fields you supply. The Strength value indicates the relationship between (among) the given and implied field values.

Implements the arules algorithm as discussed in *Michael Hahsler, Bettina Gruen and Kurt Hornik (2012)*. *arules: Mining Association Rules and Frequent Itemsets. R package version 1.0-12*. This algorithm is similar to the algorithms used for online shopping websites which suggest related items based on what items other customers have viewed or purchased.

Syntax

arules [<arules-option>...] <field-list>...

Required arguments

field-list

Syntax: <field> <field> ...

Description: The list of field names. At least two fields must be specified.

Optional arguments

<arules-option>

Syntax: <support> | <confidence>

Description: Options for arules command.

arules options

support

Syntax: sup=<int>

Description: Specify a support limit. Associations with computed support levels smaller than this value are not included in the output results. The support option must be a positive integer.

Default: 3

confidence

Syntax: conf=<float>

Description: Specify a confidence limit. Associations with a confidence (expressed as strength field) are not included in the output results. Must be between 0 and 1.

Default: .5

Usage

The `arules` command is a streaming command that is both distributable streaming and centralized streaming. See [Command types](#).

Examples

Example 1: Search for the likelihood that the fields are related.

```
... | arules field1 field2 field3
```

Example 2:

```
... | arules sup=3 conf=.6 field1 field2 field3
```

See also

[associate](#), [correlate](#)

associate

Description

The `associate` command identifies correlations between fields. The command tries to find a relationship between pairs of fields by calculating a change in entropy based on their values. This entropy represents whether knowing the value of one field helps to predict the value of another field.

In Information Theory, *entropy* is defined as a measure of the uncertainty associated with a random variable. In this case if a field has only one unique value, the field has an entropy of zero. If the field has multiple values, the more evenly those values are distributed, the higher the entropy.

The `associate` command uses Shannon entropy (log base 2). The unit is in `bits`.

Syntax

`associate [<associate-options>...] [field-list]`

Required arguments

None.

Optional arguments

`associate-option`

Syntax: `supcnt | supfreq | improv`

Description: Options for the `associate` command. See the **Associate-options** section.

`field-list`

Syntax: `<field> ...`

Description: A list of one or more fields. You cannot use wildcard characters in the field list. If you specify a list of fields, the analysis is restricted to only those fields.

Default: All fields are analyzed.

Associate-options

`supcnt`

Syntax: `supcnt=<num>`

Description: Specifies the minimum number of times that the "reference key=reference value" combination must appear. Must be a non-negative integer.

Default: 100

`supfreq`

Syntax: `supfreq=<num>`

Description: Specifies the minimum frequency of "reference key=reference value" combination as a fraction of the number of total events.

Default: 0.1

improv

Syntax: improv=<num>

Description: Specifies a limit, or minimum entropy improvement, for the "target key". The calculated entropy improvement must be greater than or equal to this limit.

Default: 0.5

Columns in the output table

The `associate` command outputs a table with columns containing the following fields.

Field	Description
Reference_Key	The <i>name</i> of the first field in a pair of fields.
Reference_Value	The <i>value</i> in the first field in a pair of fields.
Target_Key	The name of the second field in a pair of fields.
Unconditional_Entropy	The entropy of the target key.
Conditional_Entropy	The entropy of the target key when the reference key is the reference value.
Entropy_Improvement	The difference between the unconditional entropy and the conditional entropy.
Description	A message that summarizes the relationship between the field values that is based on the entropy calculations. The Description is a textual representation of the result. It is written in the format: "When the 'Reference_Key' has the value 'Reference_Value', the entropy of 'Target_Key' decreases from Unconditional_Entropy to Conditional_Entropy."
Support	Specifies how often the reference field is the reference value, relative to the total number of events. For example, how often field A is equal to value X, in the total number of events.

Examples

1. Analyze the relationship between fields in web access log files

This example uses the sample data from the Search Tutorial. 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.

This example demonstrates one way to analyze the relationship of fields in your web access logs.

```
sourcetype=access_* status!=200 | fields method, status | associate improv=0.05 | table Reference_Key, Reference_Value, Target_Key, Top_Conditional_Value, Description
```

The first part of this search retrieves web access events that returned a status that is not 200. Web access data contains many fields. You can use the `associate` command to see a relationship between all pairs of fields and values in your data. To simplify this example, restrict the search to two fields: `method` and `status`.

Because the `associate` command adds many columns to the output, this search uses the `table` command to display only select columns.

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

Reference_Key	Reference_Value	Target_Key	Top_Conditional_Value	Description
method	POST	status	503 (17.44% -> 33.96%)	

Reference_Key	Reference_Value	Target_Key	Top_Conditional_Value	Description
				When 'method' has the value 'POST', the entropy of 'status' decreases from 2.923 to 2.729.
status	400	method	GET (76.37% -> 83.45%)	When 'status' has the value '400', the entropy of 'method' decreases from 0.789 to 0.647.
status	404	method	GET (76.37% -> 81.27%)	When 'status' has the value '404', the entropy of 'method' decreases from 0.789 to 0.696.
status	406	method	GET (76.37% -> 81.69%)	When 'status' has the value '406', the entropy of 'method' decreases from 0.789 to 0.687.
status	408	method	GET (76.37% -> 80.00%)	When 'status' has the value '408', the entropy of 'method' decreases from 0.789 to 0.722.
status	500	method	GET (76.37% -> 80.73%)	When 'status' has the value '500', the entropy of 'method' decreases from 0.789 to 0.707.

In the results you can see that there is one method and five status values in the results.

From the first row of results, you can see that when `method=POST`, the `status` field is 503 for those events. The `associate` command concludes that, if `method=POST`, the `Top_Conditional_Value` is likely to be 503 as much as 33% of the time.

The `Reference_Key` and `Reference_Value` are being correlated to the `Target_Key`.

The `Top_Conditional_Value` field states three things:

- The most common value for the given `Reference_Value`
- The frequency of the `Reference_Value` for that field in the dataset
- The frequency of the most common associated value in the `Target_Key` for the events that have the specific `Reference_Value` in that Reference Key.

It is formatted to read "CV (FRV% -> FCV%)" where CV is the conditional Value, FRV is is the percentage occurrence of the reference value, and FCV is the percentage of occurrence for that conditional value, in the case of the reference value.

2. Return results that have at least 3 references to each other

Return results associated with each other (that have at least 3 references to each other).

```
index=_internal sourcetype=splunkd | associate supcnt=3
```

3. Analyze events from a host

Analyze all events from host "reports" and return results associated with each other.

```
host="reports" | associate supcnt=50 supfreq=0.2 improv=0.5
```

See also

[arules](#), [correlate](#), [contingency](#)

autoregress

Description

Prepares your events for calculating the autoregression, or the *moving average*, by copying one or more of the previous values for *field* into each event.

The first few events will lack the augmentation of prior values, since the prior values do not exist.

Syntax

```
autoregress <field> [AS <newfield>] [ p=<int> | p=<int>-<int> ]
```

Required arguments

field

Syntax: <string>

Description: The name of a field. Most usefully a field with numeric values.

Optional arguments

p

Syntax: p=<int> | p=<int>-<int>

Description: Specifies which prior events to copy values from. You can specify a single integer or a numeric range. For a single value, such as 3, the `autoregress` command copies field values from the third prior event into a new field. For a range, the `autoregress` command copies field values from the range of prior events. For example, if you specify a range such as `p=2-4`, then the field values from the second, third, and fourth prior events are copied into new fields.

Default: 1

newfield

Syntax: <field>

Description: If `p` is set to a single integer, the `newfield` argument specifies a field name to copy the single field value into. Invalid if `p` is set to a range.

If the `newfield` argument is not specified, the single or multiple values are copied into fields with the names `<field>_p<num>`. For example, if `p=2-4` and `field=count`, the field names are `count_p2`, `count_p3`, `count_p4`.

Usage

The `autoregress` command is a centralized streaming command. See [Command types](#).

Examples

Example 1:

For each event, copy the 3rd previous value of the 'ip' field into the field 'old_ip'.

```
... | autoregress ip AS old_ip p=3
```

Example 2:

For each event, copy the 2nd, 3rd, 4th, and 5th previous values of the 'count' field.

```
... | autoregress count p=2-5
```

Since the `new` field argument is not specified, the values are copied into the fields 'count_p2', 'count_p3', 'count_p4', and 'count_p5'.

Example 3:

Calculate a moving average of event size over the current event and the four prior events. This search omits the `moving_average` for the initial events, where the field would be wrong, because summing null fields is considered null.

```
... | eval rawlen=len(_raw) | autoregress rawlen p=1-4 | eval moving_average=(rawlen + rawlen_p1 + rawlen_p2 + rawlen_p3 +rawlen_p4 ) /5
```

See also

[accum](#), [delta](#), [streamstats](#), [trendline](#)

awssnsalert

The `awssnsalert` command is used with the Splunk Add-on for AWS.

For information about this command, see Use the `awssnsalert` search command in *Splunk Add-on for AWS*.

bin

Description

Puts continuous numerical values into discrete sets, or bins, by adjusting the value of <field> so that all of the items in a particular set have the same value.

The bin command is automatically called by the chart and the timechart commands. Use the bin command for only statistical operations that the chart and the timechart commands cannot process. Do not use the bin command if you plan to export all events to CSV or JSON file formats.

Syntax

The required syntax is in **bold**.

```
bin
[<bin-options>...]
<field> [AS <newfield>]
```

Required arguments

field

Syntax: <field>

Description: Specify a field name.

Optional arguments

bin-options

Syntax: bins | minspan | span | <start-end> | aligntime

Description: Discretization options. See the [Bins options](#) section in this topic for the syntax and description for each of these options.

newfield

Syntax: <string>

Description: A new name for the field.

Bin options

bins

Syntax: bins=<int>

Description: Sets the maximum number of bins to discretize into. The default is set in the [discretize] stanza in the limits.conf file.

Default: 100

minspan

Syntax: minspan=<span-length>

Description: Specifies the smallest span granularity to use automatically inferring span from the data time range.

span

Syntax: span = <log-span> | <span-length>

Description: Sets the size of each bin, using a span length based on a logarithm-based span or based on time.

When a <span-length> of a day or more is used, the span is aligned to midnight in the timezone of the user.

<start-end>

Syntax: start=<num> | end=<num>

Description: Sets the minimum and maximum extents for numerical bins. The data in the field is analyzed and the beginning and ending values are determined. The start and end arguments are used when a span value is not specified.

You can use the start or end arguments only to *expand* the range, not to shorten the range. For example, if the field represents seconds the values are from 0-59. If you specify a span of 10, then the bins are calculated in increments of 10. The bins are 0-9, 10-19, 20-29, and so forth. If you do not specify a span, but specify end=1000,

the bins are calculated based on the actual beginning value and 1000 as the end value.

If you set end=10 and the values are >10, the end argument has no effect.

aligntime

Syntax: aligntime=(earliest | latest | <time-specifier>)

Description: Align the bin times to something other than base UTC time (epoch 0). The `aligntime` option is valid only when doing a time-based discretization. Ignored if `span` is in days, months, or years.

Span options

log-span

Syntax: [<num>]log[<num>]

Description: Sets to log-based span. The first number is a coefficient. The second number is the base. If the first number is supplied, it must be a real number ≥ 1.0 and <base>. Base, if supplied, must be real number > 1.0 (strictly greater than 1).

Example: `span=2log10`

span-length

Syntax: <int>[<timescale>]

Description: A span of each bin. If discretizing based on the `_time` field or used with a timescale, this is treated as a time range. If not, this is an absolute bin length.

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month> | <subseconds>

Description: Time scale units. If discretizing based on the `_time` field.

Default: sec

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

Usage

The `bucket` command is an alias for the `bin` command.

The `bin` command is usually a dataset processing command. If the `span` argument is specified with the command, the `bin` command is a streaming command. See [Command types](#).

Subsecond bin time spans

Subsecond `span` timescales—time spans that are made up of deciseconds (ds), centiseconds (cs), milliseconds (ms), or microseconds (us)—should be numbers that divide evenly into a second. For example, 1s = 1000ms. This means that valid millisecond `span` values are 1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, or 500ms. In addition, `span = 1000ms` is not allowed. Use `span = 1s` instead.

Examples

1. Specify a time span

Return the average "thruput" of each "host" for each 5 minute time span.

```
... | bin _time span=5m | stats avg(thruput) by _time host
```

2. Specify the number of bins

Bin search results into 10 bins, and return the count of raw events for each bin.

```
... | bin size bins=10 | stats count(_raw) by size
```

3. Specify an end value

Create bins with an end value larger than you need to ensure that all possible values are included.

```
... | bin amount end=1000
```

4. Specify a relative time to align the bins to

Align the time bins to 3am (local time). Set the span to 12h. The bins will represent 3am - 3pm, then 3pm - 3am (the next day), and so on.

```
... | bin _time span=12h aligntime=@d+3h
```

5. Specify a UTC time to align the bins to

Align the bins to the specific UTC time of 1500567890.

```
... | bin _time aligntime=1500567890
```

See also

[chart](#), [timechart](#)

bucket

The `bucket` command is an alias for the `bin` command. See the [bin command](#) for syntax information and examples.

bucketdir

Description

Replaces a field value with higher-level grouping, such as replacing filenames with directories.

Returns the `maxcount` events, by taking the incoming events and rolling up multiple sources into directories, by preferring directories that have many files but few events. The field with the path is PATHFIELD (e.g., `source`), and strings are broken up by a separator character. The default `pathfield=source; sizefield=totalCount; maxcount=20; countfield=totalCount; sep="/" or "\\"`, depending on the operation system.

Syntax

```
bucketdir pathfield=<field> sizefield=<field> [maxcount=<int>] [countfield=<field>] [sep=<char>]
```

Required arguments

pathfield

Syntax: `pathfield=<field>`

Description: Specify a field name that has a path value.

sizefield

Syntax: `sizefield=<field>`

Description: Specify a numeric field that defines the size of bucket.

Optional arguments

countfield

Syntax: `countfield=<field>`

Description: Specify a numeric field that describes the count of events.

maxcount

Syntax: `maxcount=<int>`

Description: Specify the total number of events to bucket.

sep

Syntax: `<char>`

Description: The separating character. Specify either a forward slash "/" or double back slashes "\\", depending on the operating system.

Usage

The `bucketdir` command is a streaming command. It is distributable streaming by default, but centralized streaming if the `local` setting specified for the command in the `commands.conf` file is set to true. See [Command types](#).

Examples

Example 1:

Return 10 best sources and directories.

```
... | top source | bucketdir pathfield=source sizefield=count maxcount=10
```

See also

[cluster](#), [dedup](#)

chart

Description

The `chart` command is a transforming command that returns your results in a table format. The results can then be used to display the data as a chart, such as a column, line, area, or pie chart. See the [Visualization Reference](#) in the [Dashboards and Visualizations](#) manual.

You must specify a statistical function when you use the `chart` command. See [Statistical and charting functions](#).

Syntax

The required syntax is in **bold**.

```
chart
[<chart-options>]
[agg=<stats-agg-term>]
( <stats-agg-term> | <sparkline-agg-term> | "("<eval-expression>"")" )...
[ BY <row-split> <column-split> ] | [ OVER <row-split> ] [BY <column-split>]
[<dedup_splitvals>]
```

Required arguments

You must include one of the following arguments when you use the `chart` command.

`stats-agg-term`

Syntax: `<stats-func> (<evalued-field> | <wc-field>) [AS <wc-field>]`

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wildcard characters in field names.

`sparkline-agg-term`

Syntax: `<sparkline-agg> [AS <wc-field>]`

Description: A sparkline aggregation function. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names. See [Sparkline options](#).

`eval-expression`

Syntax: `<eval-math-exp> | <eval-concat-exp> | <eval-compare-exp> | <eval-bool-exp> | <eval-function-call>`

Description: A combination of literals, fields, operators, and functions that represent the value of your destination field. For more information, see the [Evaluation functions](#). See [Usage](#).

For these evaluations to work, your 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. If both operands are strings, they can be concatenated. When concatenating values with a period, the search treats both values as strings regardless of their actual type.

Optional arguments

agg

Syntax: agg=<stats-agg-term>

Description: Specify an aggregator or function. For a list of stats functions with descriptions and examples, see [Statistical and charting functions](#).

chart-options

Syntax: cont | format | limit | sep

Description: Options that you can specify to refine the result. See the [Chart options](#) section in this topic.

Default:

column-split

Syntax: <field> [<tc-options>]... [<where-clause>]

Description: Specifies a field to use as the columns in the result table. By default, when the result are visualized, the columns become the data series in the chart. If the field is numerical, discretization is applied using the `tc-options` argument. See the [tc options](#) and the [where clause](#) sections in this topic.

Default: The number of columns included is limited to 10 by default. You can change the number of columns by including a <where-clause>.

When a column-split field is included, the output is a table where each column represents a distinct value of the split-by field. This is in contrast with the by-clause, where each row represents a single unique combination of values of the group-by fields. For additional information, see the Usage section in this topic.

dedup_splitvals

Syntax: dedup_splitvals=<boolean>

Description: Specifies whether to remove duplicate values in multivalued `BY` clause fields.

Default: false

row-split

Syntax: <field> [<bin-options>]...

Description: The field that you specify becomes the first column in the results table. The field values become the row labels in the results table. In a chart, the field name is used to label the X-axis. The field values become the X-axis values. See the [Bin options](#) section in this topic.

Default: None.

Chart options

cont

Syntax: cont=<bool>

Description: Specifies if the bins are continuous. If `cont=false`, replots the x-axis so that a noncontinuous sequence of x-value bins show up adjacently in the output. If `cont=true`, bins that have no values will display with a count of 0 or null values.

Default: true

format

Syntax: format=<string>

Description: Used to construct output field names when multiple data series are used in conjunction with a split-by-field. `format` takes precedence over `sep` and allows you to specify a parameterized expression with the stats aggregator and function (\$AGG\$) and the value of the split-by-field (\$VAL\$).

limit

Syntax: limit=(top | bottom) <int>

Description: Only valid when a column-split is specified. Use the `limit` option to specify the number of results that should appear in the output. When you set `limit=N` the top or bottom N values are retained, based on the sum of each series and the prefix you have selected. If `limit=0`, all results are returned. If you opt not to provide a prefix, the Splunk software provides the top results.

Default: top 10

sep

Syntax: sep=<string>

Description: Used to construct output field names when multiple data series are used in conjunctions with a split-by field. This is equivalent to setting `format` to \$AGG\$<sep>\$VAL\$.

Stats function options

stats-func

Syntax: The syntax depends on the function you use. See [Usage](#).

Description: Statistical and charting functions that you can use with the `chart` command. Each time you invoke the `chart` command, you can use one or more functions. However, you can only use one `BY` clause.

Sparkline options

Sparklines are inline charts that appear within table cells in search results and display time-based trends associated with the primary key of each row.

sparkline-agg

Syntax: sparkline (count(<wc-field>), <span-length>) | sparkline (<sparkline-func>(<wc-field>), <span-length>)

Description: A sparkline specifier, which takes the first argument of an aggregation function on a field and an optional timespan specifier. If no timespan specifier is used, an appropriate timespan is chosen based on the time range of the search. If the sparkline is not scoped to a field, only the count aggregate function is permitted. You can use wild card characters in field names.

span-length

See the [Span options](#) section in this topic.

sparkline-func

Syntax: c() | count() | dc() | mean() | avg() | stdev() | stdevp() | var() | varp() | sum() | sumsq() | min() | max() | range()

Description: Aggregation function to use to generate sparkline values. Each sparkline value is produced by applying this aggregation to the events that fall into each particular time bin.

The size of the sparkline is defined by settings in the limits.conf file. The `sparkline_maxsize` setting defines the maximum number of elements to emit for a sparkline.

For more information see Add sparklines to your search results in the *Search Manual*.

Bin options

The bin options control the number and size of the bins that the search results are separated, or discretized, into.

Syntax: bins | span | <start-end> | aligntime

Description: Discretization options.

Default: bins=300

bins

Syntax: bins=<int>

Description: Sets the maximum number of bins to discretize into. For example, if bin=300, the search finds the smallest bin size that results in no more than 300 distinct bins.

Default: 300

span

Syntax: span=<log-span> | span=<span-length>

Description: Sets the size of each bin, using a span length based on time or log-based span. See the [Span options](#) section in this topic.

<start-end>

Syntax: end=<num> | start=<num>

Description: Sets the minimum and maximum extents for numerical bins. Data outside of the [start, end] range is discarded.

aligntime

Syntax: aligntime=(earliest | latest | <time-specifier>)

Description: Align the bin times to something other than base UNIX time (epoch 0). The aligntime option is valid only when doing a time-based discretization. Ignored if span is in days, months, or years.

Span options

<log-span>

Syntax: [<num>]log[<num>]

Description: Sets to a logarithm-based span. The first number is a coefficient. The second number is the base. If the first number is supplied, it must be a real number ≥ 1.0 and < base. Base, if supplied, must be real number > 1.0 (strictly greater than 1).

span-length

Syntax: [<timescale>]

Description: A span length based on time.

Syntax: <int>

Description: The span of each bin. If using a timescale, this is used as a time range. If not, this is an absolute bucket "length."

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month> | <subseconds>

Description: Time scale units.

Time scale	Syntax	Description
------------	--------	-------------

<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

tc options

The timechart options are part of the <column-split> argument and control the behavior of splitting search results by a field. There are options that control the number and size of the bins that the search results are separated into. There are options that control what happens when events do not contain the split-by field, and for events that do not meet the criteria of the <where-clause>.

tc-options

Syntax: <bin-options> | usenull=<bool> | useother=<bool> | nullstr=<string> | otherstr=<string>
Description: Options for controlling the behavior of splitting by a field.

bin-options

See the [Bin options](#) section in this topic.

nullstr

Syntax: nullstr=<string>
Description: Specifies the name of the field for data series for events that do not contain the split-by field. The `nullstr` option is only applicable when the `usenull` option is set to `true`.
Default: NULL

otherstr

String: otherstr=<string>
Description: Specifies the name of the field for data series that do not meet the criteria of the <where-clause>. The `otherstr` option is only applicable when the `useother` option is set to `true`.
Default: OTHER

usenull

Syntax: usenull=<bool>
Description: Controls whether or not a series is created for events that do not contain the split-by field.
Default: true

useother

Syntax: useother=<bool>
Description: Specifies if a series should be added for data series not included in the graph because the series did not meet the criteria of the <where-clause>.
Default: true

where clause

The <where-clause> is part of the <column-split> argument.

where clause

Syntax: <single-agg> <where-comp>

Description: Specifies the criteria for including particular data series when a field is given in the `tc-by-clause`. The most common use of this option is to select for spikes rather than overall mass of distribution in series selection. The default value finds the top ten series by area under the curve. Alternately one could replace sum with max to find the series with the ten highest spikes. This has no relation to the where command.

single-agg

Syntax: count | <stats-func>(<field>)

Description: A single aggregation applied to a single field, including an evaluated field. No wildcards are allowed. The field must be specified, except when using the `count` aggregate function, which applies to events as a whole.

<stats-func>

See the [Statistical functions](#) section in this topic.

<where-comp>

Syntax: <wherein-comp> | <wherethresh-comp>

Description: The criteria for the <where-clause>.

<wherein-comp>

Syntax: (in |notin) (top | bottom)<int>

Description: A grouping criteria for the <where-clause>. The aggregated series value be in or not in some top or bottom grouping.

<wherethresh-comp>

Syntax: (< | >) <num>

Description: A threshold for the <where-clause>. The aggregated series value must be greater than or less than the specified numeric threshold.

Usage

The `chart` command is a transforming command. See [Command types](#).

Evaluation expressions

You can use the `chart` command with an eval expression. Unless you specify a `split-by` clause, the eval expression must be renamed.

Supported functions

You can use a wide range of functions with the `stats` command. For general information about using functions, see [Statistical and charting functions](#).

- For a list of statistical functions by category, see [Function list by category](#)
- For an alphabetical list of statistical functions, see [Alphabetical list of functions](#)

Functions and memory usage

Some functions are inherently more expensive, from a memory standpoint, than other functions. For example, the `distinct_count` function requires far more memory than the `count` function. The `values` and `list` functions also can consume a lot of memory.

If you are using the `distinct_count` function without a split-by field or with a low-cardinality split-by by field, consider replacing the `distinct_count` function with the the `estdc` function (estimated distinct count). The `estdc` function might result in significantly lower memory usage and run times.

Apply a statistical function to all available fields

Some statistical commands, such as `stats`, process functions that are not paired with one or more fields as if they are implicitly paired with a wildcard, so the command applies the function all available fields. For example, `| stats sum` is treated as if it is `| stats sum(*)`.

The `chart` command allows this behavior only with the `count` function. If you do not specify a field for `count`, `chart` applies it to all events returned by the search. If you want to apply other functions to all fields, you must make the wildcard explicit: `| chart sum(*)`.

X-axis

You can specify which field is tracked on the x-axis of the chart. The x-axis variable is specified with a `by` field and is discretized if necessary. Charted fields are converted to numerical quantities if necessary.

Unlike the `timechart` command which generates a chart with the `_time` field as the x-axis, the `chart` command produces a table with an arbitrary field as the x-axis.

You can also specify the x-axis field after the `over` keyword, before any `by` and subsequent `split-by` clause. The `limit` and `agg` options allow easier specification of series filtering. The `limit` and `agg` options are ignored if an explicit `where`-clause is provided.

Using row-split and column-split fields

When a column-split field is included, the output is a table where each column represents a distinct value of the column-split field. This is in contrast with the `stats` command, where each row represents a single unique combination of values of the group-by fields. The number of columns included is limited to 10 by default. You can change the number of columns by including a `where`-clause.

With the `chart` and `timechart` commands, you cannot specify the same field in a function and as the row-split field.

For example, you **cannot** run this search. The field A is specified in the `sum` function and the `row-split` argument.

```
... | chart sum(A) by A span=log2
```

You must specify a different field as in the `row-split` argument.

Alternatively, you can work around this problem by using an `eval` expression. For example:

```
... | eval A1=A | chart sum(A) by A1 span=log2
```

Subsecond bin time spans

Subsecond `span` timescales—time spans that are made up of decisconds (ds), centiseconds (cs), milliseconds (ms), or microseconds (us)—should be numbers that divide evenly into a second. For example, 1s = 1000ms. This means that valid millisecond `span` values are 1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, or 500ms. In addition, `span = 1000ms` is not allowed. Use `span = 1s` instead.

Basic examples

1. Chart the max(delay) for each value in a field

Return the maximum delay for each value in the `site` field.

```
... | chart max(delay) OVER site
```

2. Chart the max(delay) for each value in a field, split by the value of another field

Return the maximum delay for each value in the `site` field split by the value in the `org` field.

```
... | chart max(delay) OVER site BY org
```

3. Chart the ratio of the average to the maximum "delay" for each distinct "host" and "user" pair

Return the ratio of the average (mean) of the `size` field to the maximum "delay" for each distinct `host` and `user` pair.

```
... | chart eval(avg(size)/max(delay)) AS ratio BY host user
```

4. Chart the maximum "delay" by "size" and separate "size" into bins

Return the maximum value in the `delay` field by the `size` field, where `size` is broken down into a maximum of 10 equal sized bins.

```
... | chart max(delay) BY size bins=10
```

5. Chart the average size for each distinct value in a filed

Return the average (mean) value in the `size` field for each distinct value in the `host` field.

```
... | chart avg(size) BY host
```

6. Chart the number of events, grouped by date and hour

Return the number of events, grouped by date and hour of the day, using `span` to group per 7 days and 24 hours per half days. The `span` applies to the field immediately prior to the command.

```
... | chart count BY date_mday span=3 date_hour span=12
```

7. Align the chart time bins to local time

Align the time bins to 5am (local time). Set the `span` to 12h. The bins will represent 5am - 5pm, then 5pm - 5am (the next day), and so on.

```
... | chart count BY _time span=12h aligntime=@d+5h
```

8. In a multivalue BY field, remove duplicate values

For each unique value of `mvfield`, chart the average value of `field`. Deduplicates the values in the `mvfield`.

```
... | chart avg(field) BY mvfield dedup_splitval=true
```

Extended examples

1. Specify <row-split> and <column-split> values with the chart command

This example uses events that list the numeric sales for each product and quarter, for example:

products	quarter	sales
ProductA	QTR1	1200
ProductB	QTR1	1400
ProductC	QTR1	1650
ProductA	QTR2	1425
ProductB	QTR2	1175
ProductC	QTR2	1550
ProductA	QTR3	1300
ProductB	QTR3	1250
ProductC	QTR3	1375
ProductA	QTR4	1550
ProductB	QTR4	1700
ProductC	QTR4	1625

To summarize the data by product for each quarter, run this search:

```
source="addtotalsData.csv" | chart sum(sales) BY products quarter
```

In this example, there are two fields specified in the BY clause with the `chart` command.

- The `products` field is referred to as the <row-split> field. In the chart, this field forms the X-axis.
- The `quarter` field is referred to as the <column-split> field. In the chart, this field forms the data series.

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

products	QTR1	QTR2	QTR3	QTR4
ProductA	1200	1425	1300	1550
ProductB	1400	1175	1250	1700
ProductC	1650	1550	1375	1625

Click on the **Visualization** tab to see the results as a chart.

See the `addtotals` command for an example that adds a total column for each product.

2. Chart the number of different page requests for each Web server

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 **All time** when you run the search.

Chart the number of different page requests, GET and POST, that occurred for each Web server.

```
sourcetype=access_* | chart count(eval(method="GET")) AS GET, count(eval(method="POST")) AS POST by host
```

This example uses `eval` expressions to specify the different field values for the `stats` command to count. The first clause uses the `count()` function to count the Web access events that contain the `method` field value `GET`. Then, using the `AS` keyword, the field that represents these results is renamed `GET`.

The second clause does the same for `POST` events. The counts of both types of events are then separated by the web server, using the `BY` clause with the `host` field.

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

host	GET	POST
www1	8431	5197
www2	8097	4815
www3	8338	4654

Click the **Visualization** tab. If necessary, format the results as a column chart. This chart displays the total count of events for each event type, `GET` or `POST`, based on the `host` value.



3. Chart the number of transactions by duration

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Create a chart to show the number of transactions based on their duration (in seconds).

```
sourcetype=access_* status=200 action=purchase | transaction clientip maxspan=10m | chart count BY duration span=log2
```

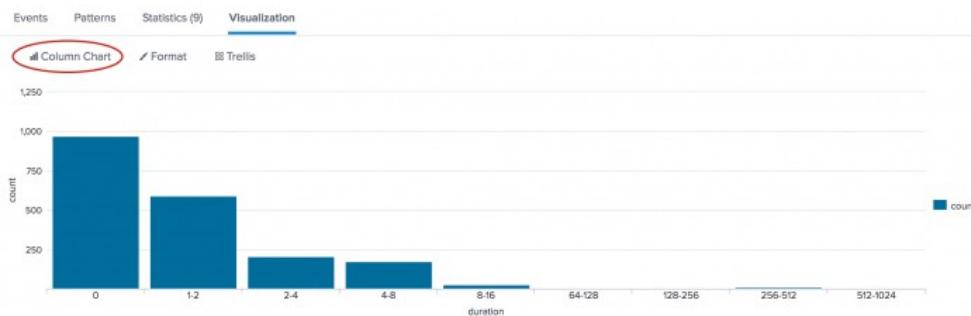
This search uses the `transaction` command to define a transaction as events that share the `clientip` field and fit within a ten minute time span. The `transaction` command creates a new field called `duration`, which is the difference between the timestamps for the first and last events in the transaction. (Because `maxspan=10s`, the `duration` value should not be greater than this.)

The transactions are then piped into the `chart` command. The `count()` function is used to count the number of transactions and separate the count by the duration of each transaction. Because the duration is in seconds and you expect there to be many values, the search uses the `span` argument to bucket the duration into bins of `log2 (span=log2)`.

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

duration	count
0	970
1-2	593
2-4	208
4-8	173
8-16	26
64-128	3
128-256	3
256-512	12
512-1024	2

Click the **Visualization** tab. If necessary, format the results as a column chart.



In this data set, most transactions take between 0 and 2 seconds to complete.

4. Chart the average number of events in a transaction, based on transaction duration

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Create a chart to show the average number of events in a transaction based on the duration of the transaction.

```
sourcetype=access_* status=200 action=purchase | transaction clientip maxspan=30m | chart avg(eventcount) by duration span=log2
```

The `transaction` command adds two fields to the results `duration` and `eventcount`. The `eventcount` field tracks the number of events in a single transaction.

In this search, the transactions are piped into the `chart` command. The `avg()` function is used to calculate the average number of events for each duration. Because the duration is in seconds and you expect there to be many values, the

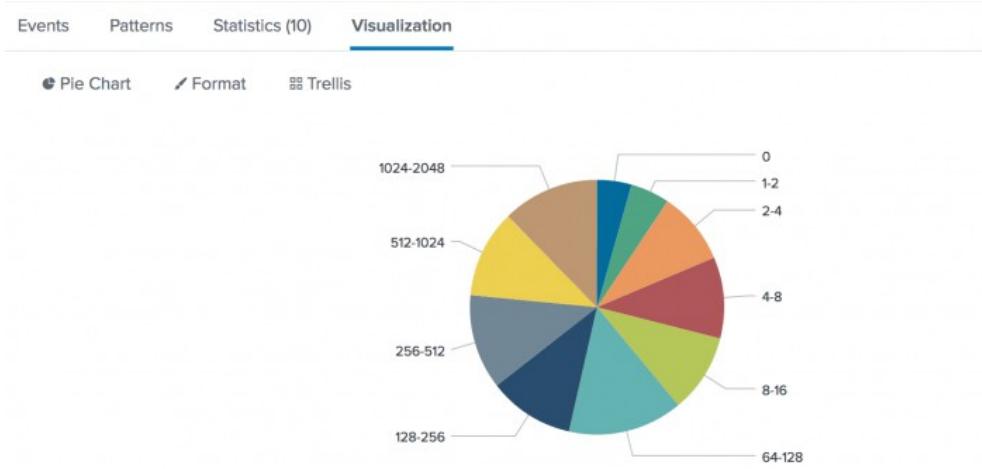
search uses the `span` argument to bucket the duration into bins using logarithm with a base of 2.

Use the field format option to enable number formatting.

The screenshot shows the Kibana search interface with the following details:

- Search query: `sourcetype=access_* status=200 action=purchase | transaction clientip maxspan=30m | chart avg(eventcount) by duration span=log2`
- Results: 1,931 events (before 3/16/18 3:44:09.000 PM) No Event Sampling
- Time range: All time
- Mode: `avg(eventcount)`
- Table view (Events tab): Shows duration bins from 0 to 1024-2048 with their respective event counts.
- Statistics (10) tab: Selected.
- Visualization tab: Available but not selected.
- Format tab: Available but not selected.
- Preview tab: Available but not selected.
- Context menu (over avg(eventcount)): Shows 'field format option' which is expanded to show 'Number Formatting' with 'Disabled' selected (highlighted).

Click the **Visualization** tab and change the display to a pie chart.



Each wedge of the pie chart represents a duration for the event transactions. You can hover over a wedge to see the average values.

5. Chart customer purchases

This example uses the sample data from the Search Tutorial. 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.

Chart how many different people bought something and what they bought at the Buttercup Games online store Yesterday.

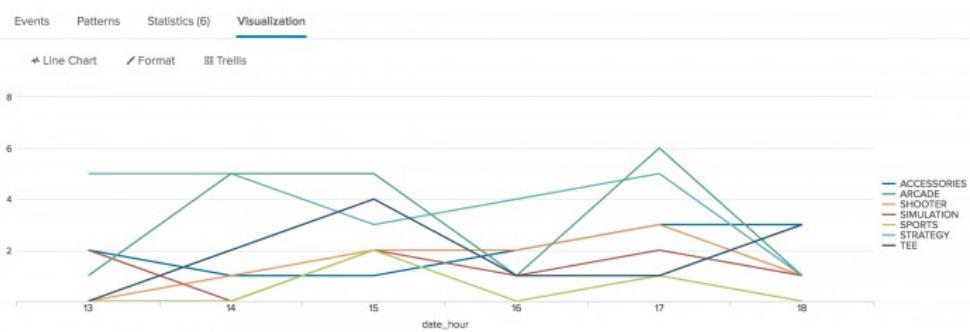
```
sourcetype=access_* status=200 action=purchase | chart dc(clientip) OVER date_hour BY categoryId usenull=f
```

This search takes the purchase events and pipes it into the `chart` command. The `dc()` or `distinct_count()` function is used to count the number of unique visitors (characterized by the `clientip` field). This number is then charted over each hour of the day and broken out based on the `category_id` of the purchase. Also, because these are numeric values, the search uses the `usenull=f` argument to exclude fields that don't have a value.

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

date_hour	ACCESSORIES	ARCADE	SHOOTER	SIMULATION	SPORTS	STRATEGY	TEE
0	2	6	0	4	0	4	4
1	4	7	2	3	0	10	5
2	2	2	2	1	1	2	0
3	3	5	3	5	0	7	1
4	3	4	0	0	1	4	0
5	3	0	3	0	1	6	1

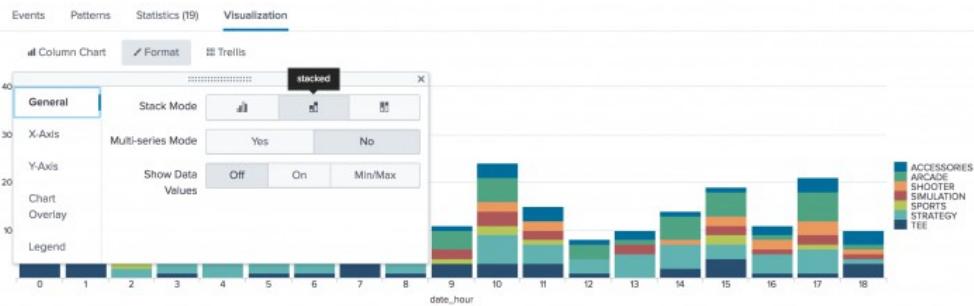
Click the **Visualization** tab. If necessary, format the results as a line chart:



Each line represents a different type of product that is sold at the Buttercup Games online store. The height of each line shows the number of different people who bought the product during that hour. In general, it looks like the most popular items at the online shop were Arcade games.

You can format the report as a stacked column chart, which will show you the total purchases at each hour of day.

1. Change the chart type to a **Column Chart**.
2. Use the **Format** menu, and on the **General** tab select **stacked**.



6. Chart the number of earthquakes and the magnitude of each earthquake

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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and add it as an input.

Create a chart that lists the number of earthquakes, and the magnitude of each earthquake that occurred in and around Alaska. Run the search using the time range **All time**.

```
source=all_month.csv place=*alaska* mag>=3.5 | chart count BY mag place useother=f | rename mag AS Magnitude
```

This search counts the number of earthquakes that occurred in the Alaska regions. The count is then broken down for each `place` based on the magnitude of the quake. Because the `place` value is non-numeric, the search uses the `useother=f` argument to exclude events that don't match.

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

Magnitude	145km ENE of Chirikof Island, Alaska	225km SE of Kodiak, Alaska	250km SE of Kodiak, Alaska	252km SE of Kodiak, Alaska	254km SE of Kodiak, Alaska	255km SE of Kodiak, Alaska	259km SE of Kodiak, Alaska	264km SE of Kodiak, Alaska	265km SE of Kodiak, Alaska	Gulf of Alaska
3.5	1	1	0	1	0	1	0	0	2	2
3.6	0	0	1	0	0	0	0	1	0	1
3.7	0	0	0	0	1	0	0	0	0	2
3.8	0	1	0	0	0	0	1	1	0	3
3.9	0	0	1	0	1	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0	1

Magnitude	145km ENE of Chirikof Island, Alaska	225km SE of Kodiak, Alaska	250km SE of Kodiak, Alaska	252km SE of Kodiak, Alaska	254km SE of Kodiak, Alaska	255km SE of Kodiak, Alaska	259km SE of Kodiak, Alaska	264km SE of Kodiak, Alaska	265km SE of Kodiak, Alaska	Gulf of Alaska
4.1	0	0	0	0	0	0	0	0	0	1
4.2	0	0	0	1	0	0	0	0	0	1
4.3	0	0	0	0	0	0	0	0	0	1
4.4	0	0	0	0	0	0	1	0	0	1
4.6	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	1

Click on the **Visualization** tab to view the results on a chart. This chart shows the number of earthquakes by magnitude.



See also

Commands

[timechart](#)
[bin](#)
[sichart](#)

Blogs

Search commands > stats, chart, and timechart

cluster

Description

The `cluster` command groups events together based on how similar they are to each other. Unless you specify a different field, `cluster` groups events based on the contents of the `_raw` field. The default grouping method is to break down the

events into terms (`match=termlist`) and compute the vector between events. Set a higher threshold value for `t`, if you want the command to be more discriminating about which events are grouped together.

The result of the cluster command appends two new fields to each event. You can specify what to name these fields with the `countfield` and `labelfield` parameters, which default to `cluster_count` and `cluster_label`. The `cluster_count` value is the number of events that are part of the cluster, or the cluster size. Each event in the cluster is assigned the `cluster_label` value of the cluster it belongs to. For example, if the search returns 10 clusters, then the clusters are labeled from 1 to 10.

Syntax

cluster [slc-options]...

Optional arguments

slc-options

Syntax: `t=<num> | delims=<string> | showcount=<bool> | countfield=<field> | labelfield=<field> | field=<field> | labelonly=<bool> | match=(termlist | termset | ngramset)`

Description: Options for configuring simple log clusters (slc).

SLC options

t

Syntax: `t=<num>`

Description: Sets the cluster threshold, which controls the sensitivity of the clustering. This value needs to be a number greater than 0.0 and less than 1.0. The closer the threshold is to 1, the more similar events have to be for them to be considered in the same cluster.

Default: 0.8

delims

Syntax: `delims=<string>`

Description: Configures the set of delimiters used to tokenize the raw string. By default, everything except 0-9, A-Z, a-z, and '_' are delimiters.

showcount

Syntax: `showcount=<bool>`

Description: If `showcount=false`, indexers cluster its own events before clustering on the search head. When `showcount=false` the event count is not added to the event. When `showcount=true`, the event count for each cluster is recorded and each event is annotated with the count.

Default: `showcount=false`

countfield

Syntax: `countfield=<field>`

Description: Name of the field to which the cluster size is to be written if `showcount=true` is true. The cluster size is the count of events in the cluster.

Default: `cluster_count`.

labelfield

Syntax: `labelfield=<field>`

Description: Name of the field to write the cluster number to. As the events are grouped into clusters, each cluster is counted and labelled with a number.

Default: cluster_label

field

Syntax: field=<field>

Description: Name of the field to analyze in each event.

Default: _raw

labelonly

Description: labelonly=<bool>

Syntax: Select whether to preserve incoming events and annotate them with the cluster they belong to (labelonly=true) or output only the cluster fields as new events (labelonly=false). When labelonly=false, outputs the list of clusters with the event that describes it and the count of events that combined with it.

Default: false

match

Syntax: match=(termlist | termset | ngramset)

Description: Select the method used to determine the similarity between events. termlist breaks down the field into words and requires the exact same ordering of terms. termset allows for an unordered set of terms. ngramset compares sets of trigram (3-character substrings). ngramset is significantly slower on large field values and is most useful for short non-textual fields, like punct.

Default: termlist

Usage

The `cluster` command is a streaming command or a dataset processing command, depending on which arguments are specified with the command. See [Command types](#).

Use the `cluster` command to find common or rare events in your data. For example, if you are investigating an IT problem, use the `cluster` command to find anomalies. In this case, anomalous events are those that are not grouped into big clusters or clusters that contain few events. Or, if you are searching for errors, use the `cluster` command to see approximately how many different types of errors there are and what types of errors are common in your data.

Examples

Example 1

Quickly return a glimpse of anything that is going wrong in your Splunk deployment. Your role must have the appropriate capabilities to access the internal indexes.

```
index=_internal source=*splunkd.log* log_level!=info | cluster showcount=t | table cluster_count _raw | sort -cluster_count
```

This search takes advantage of what Splunk software logs about its operation in the `_internal` index. It returns all logs where the `log_level` is DEBUG, WARN, ERROR, FATAL and clusters them together. Then it sorts the clusters by the count of events in each cluster.

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

cluster_count	raw
303010	03-20-2018 09:37:33.806 -0700 ERROR HotDBManager - Unable to create directory

cluster_count	raw
	/Applications/Splunk/var/lib/splunk/_internaldb/db/hot_v1_49427345 because No such file or directory
151506	03-20-2018 09:37:33.811 -0700 ERROR pipeline - Uncaught exception in pipeline execution (indexer) - getting next event
16390	04-05-2018 08:30:53.996 -0700 WARN SearchResultsMem - Failed to append to multival. Original value not converted successfully to multival.
486	03-20-2018 09:37:33.811 -0700 ERROR BTreeCP - failed: failed to mkdir /Applications/Splunk/var/lib/splunk/fishbucket/splunk_private_db/snapshot.tmp: No such file or directory
216	03-20-2018 09:37:33.814 -0700 WARN DatabaseDirectoryManager - idx=_internal Cannot open file='/Applications/Splunk/var/lib/splunk/_internaldb/db/.bucketManifest99454_1652919429_tmp' for writing bucket manifest (No such file or directory)
216	03-20-2018 09:37:33.814 -0700 ERROR SearchResultsWriter - Unable to open output file: path=/Applications/Splunk/var/lib/splunk/_internaldb/db/.bucketManifest99454_1652919429_tmp error=No such file or directory

Example 2

Search for events that don't cluster into large groups.

```
... | cluster showcount=t | sort cluster_count
```

This returns clusters of events and uses the `sort` command to display them in ascending order based on the cluster size, which are the values of `cluster_count`. Because they don't cluster into large groups, you can consider these rare or uncommon events.

Example 3

Cluster similar error events together and search for the most frequent type of error.

```
error | cluster t=0.9 showcount=t | sort - cluster_count | head 20
```

This searches your index for events that include the term "error" and clusters them together if they are similar. The `sort` command is used to display the events in descending order based on the cluster size, `cluster_count`, so that largest clusters are shown first. The `head` command is then used to show the twenty largest clusters. Now that you've found the most common types of errors in your data, you can dig deeper to find the root causes of these errors.

Example 4

Use the `cluster` command to see an overview of your data. If you have a large volume of data, run the following search over a small time range, such as 15 minutes or 1 hour, or restrict it to a source type or index.

```
... | cluster labelonly=t showcount=t | sort - cluster_count, cluster_label, _time | dedup 5 cluster_label
```

This search helps you to learn more about your data by grouping events together based on their similarity and showing you a few of events from each cluster. It uses `labelonly=t` to keep each event in the cluster and append them with a `cluster_label`. The `sort` command is used to show the results in descending order by its size (`cluster_count`), then its `cluster_label`, then the indexed timestamp of the event (`_time`). The `dedup` command is then used to show the first five events in each cluster, using the `cluster_label` to differentiate between each cluster.

See also

[anomalies](#), [anomalousvalue](#), [kmeans](#), [outlier](#)

cofilter

Description

Use this command to determine how many times a value in <field1> and a value in <field2> occur together. For example, if you have a field that contains user IDs and another field that contains items names, this command finds how common each pair of user and item occur.

This command implements one step in a collaborative filtering analysis for making recommendations.

Syntax

cofilter <field1> <field2>

Required arguments

field1

Syntax: <field>

Description: The name of field.

field2

Syntax: <field>

Description: The name of a field.

Usage

The `cofilter` command is a transforming command. See [Command types](#).

Examples

Example 1

Find the cofilter for `user` and `item`. The `user` field must be specified first and followed by the `item` field. The output is an event for each pair of items with: the first item and its popularity, the second item and its popularity, and the popularity of that pair of items.

Let's start with a simple search to create a few results:

```
| makeresults | eval user="a b c a b c a b c" | makemv user | mvexpand user | streamstats count
```

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

_time	count	user
-------	-------	------

2020-02-19 21:17:54	1	a
2020-02-19 21:17:54	2	b
2020-02-19 21:17:54	3	c
2020-02-19 21:17:54	4	a
2020-02-19 21:17:54	5	b
2020-02-19 21:17:54	6	c
2020-02-19 21:17:54	7	a
2020-02-19 21:17:54	8	b
2020-02-19 21:17:54	9	c

The `eval` command with the modulus (%) operator is used to create the `item` field:

```
| makeresults | eval user="a b c a b c a b c" | makemv user | mvexpand user | streamstats count | eval item = count % 5
```

The results look something like this:

_time	count	item	user
2020-02-19 21:17:54	1	1	a
2020-02-19 21:17:54	2	2	b
2020-02-19 21:17:54	3	3	c
2020-02-19 21:17:54	4	4	a
2020-02-19 21:17:54	5	0	b
2020-02-19 21:17:54	6	1	c
2020-02-19 21:17:54	7	2	a
2020-02-19 21:17:54	8	3	b
2020-02-19 21:17:54	9	4	c

Add the `cofilter` command to the search to determine how many `user` values occurred with each `item` value,

```
| makeresults | eval user="a b c a b c a b c" | makemv user | mvexpand user | streamstats count | eval item = count % 5 | cofilter user item
```

The results look something like this:

Item 1	Item 1 user count	Item 2	Item 2 user count	Pair count
1	2	2	2	1
1	2	3	2	1
1	2	4	2	2
2	2	3	2	1
2	2	4	2	1
2	2	0	1	1

Item 1	Item 1 user count	Item 2	Item 2 user count	Pair count
3	2	4	2	1
3	2	0	1	1

See also

[associate](#), [correlate](#)

collect

Description

Adds the results of a search to a **summary index** that you specify. You must create the summary index before you invoke the `collect` command.

You do not need to know how to use `collect` to create and use a summary index, but it can help. For an overview of summary indexing, see Use summary indexing for increased reporting efficiency in the *Knowledge Manager Manual*.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

The required syntax is in **bold**.

```
collect
index=<string>
[<arg-options>...]
```

Required arguments

index

Syntax: `index=<string>`

Description: Name of the summary index where the events are added. The index must exist before the events are added. The index is not created automatically.

Optional arguments

arg-options

Syntax: `addinfo=<bool> | addtime=<bool> | file=<string> | spool=<bool> | marker=<string> | uselb=<bool> | output_format [raw | hec] | testmode=<bool> | run_in_preview=<bool> | host=<string> | source=<string> | sourcetype=<string> | timeformat=<string>`

Description: Optional arguments for the `collect` command. See the **arg-options** section for the descriptions for each option.

arg-options

addinfo

Syntax: addinfo=<bool>

Description: Use this option to specify whether to prefix search time and time-range information fields on to each summary index event. If set to `true`, adds fields to each event in the following format:

```
info_min_time=<search_earliest_time>, info_max_time=<search_latest_time>,
info_search_time=<search_exec_time>
```

Default: True when summary events are destined for an events index or when `output_format=raw`. False when summary events are destined for a metrics index.

addtime

Syntax: addtime=<bool>

Description: Use this option to specify whether to prefix a time field on to each event. Some commands return results that do not have a `_raw` field, such as the `stats`, `chart`, `timechart` commands. If you specify `addtime=false`, the Splunk software uses its generic date detection against fields in whatever order they happen to be in the summary rows. If you specify `addtime=true`, the Splunk software uses the search time range `info_min_time`. This time range is added by the `sistats` command or `_time`. Splunk software adds the time field based on the first field that it finds: `info_min_time`, `_time`, or `now()`.

This option is not valid when `output_format=hec`.

Default: True when summary events are destined for an events index. False when summary events are destined for a metrics index.

file

Syntax: file=<string>

Description: The file name where you want the events to be written. You can use a timestamp or a random number for the file name by specifying either `file=$timestamp$` or `file=$random$`.

Usage: ".stash" needs to be added at the end of the file name when used with "index=". Otherwise, the data is added to the main index.

Default: <random-number>_events.stash

host

Syntax: host=<string>

Description: The name of the host that you want to specify for the events.

This option is not valid when `output_format=hec`.

marker

Syntax: marker=<string>

Description: A string, usually of key-value pairs, to append to each event written out. Each key-value pair must be separated by a comma and a space.

If the value contains spaces or commas, it must be escape quoted. For example if the key-value pair is `search_name=vpn starts and stops`, you must change it to `search_name=\"vpn starts and stops\"`.

This option is not valid when `output_format=hec`.

output_format

Syntax: output_format=[raw | hec]

Description: Specifies the output format for the summary indexing. If set to `raw`, uses the traditional non-structured log style summary indexing stash output format.

If set to `hec`, it generates HTTP Event Collector (HEC) JSON formatted output:

◊ All fields are automatically indexed when the stash file is indexed.

- ◊ The file that is written to the `var/spool/splunk` path ends in `.stash_hec` instead of `.stash`.
- ◊ Allows the source, sourcetype, and host from the original data to be used directly in the summary index. Does not re-map these fields to the `extract_host/extracted_sourcetype/...` path.
- ◊ No license is counted for the internal stash source type. License is counted when the original source type is used instead of stash in `output_mode=hec`.
- ◊ The `index` and `splunk_server` fields in the original data are ignored.
- ◊ You cannot use the `addtime`, `host`, `marker`, `source`, `sourcetype`, or `useLB` options when `output_format=hec`.

Default: raw

run_in_preview

Syntax: `run_in_preview=<bool>`

Description: Controls whether the `collect` command is enabled during preview generation. Generally, you do not want to insert preview results into the summary index, `run_in_preview=false`. In some cases, such as when a custom search command is used as part of the search, you might want to turn this on to ensure correct summary indexable previews are generated.

Default: false

spool

Syntax: `spool=<bool>`

Description: If set to true, the summary indexing file is written to the Splunk spool directory, where it is indexed automatically. If set to false, the file is written to the `$SPLUNK_HOME/var/run/splunk` directory. The file remains in this directory unless some form of further automation or administration is done. If you have Splunk Enterprise, you can use this command to troubleshoot summary indexing by dumping the output file to a location on disk where it will not be ingested as data.

Default: true

source

Syntax: `source=<string>`

Description: The name of the source that you want to specify for the events.

This option is not valid when `output_format=hec`.

sourcetype

Syntax: `sourcetype=<string>`

Description: The name of the source type that you want to specify for the events. By specifying a sourcetype outside of stash, you **will incur license usage**.

This option is not valid when `output_format=hec`.

Default: stash

testmode

Syntax: `testmode=<bool>`

Description: Toggle between testing and real mode. In testing mode the results are not written into the new index but the search results are modified to appear as they would if sent to the index.

Default: false

timeformat

Syntax: `timeformat=<string>`

Description: Controls the format of the timestamp that is written to the stash file before it is indexed. The `addtime` argument must be set to `true` for the same invocation of the command in order to take advantage of this functionality.

Use this argument only if you need precise control over the format of output files that the `collect` command generates.

This option is not valid when `output_format=hec`.

Default: `%m/%d/%Y %H:%M:%S %z`

`uselb`

Syntax: `uselb=<bool>`

Description: Controls how line breaks are used to split events.

- ◊ When set to `true`, the data that is ingested using the `collect` command is split into individual events. A string identical to the `LINE_BREAKER` setting defined for the `stash_new` source type in the `props.conf` file is used.
- ◊ When set to `false`, a simple line break is used to split events.
- ◊ Do not use this argument unless you are intentionally generating events with the `collect` command in a line-oriented format.
- ◊ This option is not valid when `output_format=hec`.
- ◊ NOTE: While the default behavior of the `collect` command is to use a `LINE_BREAKER` setting identical to that used in the `props.conf` file, the default `LINE_BREAKER` for the `collect` command is hardcoded. Changes to `props.conf` setting do NOT affect the behavior of the `uselb` option.

Default: `true`

Usage

The events are written to a file whose name format is: `random-num_events.stash`, unless overwritten, in a directory that your Splunk deployment is monitoring. If the events contain a `_raw` field, then this field is saved. If the events do not have a `_raw` field, one is created by concatenating all the fields into a comma-separated list of key=value pairs.

The `collect` command also works with real-time searches that have a time range of **All time**.

Events without timestamps

If you apply the `collect` command to events that do not have timestamps, the command designates a time for all of the events using the earliest (or minimum) time of the search range. For example, if you use the `collect` command over the past four hours (range: `-4h` to `+0h`), the command assigns a timestamp that is four hours prior to the time that the search was launched. The timestamp is applied to all of the events without a timestamp.

If you use the `collect` command with a time range of **All time** and the events do not have timestamps, the current system time is used for the timestamps.

For more information on summary indexing of data without timestamps, see Use summary indexing for increased reporting efficiency in the *Knowledge Manager Manual*.

Copying events to a different index

You can use the `collect` command to copy search results to another index. Construct a search that returns the data you want to copy, and pipe the results to the `collect` command. For example:

```
index=foo | ... | collect index=bar
```

This search writes the results into the `bar` index. The sourcetype is changed to `stash`.

You can specify a sourcetype with the `collect` command. However, specifying a sourcetype counts against your license, as if you indexed the data again.

Change how collect summarizes multivalue fields

By default, the `collect` command summarizes multivalue fields as multivalue fields. For example, when `collect` summarizes the multivalue field `alphabet = a, b, c`, it adds the following field to the summary index:

```
alphabet: "a  
          b  
          c"
```

However, you may prefer that `collect` break multivalue fields into separate field-value pairs when it adds them to a `_raw` field in a summary index. For example, if given the multivalue field `alphabet = a,b,c`, you can have the `collect` command add the following fields to a `_raw` event in the summary index: `alphabet = "a", alphabet = "b", alphabet = "c"`

If you prefer to have `collect` follow this multivalue field summarization format, set the `limits.conf` setting `format_multivalue_collect` to `true`.

Splunk Cloud Platform

To change the `format_multivalue_collect` setting in your `limits.conf` file, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `format_multivalue_collect` setting in your local `limits.conf` file and enable `collect` to break multivalue fields into separate fields, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in How to edit a configuration file in the Splunk Enterprise *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps:

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[collect]` stanza, set `format_multivalue_collect` to `true`.

The `collect` and `tstats` commands

The `collect` command does not segment data by **major breakers** and **minor breakers**, such as characters like spaces, square or curly brackets, parenthesis, semicolons, exclamation points, periods, and colons. As a result, if either major or minor breakers are found in value strings, Splunk software places quotation marks around field values when it adds events to the summary index. These extra quotation marks can cause problems for subsequent searches. In particular, field values that have quotation marks around them can't be used in `tstats` searches with the `PREFIX()` directive. This is because `PREFIX()` does not support major breakers like quotation marks.

For example, in the following search with the `collect` command, the field values in quotes include periods as minor breakers.

```
| makeresults | eval application="buttercupgames.com", version="2.0" | collect index=summary source=devtest
```

The search results look something like this.

_time	application	version
2021-12-07 11:43:48	buttercupgames.com	2.0

So far, that looks fine, right? Not exactly. Although there aren't any extra quotation marks around the field values `buttercupgames.com` and `2.0` that are displayed in the search results, you will see them if you look in summary index. To see what is in the summary index, run the following search:

```
index=summary source=devtest
```

Now you can see `version="2.0"` and `application="buttercupgames.com"`. The results look something like this:

Time	Event
12/7/21	12/07/2021 11:43:48 -0800, info_search_time=1638906228.401, version="2.0", application="buttercupgames.com"
11:43:48.000 AM	host = PF32198D source = devtest sourcetype = stash

If you want to run a `tstats` search with the `PREFIX()` directive using those field values with quotation marks that are collected in a summary index like our previous example, you will need to edit your `limits.conf` file. You can do this by changing the `collect_ignore_minor_breakers` setting in the `[collect]` stanza from the default to `true`.

Splunk Cloud Platform

To change the `collect_ignore_minor_breakers` setting in your `limits.conf` file, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `collect_ignore_minor_breakers` setting in your local `limits.conf` file, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in How to edit a configuration file in the Splunk Enterprise *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps:

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[collect]` stanza, add the line `collect_ignore_minor_breakers=true`.

Examples

1. Put "download" events into an index named "download count"

```
eventtype tag="download" | collect index=downloadcount
```

2. Collect statistics on VPN connects and disconnects

You want to collect hourly statistics on VPN connects and disconnects by country.

```
index=mysummary | geoip REMOTE_IP | eval country_source=if(REMOTE_IP_country_code=="US","domestic","foreign") | bin _time span=1h | stats count by _time, vpn_action, country_source | addinfo | collect index=mysummary marker="summary_type=vpn, summary_span=3600, summary_method=bin, search_name=\"vpn starts and stops\""
```

The `addinfo` command ensures that the search results contain fields that specify when the search was run to populate these particular index values.

See also

Commands

[overlap](#)
[sichart](#)
[sirare](#)
[sistats](#)
[sitimechart](#)
[sitop](#)
[tscollect](#)

concurrency

Description

Concurrency measures the number of events which have spans that overlap with the start of each event. Alternatively, this measurement represents the total number of events in progress at the time that each particular event started, including the event itself. This command does not measure the total number of events that a particular event overlapped with during its total span.

Syntax

```
concurrency duration=<field> [start=<field>] [output=<field>]
```

Required arguments

duration

Syntax: `duration=<field>`
Description: A field that represents a span of time. This field must be a numeric with the same units as the `start` field. For example, the `duration` field generated by the [transaction command](#) is in seconds (see Example 1), which can be used with the default of `_time` which is also in units of seconds.

Optional arguments

start

Syntax: `start=<field>`
Description: A field that represents the start time.

Default: _time

output

Syntax: output=<field>

Description: A field to write the resulting number of concurrent events.

Default: "concurrency"

Usage

The `concurrency` command is a dataset processing command. See [Command types](#).

An event X is concurrent with event Y if X.start is between Y.start and (Y.start + Y.duration)

If your events have a time that represents event completion and a span that represents the time before the completion, you need to subtract duration from the start time before the concurrency command:

```
... | eval new_start = start - duration | concurrency start=new_start duration=duration
```

Limits

There is a limitation on quantity of overlapping items. If the maximum tracked concurrency exceeds max_count, from the [concurrency] stanza in limits.conf, a warning will be produced in the UI / search output, and the values will be clamped, making them potentially inaccurate. This limit defaults to 10000000 or ten million.

Basic examples

1. Determine the number of overlapping HTTP requests

Determine the number of overlapping HTTP requests outstanding from browsers accessing splunkd at the time that each http request begins.

This relies on the fact that the timestamp of the logged message is the time that the request came in, and the 'spent' field is the number of milliseconds spent handling the request. As always, you must be an 'admin' user, or have altered your roles scheme in order to access the _internal index.

```
index=_internal sourcetype=splunkd_ui_access | eval spent_in_seconds = spent / 1000 | concurrency duration=spent_in_seconds
```

2. Calculate the number of concurrent events

Calculate the number of concurrent events for each event and emit as field 'foo':

```
... | concurrency duration=total_time output=foo
```

3. Use existing fields to specify the start time and duration

Calculate the number of concurrent events using the 'et' field as the start time and 'length' as the duration:

```
... | concurrency duration=length start=et
```

Extended examples

1. Count the transactions that occurred at the same time

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Use the duration or span of a transaction to count the number of other transactions that occurred at the same time.

```
sourcetype=access_* | transaction JSESSIONID clientip startswith="view" endswith="purchase" | concurrency duration=duration | eval duration=tosstring(duration,"duration")
```

- This search groups events into transactions if they have the same values of `JSESSIONID` and `clientip`. An event is the beginning of the transaction if the event contains the string "view". An event is the last event of the transaction if the event contains the string "purchase".
- The `transaction` command returns a field called `duration`.
- The transactions are then piped into the `concurrency` command, which counts the number of events that occurred at the same time based on the timestamp and `duration` of the transaction.
- The search also uses the `eval` command and the `tosstring()` function to reformat the values of the `duration` field to a more readable format, HH:MM:SS.

< Hide Fields	All Fields	i Time	Event
SELECTED FIELDS			
a host 3			
a source 3			
a sourcetype 1			
INTERESTING FIELDS			
a action 5			
a bytes 100+			
a categoryId 8			
a clientip 100+			
# closed_bxn 1			
# concurrency 4			
# date_hour 24			
# date_mday 8			
# date_minute 60			
a date_month 1			
a date_second 60			
a date_wday 7			
# date_year 1			
a date_zone 1			
a duration 14			
# eventcount 12			
# field_match_sum 12			
a file 14			

host = www3 | source = tutorialdata.zip://www3/access.log | sourcetype = access_combined_wcookie

host = www2 | source = tutorialdata.zip://www2/access.log | sourcetype = access_combined_wcookie

host = www2 | source = tutorialdata.zip://www2/access.log | sourcetype = access_combined_wcookie

To see the values in each transaction for the `JSESSIONID`, `clientip`, `concurrency`, and `duration` fields:

1. In the list of Interesting Fields, click the field name.
2. In the information box, for **Selected**, click **Yes**.
3. Select the next field in the list of Interesting Fields. The information box automatically refreshes. For **Selected**, click **Yes**.
4. Repeat these steps for every field you want to appear in the result list. The results should appear similar to the

	Time	Event
>	4/1/18 7:01:18.000 PM	84.34.159.23 - - [01/Apr/2018:19:01:18] "GET /oldlink?itemId=EST-19&JSESSIONID=SD1SL8FF9ADFF5125 HTTP 1.1" 200 2533 "http://www.buttercupgames.com/cart.do?action=view&itemId=EST-19&productId=WCSH-T02" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727; .NET CLR 3.0.4586.2152; .NET CLR 3.5.30729; InfoPath.1; .NET4.0C; .NET4.0E; MS-RTC LM 8)" 771 84.34.159.23 - - [01/Apr/2018:19:01:20] "POST /cart.do?action=purchase&itemId=EST-27&JSESSIONID=SD1SL8FF9ADFF5125 HTTP 1.1" 200 1936 "http://www.buttercupgames.com/cart.do?action=addtocart&itemId=EST-27&categoryid=STRATEGY&productId=OC-SG-G02" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; InfoPath.1; .NET4.0C; .NET4.0E; MS-RTC LM 8)" 798 JSESSIONID = SD1SL8FF9ADFF5125 : clientip = 84.34.159.23 : concurrency = 2 : duration = 00:00:02 host = www1 source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie
>	4/1/18 7:03:44.000 PM	59.99.230.91 - - [01/Apr/2018:19:03:44] "GET /category.screen?categoryId=SHOOTER&JSESSIONID=SD8SL6FF7ADFF5131 HTTP 1.1" 200 2640 "http://www.buttercupgames.com/cart.do?action=view&itemId=EST-6&productId=WCSH-G04" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; MS-RTC LM 8; InfoPath.2)" 335 59.99.230.91 - - [01/Apr/2018:19:03:45] "POST /cart.do?action=purchase&itemId=EST-19&JSESSIONID=SD8SL6FF7ADFF5131 HTTP 1.1" 200 1180 "http://www.buttercupgames.com/cart.do?action=addtocart&itemId=EST-19&categoryid=SHOOTER&productId=WCSH-G04" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; MS-RTC LM 8; InfoPath.2)" 881 JSESSIONID = SD8SL6FF7ADFF5131 : clientip = 59.99.230.91 : concurrency = 1 : duration = 00:00:01 host = www3 source = tutorialdata.zip://www3/access.log sourcetype = access_combined_wcookie

following image:

2. Count the purchases that occurred at the same time

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Use the time between each purchase to count the number of different purchases that occurred at the same time.

```
sourcetype=access_* action=purchase | delta _time AS timeDelta p=1 | eval timeDelta=abs(timeDelta) | concurrency duration=timeDelta
```

- This search uses the `delta` command and the `_time` field to calculate the time between one purchase event (`action=purchase`) and the purchase event immediately preceding it.
- The search renames this change in time as `timeDelta`.
- Some of the values of `timeDelta` are negative. Because the `concurrency` command does not work with negative values, the `eval` command is used to redefine `timeDelta` as its absolute value (`abs(timeDelta)`).
- The `timeDelta` is then used as the `duration` for calculating concurrent events.

Hide Fields	All Fields	Time	Event
SELECTED FIELDS		> 4/1/18 6:22:21.000 PM	209.160.24.63 - - [01/Apr/2018:18:22:21] "POST /cart.do?action=purchase&itemId=EST-21&JSESSIONID=SD0SL6FF7ADFF4953 HTTP 1.1" 200 486 "http://www.buttercupgames.com/cart.do?action=addtocart&itemId=EST-21&categoryid=STRATEGY&productId=FS-SG-G03" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 293 JSESSIONID = SD0SL6FF7ADFF4953 : clientip = 209.160.24.63 : concurrency = 1 : host = www1 : source = tutorialdata.zip://www1/access.log : sourcetype = access_combined_wcookie timeDelta = 1
INTERESTING FIELDS		> 4/1/18 6:22:22.000 PM	209.160.24.63 - - [01/Apr/2018:18:22:22] "POST /cart/submit.success.do?JSESSIONID=SD0SL6FF7ADFF4953 HTTP 1.1" 200 3280 "http://www.buttercupgames.com/cart.do?action=purchase&itemId=EST-21" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 952 JSESSIONID = SD0SL6FF7ADFF4953 : clientip = 209.160.24.63 : concurrency = 1 : host = www1 : source = tutorialdata.zip://www1/access.log : sourcetype = access_combined_wcookie timeDelta = 256
		> 4/1/18 6:26:38.000 PM	112.111.162.4 - - [01/Apr/2018:18:26:38] "POST /cart/error.do?msg=CreditDoesNotMatch&JSESSIONID=SD7SL8FF5ADFF4964 HTTP 1.1" 200 1232 "http://www.buttercupgames.com/cart.do?action=purchase&itemId=EST-18" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5" 841 JSESSIONID = SD7SL8FF5ADFF4964 : clientip = 112.111.162.4 : concurrency = 1 : host = www1 : source = tutorialdata.zip://www1/access.log : sourcetype = access_combined_wcookie timeDelta = 72

3. Calculate the transactions using the time between consecutive transactions

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Use the time between each consecutive transaction to calculate the number of transactions that occurred at the same time.

```
sourcetype=access_* | transaction JSESSIONID clientip startswith="view" endswith="purchase" | delta _time AS timeDelta p=1 | eval timeDelta=abs(timeDelta) | concurrency duration=timeDelta | eval timeDelta=toString(timeDelta,"duration")
```

- This search groups events into transactions if they have the same values of `JSESSIONID` and `clientip`. An event is the beginning of the transaction if the event contains the string "view". An event is the last event of the transaction if the event contains the string "purchase".
- The `transaction` command returns a field called `duration`.
- The transactions are then piped into the `delta` command, which uses the `_time` field to calculate the time between one transaction and the transaction immediately preceding it.
- The search renames this change in time as `timeDelta`.
- Some of the values of `timeDelta` are negative. Because the `concurrency` command does not work with negative values, the `eval` command is used to redefine `timeDelta` as its absolute value (`abs(timeDelta)`).
- This `timeDelta` is then used as the `duration` for calculating concurrent transactions.

Time	Event
4/1/18 6:27:50.000 PM	74.125.19.106 - - [01/Apr/2018:18:27:50] "GET /oldlink?itemId=EST-12&JSESSIONID=SD10SL4FF4ADFF4976 HTTP/1.1" 400 1224 "http://www.buttercupgames.com/cart.do?action=view&itemId=EST-12" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 774 74.125.19.106 - - [01/Apr/2018:18:27:51] "POST /cart/error.do?msg=CreditIdDoesNotMatch&JSESSIONID=SD10SL4FF4ADFF4976 HTTP/1.1" 200 2934 "http://www.buttercupgames.com/cart.do?action=purchase&itemId=EST-26" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 774 JSESSIONID = SD10SL4FF4ADFF4976 : clientip = 74.125.19.106 : concurrency = 1 : duration = 1 : host = www3 : source = tutorialdata.zip:/www3/access.log : sourcetype = access_combined_wcookie : timeDelta = 00:16:52
4/1/18 6:44:42.000 PM	175.44.24.82 - - [01/Apr/2018:18:44:42] "POST /cart.do?action=view&itemId=EST-21&productId=WC-SH-G04&JSESSIONID=SD75L9FF5ADFF5066 HTTP/1.1" 200 675 "http://www.buttercupgames.com/oldlink?itemId=EST-21" "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; BOIE9;ENUS)" 142 175.44.24.82 - - [01/Apr/2018:18:44:44] "POST /cart.do?action=purchase&itemId=EST-12&JSESSIONID=SD75L9FF5ADFF5066 HTTP/1.1" 200 2399 "http://www.buttercupgames.com/cart.do?action=addtocart&siteId=EST-12&categoryId=STRATEGY&productId=DC-SG-G02" "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; BOIE9;ENUS)" 594 JSESSIONID = SD75L9FF5ADFF5066 : clientip = 175.44.24.82 : concurrency = 1 : duration = 2 : host = www2 : source = tutorialdata.zip:/www2/access.log : sourcetype = access_combined_wcookie : timeDelta = 00:15:12

See also

[timechart](#)

contingency

Description

In statistics, contingency tables are used to record and analyze the relationship between two or more (usually categorical) variables. Many metrics of association or independence, such as the *phi coefficient* or the *Cramer's V*, can be calculated based on contingency tables.

You can use the `contingency` command to build a contingency table, which in this case is a co-occurrence matrix for the values of two fields in your data. Each cell in the matrix displays the count of events in which both of the cross-tabulated field values exist. This means that the first row and column of this table is made up of values of the two fields. Each cell in the table contains a number that represents the count of events that contain the two values of the field in that row and column combination.

If a relationship or pattern exists between the two fields, you can spot it easily just by analyzing the information in the table. For example, if the column values vary significantly between rows (or vice versa), there is a contingency between the two fields (they are not independent). If there is no contingency, then the two fields are independent.

Syntax

`contingency [<contingency-options>...] <field1> <field2>`

Required arguments

`<field1>`

Syntax: `<field>`

Description: Any field. You cannot specify wildcard characters in the field name.

`<field2>`

Syntax: `<field>`

Description: Any field. You cannot specify wildcard characters in the field name.

Optional arguments

`contingency-options`

Syntax: `<maxopts> | <mincover> | <usetotal> | <totalstr>`

Description: Options for the contingency table.

Contingency options

`maxopts`

Syntax: `maxrows=<int> | maxcols=<int>`

Description: Specify the maximum number of rows or columns to display. If the number of distinct values of the field exceeds this maximum, the least common values are ignored. A value of 0 means a maximum limit on rows

or columns. This limit comes from the `maxvalues` setting in the `[ctable]` stanza in the `limits.conf` file.

Default: 1000

`mincover`

Syntax: `mincolcover=<num> | minrowcover=<num>`

Description: Specify a percentage of values per column or row that you would like represented in the output table. As the table is constructed, enough rows or columns are included to reach this ratio of displayed values to total values for each row or column. The maximum rows or columns take precedence if those values are reached.

Default: 1.0

`usetotal`

Syntax: `usetotal=<bool>`

Description: Specify whether or not to add row, column, and complete totals.

Default: true

`totalstr`

Syntax: `totalstr=<field>`

Description: Field name for the totals row and column.

Default: TOTAL

Usage

The `contingency` command is a transforming command. See [Command types](#).

This command builds a contingency table for two fields. If you have fields with many values, you can restrict the number of rows and columns using the `maxrows` and `maxcols` arguments.

Totals

By default, the contingency table displays the row totals, column totals, and a grand total for the counts of events that are represented in the table. If you don't want the totals to appear in the results, include the `usetotal=false` argument with the `contingency` command.

Empty values

Values which are empty strings ("") will be represented in the results table as EMPTY_STR.

Limits

There is a limit on the value of `maxrows` or `maxcols`, which means more than 1000 values for either field will not be used.

Examples

1. Build a contingency table of recent data

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days. Use the time range **All time** when you run the searches.

You want to build a contingency table to look at the relationship between the magnitudes and depths of recent earthquakes. You start with a simple search.

```
source=all_month.csv | contingency mag depth | sort mag
```

There are quite a range of values for the `Magnitude` and `Depth` fields, which results in a very large table. The magnitude values appear in the first column. The depth values appear in the first row. The list is sorted by magnitude.

The results appear on the Statistics tab. The following table shows only a small portion of the table of results returned from the search.

mag	10	0	5	35	8	12	15	11.9	11.8	6.4	5.4	8.2	6.5	8.1	5.6	10.1	9	8.5	9.8	8.7	7.9
-0.81	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-0.59	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-0.56	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-0.45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-0.43	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

As you can see, earthquakes can have negative magnitudes. Only where an earthquake occurred that matches the magnitude and depth will a count appear in the table.

To build a more usable contingency table, you should reformat the values for the magnitude and depth fields. Group the magnitudes and depths into ranges.

```
source=all_month.csv | eval Magnitude=case(mag<=1, "0.0 - 1.0", mag>1 AND mag<=2, "1.1 - 2.0", mag>2 AND mag<=3, "2.1 - 3.0", mag>3 AND mag<=4, "3.1 - 4.0", mag>4 AND mag<=5, "4.1 - 5.0", mag>5 AND mag<=6, "5.1 - 6.0", mag>6 AND mag<=7, "6.1 - 7.0", mag>7, "7.0+") | eval Depth=case(depth<=70, "Shallow", depth>70 AND depth<=300, "Mid", depth>300 AND depth<=700, "Deep") | contingency Magnitude Depth | sort Magnitude
```

This search uses the `eval` command with the `case()` function to redefine the values of Magnitude and Depth, bucketing them into a range of values. For example, the Depth values are redefined as "Shallow", "Mid", or "Deep". Use the `sort` command to sort the results by magnitude. Otherwise the results are sorted by the row totals.

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

Magnitude	Shallow	Mid	Deep	TOTAL
0.0 - 1.0	3579	33	0	3612
1.1 - 2.0	3188	596	0	3784
2.1 - 3.0	1236	131	0	1367
3.1 - 4.0	320	63	1	384
4.1 - 5.0	400	157	43	600
5.1 - 6.0	63	12	3	78
6.1 - 7.0	2	2	1	5
TOTAL	8788	994	48	9830

There were a lot of quakes in this month. Do higher magnitude earthquakes have a greater depth than lower magnitude earthquakes? Not really. The table shows that the majority of the recent earthquakes in all of magnitude ranges were

shallow. There are significantly fewer earthquakes in the mid-to-deep range. In this data set, the deep-focused quakes were all in the mid-range of magnitudes.

2. Identify potential component issues in the Splunk deployment

Determine if there are any components that might be causing issues in your Splunk deployment. Build a contingency table to see if there is a relationship between the values of `log_level` and `component`. Run the search using the time range **All time** and limit the number of columns returned.

```
index=_internal | contingency maxcols=5 log_level component
```

Your results should appear something like this:

The screenshot shows a Splunk search interface with the following details:

- Search Bar:** index=_internal | contingency log_level component maxcols=5
- Time Range:** All time
- Results Count:** 1,776,047 events (before 5/22/18 10:55:43.000 AM)
- Event Sampling:** No Event Sampling
- Statistics Tab:** Selected (5 results)
- Preview:** 20 Per Page
- Table Headers:** log_level, Metrics, HotDBManager, pipeline, PeriodicHealthReporter, root, TOTAL
- Data Rows:**

log_level	Metrics	HotDBManager	pipeline	PeriodicHealthReporter	root	TOTAL
INFO	1039556	318	0	57038	935	1101239
ERROR	0	302768	302762	0	0	605797
WARN	0	0	0	0	0	652
WARNING	0	0	0	0	0	74
TOTAL	1039556	303078	302762	57038	935	1707762

These results show you any components that might be causing issues in your Splunk deployment. The `component` field has more than 50 values. In this search, the `maxcols` argument is used to show 5 components with the highest values.

See also

[associate](#), [correlate](#)

convert

Description

The `convert` command converts field values in your search results into numerical values. Unless you use the AS clause, the original values are replaced by the new values.

Alternatively, you can use [evaluation functions](#) such as `strftime()`, `strptime()`, or `tonumber()` to convert field values.

Syntax

```
convert [timeformat=string] (<convert-function> [AS <field>] )...
```

Required arguments

<convert-function>

Syntax: `auto()` | `ctime()` | `dur2sec()` | `memk()` | `mktme()` | `mstime()` | `none()` | `num()` | `rmcomma()` | `rmunit()`

Description: Functions to use for the conversion.

Optional arguments

timeformat

Syntax: timeformat=<string>

Description: Specify the output format for the converted time field. The `timeformat` option is used by `ctime` and `mktme` functions. For a list and descriptions of format options, see [Common time format variables](#) in the [Search Reference](#).

Default: `%m/%d/%Y %H:%M:%S`. Note that this default does not conform to the locale settings.

<field>

Syntax: <string>

Description: Creates a new field with the name you specify to place the converted values into. The original field and values remain intact.

Convert functions

auto()

Syntax: auto(<wc-field>)

Description: Automatically convert the fields to a number using the best conversion. Note that if not all values of a particular field can be converted using a known conversion type, the field is left untouched and no conversion at all is done for that field. You can use a wildcard (*) character to specify all fields.

ctime()

Syntax: ctime(<wc-field>)

Description: Convert a UNIX time to an ASCII human readable time. Use the `timeformat` option to specify the exact format to convert to. You can use a wildcard (*) character to specify all fields.

dur2sec()

Syntax: dur2sec(<wc-field>)

Description: Convert a duration format "[D+]HH:MM:SS" to seconds. You can use a wildcard (*) character to specify all fields.

memk()

Syntax: memk(<wc-field>)

Description: Accepts a positive number (integer or float) followed by an optional "k", "m", or "g". The letter k indicates kilobytes, m indicates megabytes, and g indicates gigabytes. If no letter is specified, kilobytes is assumed. The output field is a number expressing quantity of kilobytes. Negative values cause data incoherency. You can use a wildcard (*) character to specify all fields.

mktme()

Syntax: mktme(<wc-field>)

Description: Convert a human readable time string to an epoch time. Use `timeformat` option to specify exact format to convert from. You can use a wildcard (*) character to specify all fields.

mstime()

Syntax: mstime(<wc-field>)

Description: Convert a [MM:]SS.SSS format to seconds. You can use a wildcard (*) character to specify all fields.

none()

Syntax: none(<wc-field>)

Description: In the presence of other wildcards, indicates that the matching fields should not be converted. You can use a wildcard (*) character to specify all fields.

num()

Syntax: num(<wc-field>)

Description: Like auto(), except non-convertible values are removed. You can use a wildcard (*) character to specify all fields.

rmcomma()

Syntax: rmcomma(<wc-field>)

Description: Removes all commas from value, for example rmcomma(1,000,000.00) returns 1000000.00. You can use a wildcard (*) character to specify all fields.

rmunit()

Syntax: rmunit(<wc-field>)

Description: Looks for numbers at the beginning of the value and removes trailing text. You can use a wildcard (*) character to specify all fields.

Usage

The `convert` command is a distributable streaming command. See [Command types](#).

Basic examples

1. Convert all field values to numeric values

Use the `auto` convert function to convert all field values to numeric values.

```
... | convert auto(*)
```

2. Convert field values except for values in specified fields

Convert every field value to a number value except for values in the field `src_ip`. Use the `none` convert function to specify fields to ignore.

```
... | convert auto(*) none(src_ip)
```

3. Change the duration values to seconds for the specified fields

Change the duration values to seconds for the specified fields

```
... | convert dur2sec(xdelay) dur2sec(delay)
```

4. Change the sendmail syslog duration format to seconds

Change the sendmail syslog duration format (D+HH:MM:SS) to seconds. For example, if `delay="00:10:15"`, the resulting value is `delay="615"`. This example uses the `dur2sec` convert function.

```
... | convert dur2sec(delay)
```

5. Convert field values that contain numeric and string values

Convert the values in the `duration` field, which contain numeric and string values, to numeric values by removing the string portion of the values. For example, if `duration="212 sec"`, the resulting value is `duration="212"`. This example uses the `rmunit` convert function.

```
... | convert rmunit(duration)
```

6. Change memory values to kilobytes

Change all memory values in the `virt` field to KBs. This example uses the `memk` convert function.

```
... | convert memk(virt)
```

Extended Examples

1. Convert a UNIX time to a more readable time format

Convert a UNIX time to a more readable time formatted to show hours, minutes, and seconds.

```
source="all_month.csv" | convert timeformat="%H:%M:%S" ctime(_time) AS c_time | table _time, c_time
```

- The `ctime()` function converts the `_time` value in the CSV file events to the format specified by the `timeformat` argument.
- The `timeformat="%H:%M:%S"` argument tells the search to format the `_time` value as HH:MM:SS.
- The converted time `ctime` field is renamed `c_time`.
- The `table` command is used to show the original `_time` value and the `ctime` field.

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

<code>_time</code>	<code>c_time</code>
2018-03-27 17:20:14.839	17:20:14
2018-03-27 17:21:05.724	17:21:05
2018-03-27 17:27:03.790	17:27:03
2018-03-27 17:28:41.869	17:28:41
2018-03-27 17:34:40.900	17:34:40
2018-03-27 17:38:47.120	17:38:47
2018-03-27 17:40:10.345	17:40:10
2018-03-27 17:41:55.548	17:41:55

The `ctime()` function changes the timestamp to a non-numerical value. This is useful for display in a report or for readability in your events list.

2. Convert a time in MM:SS.SSS to a number in seconds

Convert a time in MM:SS.SSS (minutes, seconds, and subseconds) to a number in seconds.

```
sourcetype=syslog | convert mstime(_time) AS ms_time | table _time, ms_time
```

- The `mstime()` function converts the `_time` field values from a minutes and seconds to just seconds.

The converted time field is renamed `ms_time`.

- The `table` command is used to show the original `_time` value and the converted time.

<code>_time</code>	<code>ms_time</code>
2018-03-27 17:20:14.839	1522196414.839
2018-03-27 17:21:05.724	1522196465.724
2018-03-27 17:27:03.790	1522196823.790
2018-03-27 17:28:41.869	1522196921.869
2018-03-27 17:34:40.900	1522197280.900
2018-03-27 17:38:47.120	1522197527.120
2018-03-27 17:40:10.345	1522197610.345
2018-03-27 17:41:55.548	1522197715.548

The `mstime()` function changes the timestamp to a numerical value. This is useful if you want to use it for more calculations.

3. Convert a string time in HH:MM:SS into a number

Convert a string field `time_elapsed` that contains times in the format HH:MM:SS into a number. Sum the `time_elapsed` by the `user_id` field. This example uses the `eval` command to convert the converted results from seconds into minutes.

```
... | convert num(time_elapsed) | stats sum(eval(time_elapsed/60)) AS Minutes BY user_id
```

See also

Commands

[eval](#)
[fieldformat](#)

Functions

[tonumber](#)
[strptime](#)

correlate

Description

Calculates the correlation between different fields.

You can use the `correlate` command to see an overview of the co-occurrence between fields in your data. The results are presented in a matrix format, where the cross tabulation of two fields is a cell value. The cell value represents the percentage of times that the two fields exist in the same events.

The field the result is specific to is named in the value of the `RowField` field, while the fields it is compared against are the names of the other fields.

Note: This command looks at the relationship among all the fields in a set of search results. If you want to analyze the relationship between the values of fields, refer to the [contingency](#) command, which counts the co-occurrence of pairs of field values in events.

Syntax

correlate

Limits

There is a limit on the number of fields that `correlate` considers in a search. From `limits.conf`, stanza [correlate], the `maxfields` sets this ceiling. The default is 1000.

If more than this many fields are encountered, the `correlate` command continues to process data for the first N (eg thousand) field names encountered, but ignores data for additional fields. If this occurs, the notification from the search or alert contains a message "correlate: input fields limit (N) reached. Some fields may have been ignored."

As with all designed-in limits, adjusting this might have significant memory or cpu costs.

Examples

Example 1:

Look at the co-occurrence between all fields in the `_internal` index.

```
index=_internal | correlate
```

Here is a snapshot of the results.

RowField	abandoned_channels	actions_triggered	active_hist_searches	active_realtime_searches	average_kbps	avg_age
1 abandoned_channels	1.00	0.00	0.00	0.00	0.00	0.00
2 actions_triggered	0.00	1.00	0.00	0.00	0.00	0.00
3 active_hist_searches	0.00	0.00	1.00	1.00	0.00	0.00
4 active_realtime_searches	0.00	0.00	1.00	1.00	0.00	0.00
5 average_kbps	0.00	0.00	0.00	0.00	1.00	0.00
6 avg_age	0.00	0.00	0.00	0.00	0.00	1.00
7 browser	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.00

Because there are different types of logs in the `_internal`, you can expect to see that many of the fields do not co-occur.

Example 2:

Calculate the co-occurrences between all fields in Web access events.

```
sourcetype=access_* | correlate
```

You expect all Web access events to share the same fields: clientip, referer, method, and so on. But, because the `sourcetype=access_*` includes both `access_common` and `access_combined` Apache log formats, you should see that the percentages of some of the fields are less than 1.0.

Example 3:

Calculate the co-occurrences between all the fields in download events.

```
eventtype=download | correlate
```

The more narrow your search is before you pass the results into `correlate`, the more likely it is that all the field value pairs have a correlation of 1.0. A correlation of 1.0 means the values co-occur in 100% of the search results. For these download events, you might be able to spot an issue depending on which pairs have less than 1.0 co-occurrence.

See also

[associate](#), [contingency](#)

ctable

The `ctable`, or `counttable`, command is an alias for the `contingency` command. See the [contingency command](#) for the syntax and examples.

datamodel

Description

Examine and search data model datasets.

Use the `datamodel` command to return the JSON for all or a specified data model and its datasets. You can also search against the specified data model or a dataset within that datamodel.

A data model is a hierarchically-structured search-time mapping of semantic knowledge about one or more datasets. A data model encodes the domain knowledge necessary to build a variety of specialized searches of those datasets. For more information, see [About data models](#) and [Design data models](#) in the *Knowledge Manager Manual*.

The `datamodel` search command lets you search existing data models and their datasets from the search interface.

The `datamodel` command is a generating command and should be the first command in the search. Generating commands use a leading pipe character.

Syntax

```
| datamodel [<data model name>] [<dataset name>] [<data model search mode>] [strict_fields=<bool>]  
[allow_old_summaries=<bool>] [summariesonly=<bool>]
```

Required arguments

None

Optional arguments

data model name

Syntax: <string>

Description: The name of the data model to search. When only the data model is specified, the search returns the JSON for the single data model.

dataset name

Syntax: <string>

Description: The name of a data model dataset to search. Must be specified after the data model name. The search returns the JSON for the single dataset.

data model search mode

Syntax: <data model search result mode> | <data model search string mode>

Description: You can use `datamodel` to run a search against a data model or a data model dataset that returns either results or a search string. If you want to do this, you must provide a <`data model search mode`>. There are two <`data model search mode`> subcategories: modes that return results and modes that return search strings. See [<data model search mode> options](#).

allow_old_summaries

Syntax: `allow_old_summaries=<bool>`

Description: This argument applies only to accelerated data models. When you change the constraints that define a data model but the Splunk software has not fully updated the summaries to reflect that change, the summaries may have some data that matches the old definition and some data that matches the new definition. By default, `allow_old_summaries = false`, which means that the search head does not use summary directories that are older than the new summary definition. This ensures that the `datamodel` search results always reflect your current configuration. When you set `allow_old_summaries = true`, `datamodel` uses both current summary data and summary data that was generated prior to the definition change. You can set `allow_old_summaries=true` in your search if you feel that the old summary data is close enough to the new summary data that its results are reliable.

Default: false

summariesonly

Syntax: `summariesonly=<bool>`

Description: This argument applies only to accelerated data models. When set to false, the `datamodel` search returns both summarized and unsummarized data for the selected data model. When set to true, the search returns results only from the data that has been summarized in TSIDX format for the selected data model. You can use this argument to identify what data is currently summarized for a given data model, or to ensure that a particular data model search runs efficiently.

Default: false

strict_fields

Syntax: `strict_fields=<bool>`

Description: Determines the scope of the `datamodel` search in terms of fields returned. When `strict_fields=true`, the search returns only default fields and fields that are included in the constraints of the specified data model dataset. When `strict_fields=false`, the search returns all fields defined in the data model, including fields inherited from parent data model datasets, extracted fields, calculated fields, and fields derived from lookups.

You can also arrange for `strict_fields` to default to `false` for a specific data model. See Design data models in the *Knowledge Manager Manual*.

Default: true

<data model search mode> options

data model search result mode

Syntax: search | flat | acceleration_search

Description: The modes for running searches on a data model or data model dataset that return results.

Mode	Description
search	Returns the search results exactly how they are defined.
flat	Returns the same results as the search, except that it strips the hierarchical information from the field names. For example, where search mode might return a field named <code>dmdataset.server</code> , the flat mode returns a field named <code>server</code> .
acceleration_search	Runs the search that the search head uses to accelerate the data model. This mode works only on root event datasets and root search datasets that only use streaming commands.

data model search string mode

Syntax: search_string | flat_string | acceleration_search_string

Description: These modes return the strings for the searches that the Splunk software is actually running against the data model when it runs your SPL through the corresponding <data model search result mode>. For example, if you choose `acceleration_search_string`, the Splunk software returns the search string it would actually use against the data model when you run your SPL through `acceleration_search` mode.

Usage

The `datamodel` command is a report-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Examples

1. Return the JSON for all data models

Return JSON for all data models available in the current app context.

```
| datamodel
```

i	Time	Event
>	{ [-] description: Splunk's Internal Audit Logs record user activity, including searches and configuration changes. displayName: Splunk's Internal Audit Logs - SAMPLE modelName: internal_audit_logs objectNameList: [[+]] objectSummary: { [+] } objects: [[+]] } Show as raw text	
>	{ [-] description: Splunk's Internal Server Logs record information about system usage and performance. displayName: Splunk's Internal Server Logs - SAMPLE modelName: internal_server objectNameList: [[+]] objectSummary: { [+] } objects: [[+]] } Show as raw text	

2. Return the JSON for a specific datamodel

Return JSON for the **Splunk's Internal Audit Logs - SAMPLE** data model, which has the model ID **internal_audit_logs**.

```
| datamodel internal_audit_logs
```

i	Time	Event
>	{ [-] description: Splunk's Internal Audit Logs record user activity, including searches and configuration changes. displayName: Splunk's Internal Audit Logs - SAMPLE modelName: internal_audit_logs objectNameList: [[+]] objectSummary: { [+] } objects: [[+]] } Show as raw text	

3. Return the JSON for a specific dataset

Return JSON for Buttercup Games's Client_errors dataset.

```
| datamodel Tutorial Client_errors
```

4. Run a search on a specific dataset

Run the search for Buttercup Games's Client_errors.

```
| datamodel Tutorial Client_errors search
```

5. Run a search on a dataset for specific criteria

Search Buttercup Games's Client_errors dataset for 404 errors and count the number of events.

```
| datamodel Tutorial Client_errors search | search Tutorial.status=404 | stats count
```

6. For an accelerated data model, reveal what data has been summarized over a selected time range

After the Tutorial data model is accelerated, this search uses the `summariesonly` argument in conjunction with `timechart` to reveal what data has been summarized for the Client_errors dataset over a selected time range.

```
| datamodel Tutorial summariesonly=true search | timechart span=1h count
```

See also

[pivot](#)

datamodelsimple

The `datamodelsimple` command is used with the Splunk Common Information Model Add-on.

For information about this command, see Use the `datamodelsimple` command in the *Common Information Model Add-on Manual*.

dbinspect

Description

Returns information about the buckets in the specified index. If you are using Splunk Enterprise, this command helps you understand where your data resides so you can optimize disk usage as required. Searches on an indexer cluster return results from the primary buckets and replicated copies on other peer nodes.

The Splunk **index** is the repository for data ingested by Splunk software. As incoming data is indexed and transformed into **events**, Splunk software creates files of **rawdata** and metadata (**index files**). The files reside in sets of directories organized by age. These directories are called **buckets**.

For more information, see Indexes, indexers, and clusters and How the indexer stores indexes in *Managing Indexers and Clusters of Indexers*.

Syntax

The required syntax is in **bold**.

```
| dbinspect  
[index=<wc-string>]...  
[<span> | <timeformat>]  
[corruptonly=<bool>]  
[cached=<bool>]
```

Required arguments

None.

Optional arguments

index

Syntax: index=<wc-string>...

Description: Specifies the name of an index to inspect. You can specify more than one index. For all internal and non-internal indexes, you can specify an asterisk (*) in the index name.

Default: The default index, which is typically **main**.

Syntax: span=<int> | span=<int><timescale>

Description: Specifies the span length of the bucket. If using a timescale unit (second, minute, hour, day, month, or subseconds), this is used as a time range. If not, this is an absolute bucket "length".

When you invoke the `dbinspect` command with a bucket span, a table of the spans of each bucket is returned.

When `span` is not specified, information about the buckets in the index is returned. See [Information returned when no span is specified](#).

<timeformat>

Syntax: timeformat=<string>

Description: Sets the time format for the `modTime` field.

Default: `timeformat=%m/%d/%Y:%H:%M:%S`

<corruptonly>

Syntax: corruptonly=<bool>

Description: Specifies that each bucket is checked to determine if any buckets are corrupted and displays only the corrupted buckets. A bucket is corrupt when some of the files in the bucket are incorrect or missing such as `Hosts.data` or `tsidx`. A corrupt bucket might return incorrect data or render the bucket unsearchable. In most cases the software will auto-repair corrupt buckets.

When `corruptonly=true`, each bucket is checked and the following informational message appears.

Not supported on Splunk SmartStore indexes.

INFO: The "corruptonly" option will check each of the specified buckets. This search might be slow and will take time.

Default: false

cached

Syntax: cached=<bool>

Description: If set to `cached=true`, the `dbinspect` command gets the statistics from the bucket's manifest. If set to `cached=false`, the `dbinspect` command examines the bucket itself. For SmartStore buckets, `cached=false` examines an indexer's local copy of the bucket. However, specifying `cached=true` examines instead the bucket's manifest, which contains information about the canonical version of the bucket that resides in the remote store. For more information see Troubleshoot SmartStore in *Managing Indexers and Clusters of Indexers*.

Default: For non-SmartStore indexes, the default is `false`. For SmartStore indexes, the default is `true`.

Time scale units

These are options for specifying a timescale as the bucket span.

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month> | <subseconds>

Description: Time scale units.

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

Information returned when no span is specified

When you invoke the `dbinspect` command without the `span` argument, the following information about the buckets in the index is returned.

Field name	Description
bucketId	A string comprised of <index>~<id>~<guId>, where the delimiters are tilde characters. For example, summary~2~4491025B-8E6D-48DA-A90E-89AC3CF2CE80.
endEpoch	The timestamp for the last event in the bucket, which is the time-edge of the bucket furthest towards the future. Specify the timestamp in the number of seconds from the UNIX epoch.
eventCount	The number of events in the bucket.
guId	The globally unique identifier (GUID) of the server that hosts the index. This is relevant for index replication.
hostCount	The number of unique hosts in the bucket.
id	The local ID number of the bucket, generated on the indexer on which the bucket originated.
index	The name of the index specified in your search. You can specify <code>index=*</code> to inspect all of the indexes, and the <code>index</code> field will vary accordingly.
modTime	The timestamp for the last time the bucket was modified or updated, in a format specified by the <code>timeformat</code> flag.
path	<p>The location to the bucket. The naming convention for the bucket <code>path</code> varies slightly, depending on whether the bucket rolled to warm while its indexer was functioning as a cluster peer:</p> <ul style="list-style-type: none">For non-clustered buckets: <code>db_<newest_time>_<oldest_time>_<localid></code>For clustered original bucket copies: <code>db_<newest_time>_<oldest_time>_<localid>_<guid></code>For clustered replicated bucket copies: <code>rb_<newest_time>_<oldest_time>_<localid>_<guid></code> <p>For more information, read "How Splunk stores indexes" and "Basic cluster architecture" in <i>Managing Indexers and Clusters of Indexers</i>.</p>
rawSize	The volume in bytes of the raw data files in each bucket. This value represents the volume before compression and the addition of index files.
sizeOnDiskMB	The size in MB of disk space that the bucket takes up expressed as a floating point number. This value represents the volume of the compressed raw data files and the index files.

Field name	Description
sourceCount	The number of unique sources in the bucket.
sourceTypeCount	The number of unique sourcetypes in the bucket.
splunk_server	The name of the Splunk server that hosts the index in a distributed environment.
startEpoch	The timestamp for the first event in the bucket (the time-edge of the bucket furthest towards the past), in number of seconds from the UNIX epoch.
state	Specifies whether the bucket is warm, hot, cold.
tsidxState	Specifies whether each bucket contains full-size or reduced tsidx files. If the value of this field in the results is <code>full</code> , the tsidx files are full-size. If the value is <code>mini</code> , the tsidx files are reduced. See Determine whether a bucket is reduced in Splunk Enterprise <i>Managing Indexers and Clusters of Indexers</i> .
corruptReason	Specifies the reason why the bucket is corrupt. The <code>corruptReason</code> field appears only when <code>corruptiononly=true</code> .

Usage

The `dbinspect` command is a generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Accessing data and security

If no data is returned from the index that you specify with the `dbinspect` command, it is possible that you do not have the authorization to access that index. The ability to access data in the Splunk indexes is controlled by the authorizations given to each role. See Use access control to secure Splunk data in *Securing Splunk Enterprise*.

Non-searchable bucket copies

For hot non-searchable bucket copies on target peers, tsidx and other metadata files are not maintained. Because accurate information cannot be reported, the following fields show NULL:

- `eventCount`
- `hostCount`
- `sourceCount`
- `sourceTypeCount`
- `startEpoch`
- `endEpoch`

Examples

1. CLI use of the `dbinspect` command

Display a chart with the span size of 1 day, using the command line interface (CLI).

```
myLaptop $ splunk search "| dbinspect index=_internal span=1d"
```

```
_time          hot-3  warm-1  warm-2
-----
2015-01-17 00:00:00.000 PST      0
2015-01-17 14:56:39.000 PST      0
```

```
2015-02-19 00:00:00.000 PST          0      1
2015-02-20 00:00:00.000 PST          2      1
```

2. Default dbinspect output

Default dbinspect output for a local _internal index.

```
| dbinspect index=_internal
```

20 Per Page *		Format *	Preview *								
bucketId	endEpoch	eventCount	guid	hostCount	id	index	modTime	path	rawSize	sizeOnDiskMB	sourceCount
_internal-0~A11D93E6-	1381275242	555490	A11D93E6-		1	0	_internal	10/08/2013:16:34:20 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381275242_1379996500_0	102033831	44.742188	13
C13C-4192-			A832-9C4B3175273F								
_internal-1~A11D93E6-	1381275250	6310	A11D93E6-		1	1	_internal	10/08/2013:16:58:07 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381275250_1381242842_1	1315790	0.847656	10
C13C-4192-			A832-9C4B3175273F								
_internal-2~A11D93E6-	1381246070	7704	A11D93E6-		1	2	_internal	10/08/2013:17:28:07 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381246070_1381244270_2	1497111	0.964844	10
C13C-4192-			A832-9C4B3175273F								
_internal-3~A11D93E6-	1381753579	293107	A11D93E6-		1	3	_internal	10/14/2013:15:31:57 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381753579_1381245988_3	50589038	20.003906	12
C13C-4192-			A832-9C4B3175273F								
_internal-4~A11D93E6-	1381940762	171863	A11D93E6-		1	4	_internal	10/16/2013:18:26:21 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381940762_1381753580_4	30590658	13.128906	11
C13C-4192-			A832-9C4B3175273F								
_internal-5~A11D93E6-	1381941015	1572	A11D93E6-		1	5	_internal	10/16/2013:18:31:15 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381941015_1381940762_5	309869	0.312500	10
C13C-4192-			A832-9C4B3175273F								
_internal-6~A11D93E6-	1381941741	1967	A11D93E6-		1	6	_internal	10/16/2013:18:42:33 /Applications/splunk/var/lib/splunk/_internal/_internaldb/_db/_db_1381941741_1381941016_6	319026	0.281250	10
C13C-4192-			A832-9C4B3175273F								

This screen shot does not display all of the columns in the output table. On your computer, scroll to the right to see the other columns.

3. Check for corrupt buckets

Use the `corruptonly` argument to display information about corrupted buckets, instead of information about all buckets. The output fields that display are the same with or without the `corruptonly` argument.

```
| dbinspect index=_internal corruptonly=true
```

4. Count the number of buckets for each Splunk server

Use this command to verify that the Splunk servers in your distributed environment are included in the `dbinspect` command. Counts the number of buckets for each server.

```
| dbinspect index=_internal | stats count by splunk_server
```

5. Find the index size of buckets in GB

Use dbinspect to find the index size of buckets in GB. For current numbers, run this search over a recent time range.

```
| dbinspect index=_internal | eval GB=sizeOnDiskMB/1024| stats sum(GB)
```

6. Determine whether a bucket is reduced

Run the `dbinspect` search command:

```
| dbinspect index=_internal
```

If the value of the `tsidxState` field for each bucket is `full`, the tsidx files are full-size. If the value is `mini`, the tsidx files are reduced.

dbxquery

The dbxquery command is used with Splunk DB Connect.

For information about this command, see Execute SQL statements and stored procedures with the dbxquery command in *Deploy and Use Splunk DB Connect*.

dedup

Description

Removes the events that contain an identical combination of values for the fields that you specify.

With the `dedup` command, you can specify the number of duplicate events to keep for each value of a single field, or for each combination of values among several fields. Events returned by `dedup` are based on search order. For **historical searches**, the most recent events are searched first. For **real-time searches**, the first events that are received are searched, which are not necessarily the most recent events.

You can specify the number of events with duplicate values, or value combinations, to keep. You can sort the fields, which determines which event is retained. Other options enable you to retain events with the duplicate fields removed, or to keep events where the fields specified do not exist in the events.

Syntax

The required syntax is in **bold**.

```
dedup
[<int>]
<field-list>
[keepevents=<bool>]
[keepempty=<bool>]
[consecutive=<bool>]
[sortby <sort-by-clause>]
```

Required arguments

<field-list>

Syntax: <string> <string> ...

Description: A list of field names to remove duplicate values from.

Optional arguments

consecutive

Syntax: consecutive=<bool>

Description: If true, only remove events with duplicate combinations of values that are consecutive.

Default: false

keepempty

Syntax: keepempty=<bool>

Description: If set to true, keeps every event where one or more of the specified fields is not present (null).

Default: false. All events where any of the selected fields are null are dropped.

The `keepempty=true` argument keeps every event that does not have one or more of the fields in the field list. To keep N representative events for combinations of field values including null values, use the `fillnull` command to provide a non-null value for these fields. For example:

```
... | fillnull value="MISSING" field1 field2 | dedup field1 field2
```

keepevents

Syntax: keepevents=<bool>

Description: If true, keep all events, but will remove the selected fields from events after the first event containing a particular combination of values.

Default: false. Events are dropped after the first event of each particular combination.

<N>

Syntax: <int>

Description: The `dedup` command retains multiple events for each combination when you specify `N`. The number for `N` must be greater than 0. If you do not specify a number, only the first occurring event is kept. All other duplicates are removed from the results.

<sort-by-clause>

Syntax: sortby (- | +) <sort-field> [(- | +) <sort_field> ...]

Description: List of the fields to sort by and the sort order. Use the dash symbol (-) for descending order and the plus symbol (+) for ascending order. You must specify the sort order for each field specified in the <sort-by-clause>. The <sort-by-clause> determines which of the duplicate events to keep. When the list of events is sorted, the top-most event, of the duplicate events in the sorted list, is retained.

Sort field options

<sort-field>

Syntax: <field> | auto(<field>) | str(<field>) | ip(<field>) | num(<field>)

Description: The options that you can specify to sort the events.

<field>

Syntax: <string>

Description: The name of the field to sort.

auto

Syntax: auto(<field>)

Description: Determine automatically how to sort the field values.

ip

Syntax: ip(<field>)

Description: Interpret the field values as IP addresses.

num

Syntax: num(<field>)

Description: Interpret the field values as numbers.

str

Syntax: str(<field>)

Description: Order the field values by using the lexicographic order.

Usage

The `dedup` command is a streaming command or a dataset processing command, depending on which arguments are specified with the command. For example, if you specify the `<sort-by-clause>`, the `dedup` command acts as a dataset processing command. All of the results must be collected before sorting. See [Command types](#).

Avoid using the `dedup` command on the `_raw` field if you are searching over a large volume of data. If you search the `_raw` field, the text of every event in memory is retained which impacts your search performance. This is expected behavior. This behavior applies to any field with high cardinality and large size.

Multivalue fields

To use the `dedup` command on multivalue fields, the fields must match all values to be deduplicated.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Examples

1. Remove duplicate results based on one field

Remove duplicate search results with the same `host` value.

```
... | dedup host
```

2. Remove duplicate results and sort results in ascending order

Remove duplicate search results with the same `source` value and sort the results by the `_time` field in ascending order.

```
... | dedup source sortby +_time
```

3. Remove duplicate results and sort results in descending order

Remove duplicate search results with the same `source` value and sort the results by the `_size` field in descending order.

```
... | dedup source sortby -_size
```

4. Keep the first 3 duplicate results

For search results that have the same `source` value, keep the first 3 that occur and remove all subsequent results.

```
... | dedup 3 source
```

5. Keep results that have the same combination of values in multiple fields

For search results that have the same `source` AND `host` values, keep the first 2 that occur and remove all subsequent results.

```
... | dedup 2 source host
```

6. Remove only consecutive duplicate events

Remove only consecutive duplicate events. Keep non-consecutive duplicate events. In this example duplicates must have the same combination of values in the `source` and `host` fields.

```
... | dedup consecutive=true source host
```

See also

[uniq](#)

delete

Description

Using the `delete` command marks all of the events returned by the search as deleted. Subsequent searches do not return the marked events. No user, not even a user with admin permissions, is able to view this data after deletion. **The delete command does not reclaim disk space.**

Removing data is irreversible. If you want to get your data back after the data is deleted, you must re-index the applicable data sources.

You cannot run the `delete` command in a real-time search to delete events as they arrive.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

`delete`

Usage

The `delete` command can be accessed only by a user with the "delete_by_keyword" **capability**. By default, only the "can_delete" role has the ability to delete events. No other role, including the admin role, has this ability. You should create a special userid that you log on with when you intend to delete indexed data.

To use the `delete` command, run a search that returns the events you want deleted. Make sure that the search returns ONLY the events that you want to delete, and no other events. After you confirm that the results contain the data that you want to delete, pipe the search to the `delete` command.

The `delete` command does not trigger a roll of hot buckets to warm in the affected indexes.

The output of the `delete` command is a table of the quantity of events removed by the fields `splunk_server` (the name of the indexer or search head), and `index`, as well as a rollup record for each server by index "`__ALL__`". The quantity of deleted events is in the `deleted` field. An `errors` field is also emitted, which will normally be 0.

Delete command restrictions

The `delete` command does not work in all situations:

Searches with centralized streaming commands.

You cannot use the `delete` command after a centralized streaming command. For example, you can't delete events using a search like this:

```
index=myindex ... | head 100 | delete
```

Centralized streaming commands include: `head`, `streamstats`, some modes of `dedup`, and some modes of `cluster`. See [Command types](#).

Events with an `index` field.

If your events contain a field named `index` aside from the default `index` field that is applied to all events. If your events do contain an additional `index` field, you can use `eval` before invoking `delete`, as in this example:

```
index=fbus_summary latest=1417356000 earliest=1417273200 | eval index = "fbus_summary" | delete
```

Permanently removing data from an index

The `delete` command does not remove the data from your disk space. You must use the `clean` command from the CLI to permanently remove the data. The `clean` command removes all of the data in an index. **You cannot select the specific data that you want to remove.** See Remove indexes and indexed data in *Managing Indexers and Clusters of Indexers*.

Examples

1. Delete events with Social Security numbers

Delete the events from the `insecure` index that contain strings that look like Social Security numbers. Use the `regex` command to identify events that contain the strings that you want to match.

1. Run the following search to ensure that you are retrieving the correct data from the `insecure` index.

```
index=insecure | regex _raw = "\d{3}-\d{2}-\d{4}"
```

2. If necessary, adjust the search to retrieve the correct data. Then add the `delete` command to the end of the search to delete the events.

```
index=insecure | regex _raw = "\d{3}-\d{2}-\d{4}" | delete
```

2. Delete events that contain a specific word

Delete events from the `imap` index that contain the word `invalid`.

```
index=imap invalid | delete
```

3. Remove the Search Tutorial events

Remove all of the Splunk Search Tutorial events from your index.

1. Login as a user with an administrator role:
 - ◆ For Splunk Cloud Platform, the role is `sc_admin`.
 - ◆ For Splunk Enterprise, the role is `admin`.
2. Click **Settings > Users** and create a new user with the `can_delete` role.
3. Log out as the administrator and log back in as the user with the `can_delete` role.
4. Set the time range picker to `All time`.
5. Run the following search to retrieve all of the Search Tutorial events.

```
source=tutorialdata.zip:*
```

6. Confirm that the search is retrieving the correct data.
7. Add the `delete` command to the end of the search criteria and run the search again.

```
source=tutorialdata.zip:* | delete
```

The events are removed from the index.

8. Log out as the user with the `can_delete` role.

delta

Description

Computes the difference between nearby results using the value of a specific numeric field. For each event where `<field>` is a number, the `delta` command computes the difference, in search order, between the `<field>` value for the current event and the `<field>` value for the previous event. The `delta` command writes this difference into `<newfield>`.

Syntax

The required syntax is in **bold**.

```
delta
<field> [AS <newfield>]
[p=int]
```

Required arguments

field

Syntax: <field-name>

Description: The name of a field to analyze. If <field> is not a numeric field, no output field is generated.

Optional arguments

newfield

Syntax: <string>

Description: The name of a new field to write the output to.

Default: delta(<field>)

p

Syntax: p=<int>

Description: Specifies how many results prior to the current result to use for the comparison to the value in field in the current result. The prior results are determined by the search order, which is not necessarily chronological order. If p=1, compares the current result value against the value in the first result prior to the current result. If p=2, compares the current result value against the value in the result that is two results prior to the current result, and so on.

Default: 1

Usage

The `delta` command works on the events in the order they are returned by search. By default, the events for historical searches are in reverse time order from new events to old events.

Values ascending over time show negative deltas.

For real-time search, the events are compared in the order they are received.

The `delta` can be applied after any sequence of commands, so there is no input order guaranteed. For example, if you sort your results by an independent field and then use the `delta` command, the produced values are the deltas in that specific order.

Basic examples

1. Calculate the difference in activity

With the logs from a cable TV provider, `sourcetype=tv`, you can analyze broadcasting ratings, customer preferences, and so on. Which channels do subscribers watch the most, `activity=view`, and how long do the subscribers stay on those channels?

```
sourcetype=tv activity="View" | sort - _time | delta _time AS timeDeltas | eval timeDeltaS=abs(timeDeltas) | stats sum(timeDeltaS) by ChannelName
```

2. Calculate the difference between that current value and the 3rd previous value

Compute the difference between current value of count and the 3rd previous value of count and store the result in the default field, `delta(fieldname)`, which in this example is `delta(count)`.

```
... | delta count p=3
```

3. Calculate the difference between that current value and the previous value and rename the result field

For each event where 'count' exists, compute the difference between count and its previous value and store the result in the field **countdiff**.

```
... | delta count AS countdiff
```

Extended examples

1. Calculate the difference in the number of purchases between the top 10 buyers

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.

Find the top ten people who bought something yesterday, count how many purchases they made and the difference in the number of purchases between each buyer.

```
sourcetype=access_* status=200 action=purchase | top clientip | delta count p=1
```

- The purchase events, `action=purchase`, are piped into the `top` command to find the top ten users, based on `clientip`, who bought something.
- These results, which include a `count` for each `clientip` are then piped into the `delta` command to calculate the difference between the `count` value of one event and the `count` value of the event preceding it, using the `p=1` argument.
- By default, this difference is saved in a new field called **delta(count)**.
- The first event does not have a **delta(count)** value.

The results look something like this:

clientip	count	percent	delta(count)
87.194.216.51	134	2.565084	
128.241.220.82	95	1.818530	-39
211.166.11.101	91	1.741960	-4
107.3.146.207	72	1.378254	-19
194.215.205.19	60	1.148545	-12
109.169.32.135	60	1.148545	0
188.138.40.166	56	1.071975	-4
74.53.23.135	49	0.937979	-7
187.231.45.62	48	0.918836	-1
91.208.184.24	46	0.880551	-2

2. Calculate the difference in time between recent events

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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and add it as an input.

Calculate the difference in time between each of the recent earthquakes in Alaska. Run the search using the time range **All time**.

```
source=all_month.csv place=*alaska* | delta _time p=1 | rename delta(_time) AS timeDeltaS | eval
timeDeltaS=abs(timeDeltaS) | eval "Time Between Quakes"=tostring(timeDeltaS,"duration") | table place,
_time, "Time Between Quakes"
```

- This example searches for earthquakes in Alaska.

The `delta` command is used to calculate the difference in the timestamps, `_time`, between each earthquake and the one immediately before it. By default the difference is placed in a new field called **delta(_time)**. The time is in seconds.

- The `rename` command is used to change the default field name to **timeDeltaS**.
- An `eval` command is used with the `abs` function to convert the time into the absolute value of the time. This conversion is necessary because the differences between one earthquake and the earthquake immediately before it result in negative values.
- Another `eval` command is used with the `tostring` function to convert the time, in seconds, into a string value. The `duration` argument is part of the `tostring` function that specifies to convert the value to a readable time format `HH:MM:SS`.

The results look something like this:

place	_time	Time Between Quakes
32km N of Anchor Point, Alaska	2018-04-04 19:51:19.147	
6km NE of Healy, Alaska	2018-04-04 16:26:14.741	03:25:04.406
34km NE of Valdez, Alaska	2018-04-04 16:21:57.040	00:04:17.701
23km NE of Fairbanks, Alaska	2018-04-04 16:10:05.595	00:11:51.445
53km SSE of Cantwell, Alaska	2018-04-04 16:07:04.498	00:03:01.097
254km SE of Kodiak, Alaska	2018-04-04 13:57:06.180	02:09:58.318
114km NNE of Arctic Village, Alaska	2018-04-04 12:08:00.384	01:49:05.796
13km NNE of Larsen Bay, Alaska	2018-04-04 11:49:21.816	00:18:38.568
109km W of Cantwell, Alaska	2018-04-04 11:25:36.307	00:23:45.509
107km NW of Talkeetna, Alaska	2018-04-04 10:26:21.610	00:59:14.697

3. Calculate the difference in time between consecutive transactions

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.

Calculate the difference in time between consecutive transactions.

```
sourcetype=access_* | transaction JSESSIONID clientip startswith="view" endswith="purchase" | delta _time AS
timeDelta p=1 | eval timeDelta=abs(timeDelta) | eval timeDelta=tostring(timeDelta,"duration")
```

- This example groups events into transactions if they have the same values of `JSESSIONID` and `clientip`.
- The beginning of a transaction is defined by an event that contains the string **view**. The end of a transaction is defined by an event that contains the string **purchase**. The keywords **view** and **purchase** correspond to the values of the `action` field. You might also notice other values for the `action` field, such as **addtocart** and **remove**.

- The transactions are then piped into the `delta` command, which uses the `_time` field to calculate the time between one transaction and the transaction immediately preceding it. Specifically the difference between the timestamp for the last event in the transaction and the timestamp in the last event in the previous transaction.
- The search renames the time change as `timeDelta`.
- An `eval` command is used with the `abs` function to convert the time into the absolute value of the time. This conversion is necessary because the differences between one transaction and the previous transaction it result in negative values.
- Another `eval` command is used with the `tostring` function to convert the time, in seconds, into a string value. The **duration** argument is part of the `tostring` function that specifies to convert the value to a readable time format `HH:MM:SS`.

```

Time
Event
Last event from previous transaction
221.204.246.72 - - [17/Apr/2018:18:16:27] "POST /cart.do?action=purchase&itemId=EST-18&JSESSIONID=SD9SL7FF3ADFF53096 HTTP 1.1" 200 226 "http://www.buttercupgames.com/cart.do?action=addtocart&itemId=EST-18&categoryId=TEE&productId=MB-AG-T01" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.55.3 (KHTML, like Gecko) Version/5.1.5 Safari/534.55.3" 425
Collapse
duration = 3 host = www2 source = tutorialdata.zip:/www2/access.log ...
sourcetype = access_combined_wcookie timeDelta = 00:02:31

> 4/17/18
6:13:33.000 PM
Events in this transaction
91.205.189.15 - - [17/Apr/2018:18:13:33] "GET /cart.do?action=view&itemId=EST-26&productId=DB-SG-G01 &JSESSIONID=SD10SL4FF1ADFF53066 HTTP 1.1" 200 727 "http://www.buttercupgames.com/cart.do?action=view&itemId=EST-26&productId=DB-SG-G01" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 286
91.205.189.15 - - [17/Apr/2018:18:13:34] "POST /cart/success.do?JSESSIONID=SD10SL4FF1ADFF53066 HTTP 1.1" 200 3129 "http://www.buttercupgames.com/cart.do?action=purchase&itemId=EST-21" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 591
duration = 1 host = www1 source = tutorialdata.zip:/www1/access.log ...
sourcetype = access_combined_wcookie timeDelta = 00:02:51

```

timeDelta is the difference between the last timestamps in each transaction

See also

Commands

[accum](#)
[autoregress](#)
[streamstats](#)
[trendline](#)

diff

Description

Compares two search results and returns the line-by-line difference, or comparison, of the two. The two search results compared are specified by the two position values `position1` and `position2`. These values default to 1 and 2 to compare the first two results.

By default, the text (`_raw` field) of the two search results is compared. Other fields can be compared by selecting another field using `attribute`.

Syntax

```
diff [position1=int] [position2=int] [attribute=string] [diffheader=bool] [context=bool] [maxlen=int]
```

Optional arguments

position1

Datatype: <int>

Description: Of the table of input search results, selects a specific search result to compare to position2.

Default: position1=1 and refers to the first search result.

position2

Datatype: <int>

Description: Of the table of input search results, selects a specific search result to compare to position1. This value must be greater than *position1*.

Default: position2=2 and refers to the second search result.

attribute

Datatype: <field>

Description: The field name to be compared between the two search results.

Default: attribute=_raw, which refers to the text of the event or result.

diffheader

Datatype: <bool>

Description: If true, show the traditional diff header, naming the "files" compared. The diff header makes the output a valid diff as would be expected by the programmer command-line `patch` command.

Default: diffheader=false.

context

Datatype: <bool>

Description: If true, selects context-mode diff output as opposed to the default unified diff output.

Default: context=false, or unified.

maxlen

Datatype: <int>

Description: Controls the maximum content in bytes diffed from the two events. If maxlen=0, there is no limit.

Default: maxlen=100000, which is 100KB.

Examples

Example 1:

Compare the "ip" values of the first and third search results.

```
... | diff pos1=1 pos2=3 attribute=ip
```

Example 2:

Compare the 9th search results to the 10th.

```
... | diff position1=9 position2=10
```

See also

[set](#)

entitymerge

The `entitymerge` command is used with Splunk Enterprise Security.

For information about this command, see Overwrite asset or identity data with entitymerge in Splunk Enterprise Security in *Administer Splunk Enterprise Security*.

erex

Description

Use the `erex` command to extract data from a field when you do not know the regular expression to use. The command automatically extracts field values that are similar to the example values you specify.

The values extracted from the `fromfield` argument are saved to the `field`. The search also returns a regular expression that you can then use with the `rex` command to extract the field.

Syntax

The required syntax is in **bold**.

```
erex
[<field>]
examples=<string>
[counterexamples=<string>]
[fromfield=<field>]
[maxtrainers=<integer>]
```

Required arguments

examples

Syntax: `examples=<string>,<string>...`

Description: A comma-separated list of example values for the information to extract and save into a new field. Use quotation marks around the list if the list contains spaces. For example: "port 3351, port 3768".

field

Syntax: `<string>`

Description: A name for a new field that will take the values extracted from the `fromfield` argument. The resulting regular expression is generated and placed as a message under the Jobs menu in Splunk Web. That regular expression can then be used with the `rex` command for more efficient extraction.

Optional arguments

counterexamples

Syntax: `counterexamples=<string>,<string>,...`

Description: A comma-separated list of example values that represent information not to be extracted.

fromfield

Syntax: `fromfield=<field>`

Description: The name of the existing field to extract the information from and save into a new field.

Default: `_raw`

maxtrainers

Syntax: `maxtrainers=<int>`

Description: The maximum number values to learn from. Must be between 1 and 1000.

Default: 100

Usage

The values specified in the `examples` and `counterexample` arguments must exist in the events that are piped into the `erex` command. If the values do not exist, the command fails.

To make sure that the `erex` command works against your events, first run the search that returns the events you want without the `erex` command. Then copy the field values that you want to extract and use those for the `example` values with the Click the Job menu to see the generated regular expression based on your examples.

After you run a search or open a report in Splunk Web, the `erex` command returns informational log messages that are displayed in the search jobs manager window. However, these messages aren't displayed if the `infocsv_log_level` setting is set to `WARN` or `ERROR`. If you do not see the informational log messages when you click **Jobs** from the Activity menu, make sure that `infocsv_log_level` is set to the default, which is `INFO`.

Splunk Cloud Platform

To change the `infocsv_log_level` setting, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the the `infocsv_log_level` setting in the `limits.conf` file, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in How to edit a configuration file in the *Splunk Enterprise Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[search_info]` stanza, change the value for the `infocsv_log_level` setting.

View the regular expression

You can see the regular expression that is generated based on the `erex` command by clicking the **Job** menu in Splunk Web. See [Example 3](#).

The output of the `erex` command is captured in the `search.log` file. You can see the output by searching for "Successfully learned regex". The `search.log` file is located in the `$SPLUNK_HOME/var/run/splunk/dispatch/` directory. The search logs are not indexed by default. See Dispatch directory and search artifacts in the *Search Manual*.

Examples

1. Extract values based on an example

The following search extracts out month and day values like 7/01 and puts the values into the `monthday` attribute.

```
... | erex monthday examples="7/01"
```

2. Extract values based on examples and counter examples

The following search extracts out month and day values like 7/01 and 7/02, but not patterns like 99/2. The extracted values are put into the `monthday` attribute.

```
... | erex monthday examples="7/01, 07/02" counterexamples="99/2"
```

3. Extract values based on examples and return the most common values

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Determine which are the most common ports used by potential attackers.

1. Run a search to find examples of the port values, where there was a failed login attempt.

```
sourcetype=secure* port "failed password"
```

Time	Event	SELECTED FIELDS	INTERESTING FIELDS
May 01, 2018 12:15:05 AM	Thu May 01 2018 00:15:05 mailsrv sshd[5276]: Failed password for invalid user appserver from 194.8.74.23 port 3351 ssh2	host=mailsvr source=tutorialdata.zip:/mailsvr/secure.log sourcetype=secure	#date_hour 1 #date_mday 8 #date_minute 1 #date_month 2 #date_second 4 #date_wday 7 #date_year 1 #date_zone 1
May 01, 2018 12:15:05 AM	Thu May 01 2018 00:15:05 mailsrv sshd[1839]: Failed password for root from 194.8.74.23 port 3768 ssh2	host=mailsvr source=tutorialdata.zip:/mailsvr/secure.log sourcetype=secure	#date_hour 1 #date_mday 8 #date_minute 1 #date_month 2 #date_second 4 #date_wday 7 #date_year 1 #date_zone 1
May 01, 2018 12:15:05 AM	Thu May 01 2018 00:15:05 mailsrv sshd[5258]: Failed password for invalid user testuser from 194.8.74.23 port 3626 ssh2	host=mailsvr source=tutorialdata.zip:/mailsvr/secure.log sourcetype=secure	#date_hour 1 #date_mday 8 #date_minute 1 #date_month 2 #date_second 4 #date_wday 7 #date_year 1 #date_zone 1
May 01, 2018 12:15:05 AM	Thu May 01 2018 00:15:05 mailsrv sshd[1165]: Failed password for apache from 194.8.74.23 port 4684 ssh2	host=mailsvr source=tutorialdata.zip:/mailsvr/secure.log sourcetype=secure	#date_hour 1 #date_mday 8 #date_minute 1 #date_month 2 #date_second 4 #date_wday 7 #date_year 1 #date_zone 1

2. Then use the `erex` command to extract the port field. You must specify several examples with the `erex` command. Use the `top` command to return the most common port values. By default the `top` command returns the top 10 values.

```
sourcetype=secure* port "failed password" | erex port examples="port 3351, port 3768" | top port
```

This search returns a table with the count of top ports that match the search.

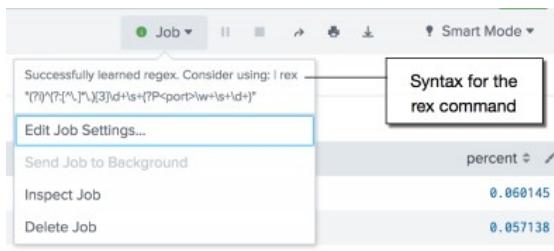
The results appear on the Statistics tab and look something like this:

port	count	percent
port 2444	20	0.060145
port 3281	19	0.057138
port 2842	19	0.057138

port	count	percent
port 2760	19	0.057138
port 1174	19	0.057138
port 4955	18	0.054130
port 1613	18	0.054130
port 1059	18	0.054130
port 4542	17	0.051123
port 4519	17	0.051123

3. Click the **Job** menu to see the generated regular expression based on your examples. You can use the `rex` command with the regular expression instead of using the `erex` command. The regular expression for this search example is

`| rex (?i)^(?:[^.]*\.){3}\d+\s+(?P<port>\w+\s+\d+)` for this search example.



You can replace the `erex` command with the `rex` command and generated regular expression in your search. For example:

```
sourcetype=secure* port "failed password" | rex (?i)^(?:[^.]*\.){3}\d+\s+(?P<port>\w+\s+\d+) | top port
```

Using the `rex` command with a regular expression is more cost effective than using the `erex` command.

See also

Commands

- [extract](#)
- [kvform](#)
- [multikv](#)
- [regex](#)
- [rex](#)
- [xmlkv](#)

eval

Description

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

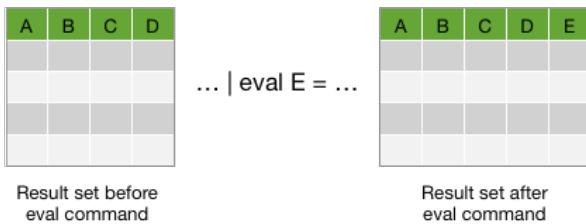
- If the field name that you specify does not match a field in the output, a new field is added to the search results.
- If the field name that you specify matches a field name that already exists in the search results, the results of the `eval` expression overwrite the values in that field.

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.



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 eval expression is case-sensitive. 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.

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`).

Numeric calculations

During calculations, numbers are treated as double-precision floating-point numbers, subject to all the usual behaviors of floating point numbers. If the calculation results in the floating-point special value NaN(Not a Number), it is represented as "nan" in your results. The special values for positive and negative infinity are represented in your results as "inf" and "-inf" respectively. Division by zero results in a null field.

Rounding

Results are rounded to a precision appropriate to the precision of the input results. The precision of the results can be no greater than the precision of the least-precise input. For example, the following search has different precision for `0.2` in each of the calculations based on the number of zeros following the number 2:

```
|makeresults | eval decimal1=8.250 * 0.2, decimal2=8.250 * 0.20, decimal3=8.250 * 0.200, exact=8.250 * exact(0.2)
```

The results look like this:

<code>_time</code>	<code>decimal1</code>	<code>decimal2</code>	<code>decimal3</code>	<code>exact</code>
2022-09-02 21:53:30	2	1.7	1.65	1.650

If you want to return an arbitrary number of digits of precision, use the `exact` function, as shown in the last calculation in the search. See the [exact evaluation function](#).

Long numbers

There are situations where the results of a calculation contain more digits than can be represented by a floating-point number. In those situations precision might be lost on the least significant digits. The limit to precision is 17 significant digits, or $-2^{53} +1$ to $2^{53} -1$.

Significant digits

If a result returns a long number with more digits than you want to use, you can specify the number of digits to return using the `sigfig` function. See **Example 2** under the [basic examples](#) for the `sigfig(x)` function.

Supported functions

You can use a wide range of functions with the `eval` command. For general information about using functions, see [Evaluation functions](#).

- For a list of functions by category, see [Function list by category](#).
- For an alphabetical list of functions, see [Alphabetical list of functions](#).

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.

Type	Operators
Arithmetic	+ - * / %
Concatenation	.
Boolean	AND OR NOT XOR < > <= >= != = == LIKE

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.

The = and == operators

In expressions, the single equal sign (`=`) and the double equal sign (`==`) are synonymous. Although you can use these operators interchangeably in your searches to make assignments or comparisons, keep the following conventions in mind when you create your searches:

- The `=` operator means either "is equal to" or "is assigned to", depending on the context. This operator is typically used to assign a value to a field.
- The `==` operator means "is equal to". This operator is typically used to compare 2 values.

For example, in the following search, `=` is used to assign the `description` to the `case` expression, and `==` is used to indicate `status` is equal to specific error codes.

```
| eval description=case(status==200, "OK", status==404, "Not found", status==500, "Internal Server Error")
```

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'`.

When assigning the value of a field to the value of another field, do not use leading spaces in the field name. However, you can use spaces at the end of the field name. For example, a search like this that assigns the same value to fields called `first` and `second` produces valid results, even though `first` has a trailing space:

```
| makeresults | eval "first" = 123 | eval second='first '
```

However, the following search does not produce valid results because `first` has a leading space:

```
| makeresults | eval " first" = 123 | eval second=' first'
```

Dynamic field name creation

You can use the value of one field as the name of another field by using curly braces (`{ }`), which dynamically creates a field name on the left side of an eval expression. For example, if you have an event with the `aName=counter` and `aValue=1234` fields, use `| eval {aName}=aValue` to return `counter=1234`.

Searches that dynamically create field names generate errors and do not complete if there are unbounded recursive replacements. For example, the following search doesn't produce results because the right side of the eval expression generates bracketed field names that are recursive. In this case, `{p}` replaces the value of `p` in the fieldname, so `v_{p}` becomes `v_{v_{p}}` over and over again, in a recursive loop.

```
| makeresults | eval p="{p}", v_{p} = p
```

The following search produces valid results because the value of one field is not recursively used as the name of another field.

```
| makeresults | eval field1="counter", {field1}="1234"
```

The search results look something like this:

_time	counter	field1
2023-05-10 21:15:49	1234	counter

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 Platform, 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:

```
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:

```
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.

```
... | 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 1000000.1278, the following search returns `x` as 1,000,000.13. The `tostring` function returns only two decimal places with the decimals rounded up or down depending on the values.

```
... | 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 \$1,000,000.13

10. Rounding with values outside of the range of supported values

The range of values supported in Splunk searches is 0 to $2^{53} - 1$. This example demonstrates the differences in results you get when you use values in eval expressions that fall outside of the range of supported values.

```
| makeresults | eval max_supported_val = pow(2, 53)-1, val1 = max_supported_val + 1, val2 = max_supported_val + 2
```

The results look something like this:

_time	max_supported_val	val1	val2
2022-09-04 10:22:11	9007199254740991	9007199254740992	9007199254740992

As you can see, because `val1` and `val2` are beyond the range of supported values, the results are the same even though they should be different. This is because of rounding on those values that are outside of the supported range of values.

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-NUL 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.

```
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:

Description	count	min(Mag)	max(Mag)
Deep	35	4.1	6.7
Mid	635	0.8	6.3
Shallow	6236	-0.60	7.70

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

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 **All time** 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:

Time	Event	INTERESTING FIELDS	SELECTED FIELDS
3/15/18 6:22:16.000 PM	91.205.189.15 - - [15/Mar/2018:18:22:16] "GET /oldlink?itemId=EST-14&JSESSIONID=SD6SL7FF7ADFF53113 HTTP 1.1" 200 1665 "http://www.buttercupgames.com/oldlink?itemId=EST-14" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 159 host = www2 network = other source = tutorialdata.zip://www2/access.log sourcetype = access_combined_wcookie		
3/15/18 6:22:15.000 PM	91.205.189.15 - - [15/Mar/2018:18:22:15] "GET /category.screen?categoryId=SHOOTER&JSESSIONID=SD6SL7FF7ADFF53113 HTTP 1.1" 200 1369 "http://www.google.com" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 779 host = www2 network = other source = tutorialdata.zip://www2/access.log sourcetype = access_combined_wcookie		
3/15/18 6:20:56.000 PM	182.236.164.11 - - [15/Mar/2018:18:20:56] "GET /cart.do?action=addtocart&itemId=EST-15&productId=85 -AG-098JSESSIONID=SD6SL8FF10ADFF53101 HTTP 1.1" 200 2252 "http://www.buttercupgames.com/oldlink?itemId=EST-15" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 586 host = www1 network = local source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie		
3/15/18 6:20:55.000 PM	182.236.164.11 - - [15/Mar/2018:18:20:55] "POST /oldlink?itemId=EST-18&JSESSIONID=SD6SL8FF10ADFF53101 HTTP 1.1" 408 893 "http://www.buttercupgames.com/product.screen?productId=SF-BVS-Q01" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 134 host = www1 network = local source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie		

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.

```
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:

mailfrom	from_user	from_domain
na.lui@sample.net	na.lui	sample.net
MAILER-DAEMON@hcp2mailsec.sample.net	MAILER-DAEMON	hcp2mailsec.sample.net
M&MService@example.com	M&MService	example.com
AlexMartin@oursample.de	AlexMartin	oursample.de
Exit_Desk@sample.net	Exit_Desk	sample.net
buttercup-forum+SEMAC8PUC4RETTUB@groups.com	buttercup-forum+SEMAC8PUC4RETTUB	groups.com
eduardo.rodriguez@sample.net	eduardo.rodriguez	sample.net
VC00110489@techexamples.com	VC00110489	techexamples.com

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 field, `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.

```
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:

location	count
abroad	3543
local	14136

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 **All time** 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:

List ▾			Format	20 Per Page ▾	< Prev	1	2	3	4	5	6	7	8	...	Next >
< Hide Fields	All Fields	i	Time	Event											
SELECTED FIELDS															
a duration	29														
a durationstr	29														
a host	3														
a source	3														
a sourcetype	1														
INTERESTING FIELDS															
a action	5														
a bytes	100+														
a categoryid	8														
a clientip	100+														
# closed_txn	2														
# date_hour	19														
# date_mday	1														
# date_minute	60														
# date_month	1														
# date_second	60														
# date_wday	1														
# date_year	1														
a date_zone	1														

See also

Functions

[Evaluation functions](#)

Commands

[where](#)

eventcount

Description

Returns the number of events in the specified indexes.

Syntax

The required syntax is in **bold**.

```
| eventcount
[index=<string>]...
[summarize=<bool>]
[report_size=<bool>]
[list_vix=<bool>]
```

Required arguments

None.

Optional arguments

index

Syntax: index=<string>

Description: A name of the index report on, or a wildcard matching many indexes to report on. You can specify this argument multiple times, for example `index=*` `index=_*`.

Default: If no index is specified, the command returns information about the default index.

list_vix

Syntax: list_vix=<bool>

Description: Specify whether or not to list virtual indexes. If list_vix=false, the command does not list virtual indexes.

Default: true

report_size

Syntax: report_size=<bool>

Description: Specify whether or not to report the index size. If report_size=true, the command returns the index size in bytes.

Default: false

summarize

Syntax: summarize=<bool>

Description: Specifies whether or not to summarize events across all peers and indexes. If summarize=false, the command splits the event counts by index and search peer.

Default: true

Usage

The `eventcount` command is a report-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Specifying a time range has no effect on the results returned by the `eventcount` command. All of the events on the indexes you specify are counted.

Specifying indexes

You cannot specify indexes to exclude from the results. For example, `index!=foo` is not valid syntax.

You can specify the `index` argument multiple times. For example:

```
|eventcount summarize=false index=_audit index=main
```

Running in clustered environments

Do not use the `eventcount` command to count events for comparison in indexer clustered environments. When a search runs, the `eventcount` command checks all **buckets**, including replicated and primary buckets, across all indexers in a cluster. As a result, the search may return inaccurate event counts.

Examples

Example 1:

Display a count of the events in the default indexes from all of the search peers. A single count is returned.

```
| eventcount
```

Example 2:

Return the number of events in only the internal default indexes. Include the index size, in bytes, in the results.

```
| eventcount summarize=false index=_* report_size=true
```

The results appear on the Statistics tab and should be similar to the results shown in the following table.

count	index	server	size_bytes
52550	_audit	buttercup-mbpr15.sv.splunk.com	7217152
1423010	_internal	buttercup-mbpr15.sv.splunk.com	122138624
22626	_introspection	buttercup-mbpr15.sv.splunk.com	98619392
10	_telemetry	buttercup-mbpr15.sv.splunk.com	135168
0	_thefishbucket	buttercup-mbpr15.sv.splunk.com	0

When you specify `summarize=false`, the command returns three fields: `count`, `index`, and `server`. When you specify `report_size=true`, the command returns the `size_bytes` field. The values in the `size_bytes` field are not the same as the index size on disk.

Example 3:

Return the event count for each index and server pair. Only the external indexes are returned.

```
| eventcount summarize=false index=*
```

20 Per Page ▾ Format ▾ Preview ▾

count	index	server
0	history	sting-mba13.sv.splunk.com
109864	main	sting-mba13.sv.splunk.com
0	summary	sting-mba13.sv.splunk.com
6906	usgs_earthquake	sting-mba13.sv.splunk.com

To return the count all of the indexes including the internal indexes, you must specify the internal indexes separately from the external indexes:

```
| eventcount summarize=false index=*_ index=_*
```

See also

[metadata](#), [fieldsummary](#)

eventstats

Description

Generates summary statistics from fields in your events and saves those statistics in a new field.

Only those events that have fields pertinent to the aggregation are used in generating the summary statistics. The generated summary statistics can be used for calculations in subsequent commands in your search. See [Usage](#).

Syntax

The required syntax is in **bold**.

```
eventstats
[allnum=<bool>]
<stats-agg-term> ...
[<by-clause>]
```

Required arguments

<stats-agg-term>

Syntax: <stats-func>(<evalued-field> | <wc-field>) [AS <wc-field>]

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names.

Optional arguments

allnum

Syntax: allnum=<bool>

Description: If set to true, computes numerical statistics on each field, if and only if ,all of the values of that field are numerical. If you have a BY clause, the allnum argument applies to each group independently.

Default: false

<by-clause>

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

Stats function options

stats-func

Syntax: The syntax depends on the function that you use. Refer to the table below.

Description: Statistical and charting functions that you can use with the `eventstats` command. Each time you invoke the `eventstats` command, you can use one or more functions. However, you can only use one BY clause. See [Usage](#).

The following table lists the supported functions by type of function. Use the links in the table to see descriptions and examples for each function. For an overview about using functions with commands, see [Statistical and charting functions](#).

Type of function	Supported functions and syntax
------------------	--------------------------------

Type of function	Supported functions and syntax			
Aggregate functions	avg() count() distinct_count() estdc() estdc_error()	exactperc<int>() max() median() min() mode()	perc<int>() range() stdev() stdevp()	sum() sumsq() upperperc<int>() var() varp()
Event order functions	earliest()	first()	last()	latest()
Multivalue stats and chart functions	list(X)	values(X)		

Usage

The `eventstats` command is a dataset processing command. See [Command types](#).

The `eventstats` search processor uses a `limits.conf` file setting named `max_mem_usage_mb` to limit how much memory the `eventstats` command can use to keep track of information. When the limit is reached, the `eventstats` command processor stops adding the requested fields to the search results.

Do not set `max_mem_usage_mb=0` as this removes the bounds to the amount of memory the `eventstats` command processor can use. This can lead to search failures.

Splunk Cloud Platform

To change the `max_mem_usage_mb` setting, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `max_mem_usage_mb` setting, follow these steps.

Prerequisites

- ◊ Have the permissions to change the `max_mem_usage_mb` setting. Only users with file system access, such as system administrators, can increase the `max_mem_usage_mb` setting using configuration files.
- ◊ Know how to edit configuration files. Review the steps in [How to edit a configuration file](#) in the *Splunk Enterprise Admin Manual*.
- ◊ Decide which directory to store configuration file changes in. There can be configuration files with the same name in your default, local, and app directories. See [Where you can place \(or find\) your modified configuration files](#) in the *Splunk Enterprise Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[default]` stanza, look for the `max_mem_usage_mb` setting.
3. Under **Note**, read the information about the `eventstats` command and how the `max_mem_usage_mb` and the `maxresultrows` settings are used to determine the maximum number of results to return.

4. Change the value for the `max_mem_usage_mb` setting and if necessary the `maxresultrows` setting.

Differences between `eventstats` and `stats`

The `eventstats` command is similar to the `stats` command. You can use both commands to generate aggregations like average, sum, and maximum.

The differences between these commands are described in the following table:

stats command	eventstats command
Events are transformed into a table of aggregated search results	Aggregations are placed into a new field that is added to each of the events in your output
You can only use the fields in your aggregated results in subsequent commands in the search	You can use the fields in your events in subsequent commands in your search, because the events have not been transformed

How `eventstats` generates aggregations

The `eventstats` command looks for events that contain the field that you want to use to generate the aggregation. The command creates a new field in every event and places the aggregation in that field. The aggregation is added to every event, even events that were not used to generate the aggregation.

For example, you have 5 events and 3 of the events have the field you want to aggregate on. The `eventstats` command generates the aggregation based on the data in the 3 events. A new field is added to every event and the aggregation is added to that field in every event.

Statistical functions that are not applied to specific fields

With the exception of the `count` function, when you pair the `eventstats` command with functions that are not applied to specific fields or `eval` expressions that resolve into fields, the search head processes it as if it were applied to a wildcard for all fields. In other words, when you have `| eventstats avg` in a search, it returns results for `| eventstats avg(*)`.

This "implicit wildcard" syntax is officially deprecated, however. Make the wildcard explicit. Write `| eventstats <function>(*)` when you want a function to apply to all possible fields.

Functions and memory usage

Some functions are inherently more expensive, from a memory standpoint, than other functions. For example, the `distinct_count` function requires far more memory than the `count` function. The `values` and `list` functions also can consume a lot of memory.

If you are using the `distinct_count` function without a split-by field or with a low-cardinality split-by by field, consider replacing the `distinct_count` function with the `estdc` function (estimated distinct count). The `estdc` function might result in significantly lower memory usage and run times.

Event order functions

Using the `first` and `last` functions when searching based on time does not produce accurate results.

- To locate the first value based on time order, use the `earliest` function, instead of the `first` function.
- To locate the last value based on time order, use the `latest` function, instead of the `last` function.

For example, consider the following search.

```
index=test sourcetype=testDb | eventstats first(LastPass) as LastPass, last(_time) as mostRecentTestTime BY testCaseId | where startTime==LastPass OR _time==mostRecentTestTime | stats first(startTime) AS startTime, first(status) AS status, first(histID) AS currentHistId, last(histID) AS lastPassHistId BY testCaseId
```

When you use the `stats` and `eventstats` commands for ordering events based on time, use the `earliest` and `latest` functions.

The following search is the same as the previous search except the `first` and `last` functions are replaced with the `earliest` and `latest` functions.

```
index=test sourcetype=testDb | eventstats latest(LastPass) AS LastPass, earliest(_time) AS mostRecentTestTime BY testCaseId | where startTime==LastPass OR _time==mostRecentTestTime | stats latest(startTime) AS startTime, latest(status) AS status, latest(histID) AS currentHistId, earliest(histID) AS lastPassHistId BY testCaseId
```

Basic examples

1. Calculate the overall average duration

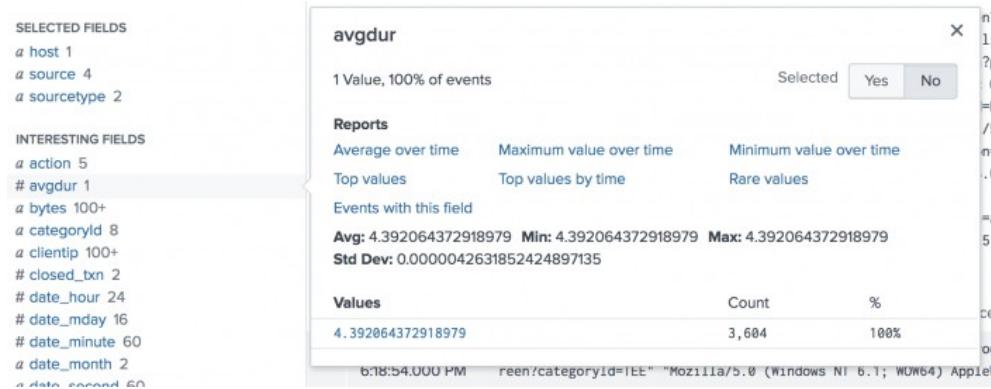
This example uses the sample data from the Search Tutorial but should work with any format of Apache web access log. To try this example on your own Splunk instance, you must download the sample data and follow the instructions to [get the tutorial data into Splunk](#). Use the time range **All time**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

Calculate the overall average duration of a set of transactions, and place the calculation in a new field called `avgdur`.

```
host=www1 | transaction clientip host maxspan=30s maxpause=5s | eventstats avg(duration) AS avgdur
```

Because no `BY` clause is specified, a single aggregation is generated and added to every event in a new field called `avgdur`.

When you look at the list of Interesting Fields, you will see that `avgdur` has only one value.

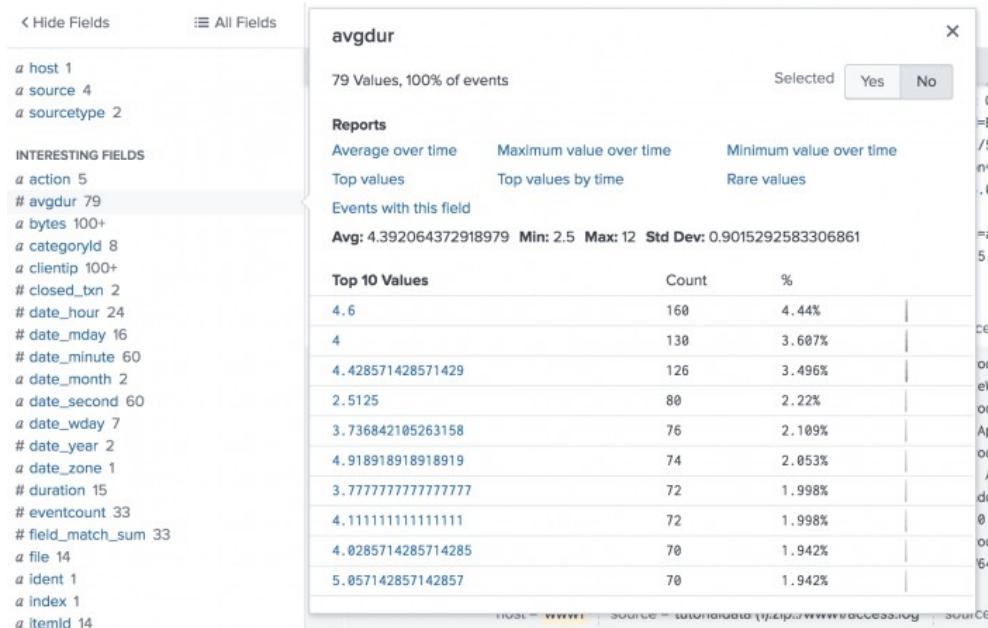


2. Calculate the average duration grouped by a specific field

This example is the same as the previous example except that an average is calculated for each distinct value of the `date_minute` field. The new field `avgdur` is added to each event with the average value based on its particular value of `date_minute`.

```
host=www1 | transaction clientip host maxspan=30s maxpause=5s | eventstats avg(duration) AS avgdur BY date_minute
```

When you look at the list of Interesting Fields, you will see that `avgdur` has 79 values, based on the timestamp, duration, and `date_minute` values.



3. Search for spikes in the volume of errors

This searches for spikes in error volume. You can use this search to trigger an alert if the count of errors is higher than average, for example.

```
eventtype="error" | eventstats avg(bytes) AS avg | where bytes>avg
```

Extended example

The following example provides you with a better understanding of how the `eventstats` command works. This example is actually a progressive set of small examples, where one example builds on or extends the previous example.

It's much easier to see what the `eventstats` command does by showing you examples, using a set of simple events.

These examples use the `makerevents` command to create a set of events. The `streamstats` and `eval` commands are used to create additional fields in the events.

Creating a set of events

Let's start by creating a set of four events. One of the events contains a null value in the `age` field.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle")
```

- The `streamstats` command is used to create the `count` field. The `streamstats` command calculates a cumulative count for each event, at the time the event is processed.
- The `eval` command is used to create two new fields, `age` and `city`. The `eval` command uses the value in the `count` field.
- The `case` function takes pairs of arguments, such as `count=1, 25`. The first argument is a Boolean expression. When that expression is TRUE, the corresponding second argument is returned.

The results of the search look like this:

<code>_time</code>	<code>age</code>	<code>city</code>	<code>count</code>
2020-02-05 18:32:07	25	San Francisco	1
2020-02-05 18:32:07	39	Seattle	2
2020-02-05 18:32:07	31	San Francisco	3
2020-02-05 18:32:07		Seattle	4

Using eventstats with a BY clause

The `BY` clause in the `eventstats` command is optional, but is used frequently with this command. The `BY` clause groups the generated statistics by the values in a field. You can use any of the statistical functions with the `eventstats` command to generate the statistics. See the [Statistical and charting functions](#).

In this example, the `eventstats` command generates the average age for each city. The generated averages are placed into a new field called `avg(age)`.

The following search is the same as the previous search, with the `eventstats` command added at the end:

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle") | eventstats avg(age) BY city
```

- For San Francisco, the average age is $28 = (25 + 31) / 2$.
- For Seattle, there is only one event with a value. The average is $39 = 39 / 1$. The `eventstats` command places that average in every event for Seattle, including events that did not contain a value for `age`.

The results of the search look like this:

<code>_time</code>	<code>age</code>	<code>avg(age)</code>	<code>city</code>	<code>count</code>
2020-02-05 18:32:07	25	28	San Francisco	1
2020-02-05 18:32:07	39	39	Seattle	2
2020-02-05 18:32:07	31	28	San Francisco	3

_time	age	avg(age)	city	count
2020-02-05 18:32:07		39	Seattle	4

Renaming the new field

By default, the name of the new field that is generated is the name of the statistical calculation. In these examples, that name is `avg(age)`. You can rename the new field using the `AS` keyword.

In the following search, the `eventstats` command has been adjusted to rename the new field to `average age by city`.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle") | eventstats avg(age) AS "average age by city" BY city
```

The results of the search look like this:

_time	age	average age by city	city	count
2020-02-05 18:32:07	25	28	San Francisco	1
2020-02-05 18:32:07	39	39	Seattle	2
2020-02-05 18:32:07	31	28	San Francisco	3
2020-02-05 18:32:07		39	Seattle	4

Events with text values

The previous examples show how an event is processed that does not contain a value in the `age` field. Let's see how events are processed that contain an alphabetic character value in the field that you want to use to generate statistics .

The following search includes the word `test` as a value in the `age` field.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, "test") | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle")
```

The results of the search look like this:

_time	age	city	count
2020-02-05 18:32:07	25	San Francisco	1
2020-02-05 18:32:07	39	Seattle	2
2020-02-05 18:32:07	31	San Francisco	3
2020-02-05 18:32:07	test	Seattle	4

Let's add the `eventstats` command to the search.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, "test") | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle") | eventstats avg(age) BY city
```

The alphabetic values are treated like null values. The results of the search look like this:

_time	age	avg(age)	city	count
2020-02-05 18:32:07	25	28	San Francisco	1
2020-02-05 18:32:07	39	39	Seattle	2
2020-02-05 18:32:07	31	28	San Francisco	3
2020-02-05 18:32:07	test	39	Seattle	4

Using the allnum argument

But suppose you don't want statistics generated when there are alphabetic characters in the field or the field is empty?

The `allnum` argument controls how the `eventstats` command processes field values. The default setting for the `allnum` argument is FALSE. Which means that the field used to generate the statistics does not need to contain all numeric values. Fields with empty values or alphabetic character values are ignored. You've seen this in the earlier examples.

You can force the `eventstats` command to generate statistics only when the fields contain all numeric values. To accomplish this, you can set the `allnum` argument to TRUE.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, "test") | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle") | eventstats allnum=true avg(age) BY city
```

The results of the search look like this:

_time	age	avg(age)	city	count
2020-02-05 18:32:07	25	28	San Francisco	1
2020-02-05 18:32:07	39		Seattle	2
2020-02-05 18:32:07	31	28	San Francisco	3
2020-02-05 18:32:07	test		Seattle	4

Because the `age` field contains values for Seattle that are not all numbers, the entire set of values for Seattle are ignored. No average is calculated.

The `allnum=true` argument applies to empty values as well as alphabetic character values.

See also

Commands

[stats](#)
[streamstats](#)

Blogs

[Search commands > stats, eventstats and streamstats](#)

extract

Description

Extracts field-value pairs from the search results. The `extract` command works only on the `_raw` field. If you want to extract from another field, you must perform some field renaming before you run the `extract` command.

Syntax

The required syntax is in **bold**.

```
extract
[<extract-options>... ]
[<extractor-name>...]
```

Required arguments

None.

Optional arguments

<extract-options>

Syntax: `clean_keys=<bool> | kvdelim=<string> | limit=<int> | maxchars=<int> | mv_add=<bool> | pairdelim=<string> | reload=<bool> | segment=<bool>`

Description: Options for defining the extraction. See the [Extract_options](#) section in this topic.

<extractor-name>

Syntax: `<string>`

Description: A stanza in the `transforms.conf` file. This is used when the `props.conf` file does not explicitly cause an extraction for this source, sourcetype, or host.

Extract options

`clean_keys`

Syntax: `clean_keys=<bool>`

Description: Specifies whether to clean keys. Overrides `CLEAN_KEYS` in the `transforms.conf` file.

Default: The value specified in the `CLEAN_KEYS` in the `transforms.conf` file.

`kvdelim`

Syntax: `kvdelim=<string>`

Description: A list of character delimiters that separate the key from the value. If the delimiter appears in the value, that value is not extracted. For example, if the delimiter is a colon (:) and a key-value pair is `Referer: https://buttercupgames.com`, the key-value pair is not extracted.

`limit`

Syntax: `limit=<int>`

Description: Specifies how many automatic key-value pairs to extract.

Default: 50

`maxchars`

Syntax: `maxchars=<int>`

Description: Specifies how many characters to look into the event.

Default: 10240

mv_add

Syntax: mv_add=<bool>

Description: Specifies whether to create multivalued fields. Overrides the value for the MV_ADD parameter in the `transforms.conf` file.

Default: false

pairdelim

Syntax: pairdelim=<string>

Description: A list of character delimiters that separate the key-value pairs from each other.

reload

Syntax: reload=<bool>

Description: Specifies whether to force reloading of the `props.conf` and `transforms.conf` files.

Default: false

segment

Syntax: segment=<bool>

Description: Specifies whether to note the locations of the key-value pairs with the results.

Default: false

Usage

The `extract` command is a **distributable streaming command**. See [Command types](#).

Alias

The alias for the `extract` command is `kv`.

Examples

1. Specify the delimiters to use for the field and value extractions

Extract field-value pairs that are delimited by the pipe (|) or semicolon (;) characters. Extract values of the fields that are delimited by the equal (=) or colon (:) characters. The delimiters are individual characters. In this example the "=" or ":" character is used to delimit the key value. Similarly, a "|" or ";" is used to delimit the field-value pair itself.

```
... | extract pairdelim="|;", kvdelim="=:"
```

2. Extract field-value pairs and reload the field extraction settings

Extract field-value pairs and reload field extraction settings from disk.

```
... | extract reload=true
```

3. Rename a field to _raw to extract from that field

Rename the `_raw` field to a temporary name. Rename the field you want to extract from, to `_raw`. In this example the field name is `uri_query`.

```
... | rename _raw AS temp uri_query AS _raw | extract pairdelim="?&" kvdelim="=" | rename _raw AS uri_query  
temp AS _raw
```

4. Extract field-value pairs from a stanza in the transforms.conf file

Extract field-value pairs that are defined in the `my-access-extractions` stanza in the `transforms.conf` file.

```
... | extract access-extractions
```

The `transforms.conf` stanza for this example looks something like this.

```
[my-access-extractions]  
REGEX=\[ (?! (?:(headerName|headerValue)) ([^\s\=]+)\=([^\\]+) )  
FORMAT=$1:$2
```

See also

[kvform](#), [multikv](#), [rex](#), [spath](#), [xmlkv](#), [xpath](#)

fieldformat

Description

With the `fieldformat` command you can use an `<eval-expression>` to change the format of a field value when the results render. This command changes the appearance of the results without changing the underlying value of the field.

Because commands that come later in the search pipeline cannot modify the formatted results, use the `fieldformat` command as late in the search pipeline as possible.

The `fieldformat` command does not apply to commands that export data, such as the `outputcsv` and `outputlookup` commands. The export retains the original data format and not the rendered format. If you want the format to apply to exported data, use the `eval` command instead of the `fieldformat` command.

Syntax

```
fieldformat <field>=<eval-expression>
```

Required arguments

`<field>`

Description: The name of a new or existing field, non-wildcarded, for the output of the eval expression.

`<eval-expression>`

Syntax: `<string>`

Description: A combination of values, variables, operators, and functions that represent the value of your destination field. You can specify only one `<eval-expression>` with the `fieldformat` command. To specify multiple formats you must use multiple `fieldformat` commands. See [Examples](#).

For more information, see the [eval command](#).

For information about supported functions, see [Usage](#).

Usage

The `fieldformat` command is a **distributable streaming command**. See [Command types](#).

Time format variables are frequently used with the `fieldformat` command. See [Date and time format variables](#).

Functions

You can use a wide range of functions with the `fieldformat` 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.

Type of function	Supported functions and syntax		
Comparison and Conditional functions	<code>case (X, "Y", ...)</code> <code>cidrmatch ("X", Y)</code> <code>coalesce (X, ...)</code> <code>false ()</code> <code>if (X, Y, Z)</code>	<code>in (VALUE-LIST)</code> <code>like (TEXT, PATTERN)</code> <code>match (SUBJECT, "REGEX")</code> <code>null ()</code>	<code>nullif (X, Y)</code> <code>searchmatch (X)</code> <code>true ()</code> <code>validate (X, Y, ...)</code>
Conversion functions	<code>printf ("format", arguments)</code>	<code>tonumber (NUMSTR, BASE)</code>	<code>tostring (X, Y)</code>
Cryptographic functions	<code>md5 (X)</code> <code>sha1 (X)</code>	<code>sha256 (X)</code>	<code>sha512 (X)</code>
Date and Time functions	<code>now ()</code> <code>relative_time (X, Y)</code>	<code>strftime (X, Y)</code> <code>strptime (X, Y)</code>	<code>time ()</code>
Informational functions	<code>isbool (X)</code> <code>isint (X)</code> <code>isnotnull (X)</code>	<code>isnull (X)</code> <code>isnum (X)</code>	<code>isstr (X)</code> <code>typeof (X)</code>
Mathematical functions	<code>abs (X)</code> <code>ceiling (X)</code> <code>exact (X)</code> <code>exp (X)</code>	<code>floor (X)</code> <code>ln (X)</code> <code>log (X, Y)</code> <code>pi ()</code>	<code>pow (X, Y)</code> <code>round (X, Y)</code> <code>sigfig (X)</code> <code>sqrt (X)</code>
Multivalue eval functions	<code>commands (X)</code> <code>mvappend (X, ...)</code> <code>mvcount (MVFIELD)</code> <code>mvdedup (X)</code>	<code>mvfilter (X)</code> <code>mvfind (MVFIELD, "REGEX")</code> <code>mvindex (MVFIELD, STARTINDEX, ENDINDEX)</code> <code>mvjoin (MVFIELD, STR)</code>	<code>mvrange (X, Y, Z)</code> <code>mvsort (X)</code> <code>mvzip (X, Y, "Z")</code>
Statistical eval functions	<code>max (X, ...)</code>	<code>min (X, ...)</code>	<code>random ()</code>
Text functions	<code>len (X)</code>	<code>rtrim (X, Y)</code>	<code>trim (X, Y)</code>

Type of function	Supported functions and syntax		
	lower(X) ltrim(X, Y) replace(X, Y, Z)	spath(X, Y) split(X, "Y") substr(X, Y, Z)	upper(X) urldecode(X)
Trigonometry and Hyperbolic functions	acos(X) acosh(X) asin(X) asinh(X) atan(X)	atan2(X, Y) atanh(X) cos(X) cosh(X) hypot(X, Y)	sin(X) sinh(X) tan(X) tanh(X)

Basic examples

1. Format numeric values to display commas

This example uses the `metadata` command to return results for the sourcetypes in the **main** index.

```
| metadata type=sourcetypes | table sourcetype totalCount
```

The `metadata` command returns many fields. The `table` command is used to return only the sourcetype and totalCount fields.

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

sourcetype	totalCount
access_combined_wcookie	39532
cisco:esa	112421
csv	9510
secure	40088
vendor_sales	30244

Use the `fieldformat` command to reformat the appearance of the field values. The values in the totalCount field are formatted to display the values with commas.

```
| metadata type=sourcetypes | table sourcetype totalCount | fieldformat totalCount=toString(totalCount, "commas")
```

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

sourcetype	totalCount
access_combined_wcookie	39,532
cisco:esa	112,421
csv	9,510
secure	40,088

sourcetype	totalCount
vendor_sales	30,244

2. Display UNIX time in a readable format

Assume that the `start_time` field contains UNIX time. Format the `start_time` field to display only the hours, minutes, and seconds that correspond to the UNIX time.

```
... | fieldformat start_time = strftime(start_time, "%H:%M:%S")
```

3. Add currency symbols to numerical values

To format numerical values in a field with a currency symbol, you must specify the symbol as a literal and enclose it in quotation marks. Use a period character as a binary concatenation operator, followed by the `tostring` function, which enables you to display commas in the currency values.

```
...| fieldformat totalSales="$".tostring(totalSales, "commas")
```

Extended example

1. Formatting multiple fields

This example shows how to change the appearance of search results to display commas in numerical values and dates into readable formats.

First, use the `metadata` command to return results for the sourcetypes in the **main** index.

```
|metadata type=sourcetypes | table sourcetype totalCount |fieldformat totalCount=tostring(totalCount, "commas")
```

```
| metadata type=sourcetypes | rename totalCount as Count firstTime as "First Event" lastTime as "Last Event" recentTime as "Last Update" | table sourcetype Count "First Event" "Last Event" "Last Update"
```

- The `metadata` command returns the fields `firstTime`, `lastTime`, `recentTime`, `totalCount`, and `type`.
- In addition, because the search specifies `types=sourcetypes`, a field called `sourcetype` is also returned.
- The `totalCount`, `firstTime`, `lastTime`, and `recentTime` fields are renamed to `Count`, `First Event`, `Last Event`, and `Last Update`.
- The `First Event`, `Last Event`, and `Last Update` fields display the values in UNIX time.

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

sourcetype	Count	First Event	Last Event	Last Update
access_combined_wcookie	39532	1520904136	1524014536	1524067875
cisco:esa	112421	1521501480	1521515900	1523471156
csv	9510	1520307602	1523296313	1523392090
secure	40088	1520838901	1523949306	1524067876
vendor_sales	30244	1520904187	1524014642	1524067875

Use the `fieldformat` command to reformat the appearance of the output of these fields. The `Count` field is formatted to display the values with commas. The `First Event`, `Last Event`, and `Last Update` fields are formatted to display the values

in readable timestamps.

```
| metadata type=sourcetypes | rename totalCount as Count firstTime as "First Event" lastTime as "Last Event" recentTime as "Last Update" | table sourcetype Count "First Event" "Last Event" "Last Update" | fieldformat Count=tostring(Count, "commas") | fieldformat "First Event"=strftime('First Event', "%c") | fieldformat "Last Event"=strftime('Last Event', "%c") | fieldformat "Last Update"=strftime('Last Update', "%c")
```

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

sourcetype	Count	First Event	Last Event	Last Update
access_combined_wcookie	39,532	Mon Mar 12 18:22:16 2018	Tue Apr 17 18:22:16 2018	Wed Apr 18 09:11:15 2018
cisco:esa	112,421	Mon Mar 19 16:18:00 2018	Mon Mar 19 20:18:20 2018	Wed Apr 11 11:25:56 2018
csv	9,510	Mon Mar 5 19:40:02 2018	Mon Apr 9 10:51:53 2018	Tue Apr 10 13:28:10 2018
secure	40,088	Mon Mar 12 00:15:01 2018	Tue Apr 17 00:15:06 2018	Wed Apr 18 09:11:16 2018
vendor_sales	30,244	Mon Mar 12 18:23:07 2018	Tue Apr 17 18:24:02 2018	Wed Apr 18 09:11:15 2018

See also

[eval](#), [where](#)

[Date and time format variables](#)

fields

Description

Keeps or removes fields from search results based on the field list criteria.

By default, the internal fields `_raw` and `_time` are included in output in Splunk Web. Additional internal fields are included in the output with the `outputcsv` command. See [Usage](#).

Syntax

`fields [+|-] <wc-field-list>`

Required arguments

`<wc-field-list>`

Syntax: `<field>, <field>, ...`

Description: Comma-delimited list of fields to keep or remove. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Optional arguments

`+ | -`

Syntax: `+ | -`

Description: If the plus (+) symbol is specified, only the fields in the `wc-field-list` are kept in the results. If the negative (-) symbol is specified, the fields in the `wc-field-list` are removed from the results.

Default: +

Usage

The `fields` command is a distributable streaming command. See [Command types](#).

Internal fields and Splunk Web

The leading underscore is reserved for names of internal fields such as `_raw` and `_time`. By default, the internal fields `_raw` and `_time` are included in the search results in Splunk Web. The `fields` command does not remove these internal fields unless you explicitly specify that the fields should not appear in the output in Splunk Web.

For example, to remove all internal fields, you specify:

```
... | fields - _*
```

To exclude a specific field, such as `_raw`, you specify:

```
... | fields - _raw
```

Be cautious removing the `_time` field. Statistical commands, such as `timechart` and `chart`, cannot display date or time information without the `_time` field.

Displaying internal fields in Splunk Web

Other than the `_raw` and `_time` fields, internal fields do not display in Splunk Web, even if you explicitly specify the fields in the search. For example, the following search does not show the `_bkt` field in the results.

```
index=_internal | head 5 | fields + _bkt | table _bkt
```

To display an internal field in the results, the field must be copied or renamed to a field name that does not include the leading underscore character. For example:

```
index=_internal | head 5 | fields + _bkt | eval bkt=_bkt | table bkt
```

Internal fields and the outputcsv command

When the `outputcsv` command is used in the search, there are additional internal fields that are automatically added to the CSV file. The most common internal fields that are added are:

- `_raw`
- `_time`
- `_indextime`

To exclude internal fields from the output, specify each field that you want to exclude. For example:

```
... | fields - _raw _indextime _sourcetype _serial | outputcsv MyTestCsvFile
```

You cannot match wildcard characters in searches that use the fields command

You can use the asterisk (*) in your searches as a wildcard character, but you can't use a backslash (\) to escape an asterisk in search strings. A backslash\ and an asterisk * match the characters * in searches, not an escaped wildcard * character. Because Splunk platform doesn't support escaping wildcards, asterisk (*) characters in field names can't be matched in searches that keep or remove fields from search results.

Support for backslash characters (\) in the fields command

To match a backslash character (\) in a field name when using the `fields` command, use 2 backslashes for each backslash. For example, to display fields that contain `http:\\\\`, use the following command in your search:

```
... | fields http:\\\\\\*
```

See Backslashes in the *Search Manual*.

Examples

Example 1:

Remove the `host` and `ip` fields from the results

```
... | fields - host, ip
```

Example 2:

Keep only the `host` and `ip` fields. Remove all of the internal fields. The internal fields begin with an underscore character, for example `_time`.

```
... | fields host, ip | fields - _*
```

Example 3:

Remove unwanted internal fields from the output CSV file. The fields to exclude are `_raw_indextime`, `_sourcetype`, `_subsecond`, and `_serial`.

```
index=_internal sourcetype="splunkd" | head 5 | fields - _raw, _indextime, _sourcetype, _subsecond, _serial  
| outputcsv MyTestCsvfile
```

Example 4:

Keep only the fields `source`, `sourcetype`, `host`, and all fields beginning with `error`.

```
... | fields source, sourcetype, host, error*
```

See also

[rename](#), [table](#)

fieldsummary

Description

The `fieldsummary` command calculates summary statistics for all fields or a subset of the fields in your events. The summary information is displayed as a results table.

Syntax

```
fieldsummary [maxvals=<unsigned_int>] [<wc-field-list>]
```

Optional arguments

maxvals

Syntax: `maxvals=<unsigned_int>`

Description: Specifies the maximum distinct values to return for each field. Cannot be negative. Set `maxvals = 0` to return all available distinct values for each field.

Default: 100

wc-field-list

Syntax: `<field> ...`

Description: A single field name or a space-delimited list of field names. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Usage

The `fieldsummary` command is a dataset processing command. See [Command types](#).

The `fieldsummary` command displays the summary information in a results table. The following information appears in the results table:

Summary field name	Description
field	The field name in the event.
count	The number of events/results with that field.
distinct_count	The number of unique values in the field.
is_exact	Whether or not the field is exact. This is related to the distinct count of the field values. If the number of values of the field exceeds <code>maxvals</code> , then <code>fieldsummary</code> will stop retaining all the values and compute an approximate distinct count instead of an exact one. 1 means it is exact, 0 means it is not.
max	If the field is numeric, the maximum of its value.
mean	If the field is numeric, the mean of its values.
min	If the field is numeric, the minimum of its values.
numeric_count	The count of numeric values in the field. This would not include NULL values.
stdev	If the field is numeric, the standard deviation of its values.
values	

Summary field name	Description
	The distinct values of the field and count of each value. The values are sorted first by highest count and then by distinct value, in ascending order.

Examples

1. Return summaries for all fields

This example returns summaries for all fields in the `_internal` index from the last 15 minutes.

```
index=_internal earliest=-15m latest=now | fieldsummary
```

In this example, the results in the `max`, `min`, and `stdev` fields are formatted to display up to 4 decimal points.

2. Return summaries for specific fields

This example returns summaries for fields in the `_internal` index with names that contain "size" and "count". The search returns only the top 10 values for each field from the last 15 minutes.

```
index=_internal earliest=-15m latest=now | fieldsummary maxvals=10 *size* *count*
```

Events		Patterns		Statistics (18)		Visualization											
20 Per Page ▾		Format		Preview ▾													
field ▾	✓	count ▾	✓	distinct_count ▾	✓	is_exact ▾	✓	max ▾ ✓	mean ▾ ✓	✓	min ▾	✓	numeric_count ▾	✓	stdev ▾ ✓	✓	values ▾ ✓
count	✓	43	✓	7	✓	1	✓	1,000.0000	353.0465	✓	-1	✓	43	✓	479.1867	✓	[{"value": "1000", "count": 15}, {"value": "20", "count": 9}, {"value": "-1", "count": 8}, {"value": "0", "count": 7}, {"value": "2", "count": 2}, {"value": "1", "count": 1}, {"value": "4", "count": 1}]
current_queue_size	✓	87	✓	1	✓	1	✓	0.0000	0.0000	✓	0	✓	87	✓	0.0000	✓	[{"value": "0", "count": 87}]
current_size	✓	551	✓	2	✓	1	✓	99.0000	5.2105	✓	0	✓	551	✓	22.1265	✓	[{"value": "0", "count": 522}, {"value": "99", "count": 29}]
current_size_kb	✓	464	✓	1	✓	1	✓	0.0000	0.0000	✓	0	✓	464	✓	0.0000	✓	[{"value": "0", "count": 464}]

See also

[analyzefields](#), [anomalies](#), [anomalousvalue](#), [stats](#)

filldown

Description

Replaces null values with the last non-null value for a field or set of fields. If no list of fields is given, the `filldown` command will be applied to all fields. If there are not any previous values for a field, it is left blank (NULL).

Syntax

```
filldown <wc-field-list>
```

Required arguments

<wc-field-list>

Syntax: <field> ...

Description: A space-delimited list of field names. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Examples

Example 1:

Filtdown null values values for all fields.

```
... | filldown
```

Example 2:

Filtdown null values for the count field only.

```
... | filldown count
```

Example 3:

Filtdown null values for the count field and any field that starts with 'score'.

```
... | filldown count score*
```

See also

[fillnull](#)

fillnull

Description

Replaces null values with a specified value. Null values are field values that are missing in a particular result but present in another result. Use the `fillnull` command to replace null field values with a string. You can replace the null values in one or more fields. You can specify a string to fill the null field values or use the default, field value which is zero (0).

Syntax

The required syntax is in **bold**.

```
fillnull
[value=<string>]
[<field-list>]
```

Required arguments

None.

Optional arguments

field-list

Syntax: <field>...

Description: A space-delimited list of one or more fields. If you specify a field list, all of the fields in that list are filled in with the `value` you specify. If you specify a field that didn't previously exist, the field is created. If you do not specify a field list, the `value` is applied to all fields.

value

Syntax: `value=<string>`

Description: Specify a string value to replace null values. If you do not specify a value, the default value is applied to the <field-list>.

Default: 0

Usage

The `fillnull` command is a distributable streaming command when a `field-list` is specified. When no `field-list` is specified, the `fillnull` command fits into the dataset processing type. See [Command types](#).

Fields in the event set should have at least one non-null value

Due to the unique behavior of the `fillnull` command, Splunk software isn't able to distinguish between a null field value and a null field that doesn't exist in the Splunk schema. In order for a field to exist in the schema, it must have at least one non-null value in the event set. To ensure downstream processing of fields by the `fillnull` command, ensure that there is at least one non-null value for the fields in the event set.

For example, consider the following search:

```
| makeresults | eval test="123123", test2=null() | fillnull value=NULL
```

The results look something like this:

_time	test
2023-06-07 17:49:45	123123

Notice that the test2 field doesn't show up in the results, even though the `eval` command created it. The reason the test2 field isn't in the results is that there isn't at least one non-null value for the field in the event set.

If a field doesn't have at least one non-null value in the event set, it's considered a nonexistent field, so downstream commands like the `fillnull` command can't process it. For example, consider the following search:

```
| makeresults | eval test="123123" | eval test2=null() | table test test2 | fillnull value=NULL
```

The results look something like this:

test	test2
123123	

The search results display the test2 field, but not the intended NULL value. This is because the upstream `eval` command initially set test2 to `null`, so the field doesn't exist in the schema.

Now consider the following search:

```
| makeresults | eval test1=split("123,456", ",") | mvexpand test1 | eval test2;if(test1=="123", null(), "abc") | fillnull value=NULL
```

The results look something like this:

_time	test1	test2
2023-06-07 18:22:24	123	NULL
2023-06-07 18:22:24	456	abc

This search generates at least one non-null value for each field and shows the expected behavior by setting the null value of the test2 field to the NULL string. Now all the values display as expected because the test2 field has at least one non-null value.

Examples

1. Fill all empty field values with the default value

Your search has produced the following search results:

_time	ACCESSORIES	ARCADE	SHOOTER	SIMULATION	SPORTS	STRATEGY	TEE
2021-03-17	5	17	6	3	5	32	
2021-03-16		63	39	30	22	127	56
2021-03-15	65	94	38	42		128	60

You can fill all of empty field values with the zero by adding the `fillnull` command to your search.

```
... | fillnull
```

The search results will look like this:

_time	ACCESSORIES	ARCADE	SHOOTER	SIMULATION	SPORTS	STRATEGY	TEE
2021-03-17	5	17	6	3	5	32	0
2021-03-16	0	63	39	30	22	127	56
2021-03-15	65	94	38	42	0	128	60

2. Fill all empty fields with the string "NULL"

For the current search results, fill all empty field values with the string "NULL".

```
... | fillnull value=NULL
```

3. Fill the specified fields with the string "unknown"

Suppose that your search has produced the following search results:

_time	host	average_kbps	instanenous_kbps	kbps
2021/02/14 12:00	danube.sample.com		1.865	3.420
2021/02/14 11:53	mekong.buttercupgames.com	0.710	0.164	1.256
2021/02/14 11:47	danube.sample.com	1.325		2.230
2021/02/14 11:42	yangtze.buttercupgames.com	2.249	0.000	2.249
2021/02/14 11:39		2.874	3.841	1.906
2021/02/14 11:33	nile.example.net	2.023	0.915	

You can fill all empty field values in the "host" and "kbps" fields with the string "unknown" by adding the `fillnull` command to your search.

```
... | fillnull value=unknown host kbps
```

The results look like this:

_time	host	average_kbps	instanenous_kbps	kbps
2021/02/14 12:00	danube.sample.com		1.865	3.420
2021/02/14 11:53	mekong.buttercupgames.com	0.710	0.164	1.256
2021/02/14 11:47	danube.sample.com	1.325		2.230
2021/02/14 11:42	yangtze.buttercupgames.com	2.249	0.000	2.249
2021/02/14 11:39	unknown	2.874	3.841	1.906
2021/02/14 11:33	nile.example.net	2.023	0.915	unknown

If you specify a field that does not exist the field is created and the value you specify is added to the new field. For example if you specify `bytes` in the field list, the `bytes` field is created and filled with the string "unknown".

```
... | fillnull value=unknown host kbps bytes
```

The results look like this:

_time	host	average_kbps	instantaneous_kbps	kbytes	bytes
2021/02/14 12:00	danube.sample.com		1.865	3.420	unknown
2021/02/14 11:53	mekong.buttercupgames.com	0.710	0.164	1.256	unknown
2021/02/14 11:47	danube.sample.com	1.325		2.230	unknown
2021/02/14 11:42	yangtze.buttercupgames.com	2.249	0.000	2.249	unknown
2021/02/14 11:39	unknown	2.874	3.841	1.906	unknown
2021/02/14 11:33	nile.example.net	2.023	0.915	unknown	unknown

4. Use the fillnull command with the timechart command

Build a time series chart of web events by host and fill all empty fields with the string "NULL".

```
sourcetype="web" | timechart count by host | fillnull value=NULL
```

See also

Related commands

[filldown](#)
[streamstats](#)

findtypes

Description

Generates suggested event types by taking the results of a search and producing a list of potential event types. At most, 5000 events are analyzed for discovering event types.

Syntax

```
findtypes max=<int> [notcovered] [useraw]
```

Required arguments

max

Datatype: <int>

Description: The maximum number of events to return.

Default: 10

Optional arguments

notcovered

Description: If this keyword is used, the `findtypes` command returns only event types that are not already covered.

useraw

Description: If this keyword is used, the `findtypes` command uses phrases in the `_raw` text of events to generate event types.

Examples

Example 1:

Discover 10 common event types.

```
... | findtypes
```

Example 2:

Discover 50 common event types and add support for looking at text phrases.

```
... | findtypes max=50 useraw
```

See also

[typer](#)

folderize

This feature is deprecated.

The `folderize` command is deprecated in the Splunk platform as of version 9.0.0. It might be removed in a future version. See the [Release Notes](#).

Description

Creates a higher-level grouping, such as replacing filenames with directories. Replaces the `attr` attribute value with a more generic value, which is the result of grouping the `attr` value with other values from other results, where grouping occurs by tokenizing the `attr` value on the `sep` separator value.

For example, the `folderize` command can group search results, such as those used on the Splunk Web home page, to list hierarchical buckets (e.g. directories or categories). Rather than listing 200 sources, the `folderize` command breaks the source strings by a separator (e.g. `/`) and determines if looking only at directories results in the number of results requested.

Syntax

```
folderize attr=<string> [sep=<string>] [size=<string>] [minfolders=<int>] [maxfolders=<int>]
```

Arguments

`attr`

Syntax: `attr=<string>`

Description: Replaces the `attr` attribute value with a more generic value, which is the result of grouping it with other values from other results, where grouping occurs by tokenizing the attribute (`attr`) value on the separator (`sep`) value.

sep

Syntax: sep=<string>

Description: Specify a separator character used to construct output field names when multiple data series are used in conjunction with a split-by field.

Default: ::

size

Syntax: size=<string>

Description: Supply a name to be used for the size of the folder.

Default: totalCount

minfolders

Syntax: minfolders=<int>

Description: Set the minimum number of folders to group.

Default: 2

maxfolders

Syntax: maxfolders=<int>

Description: Set the maximum number of folders to group.

Description:

Examples

1. Group results into folders based on URI

Consider this search.

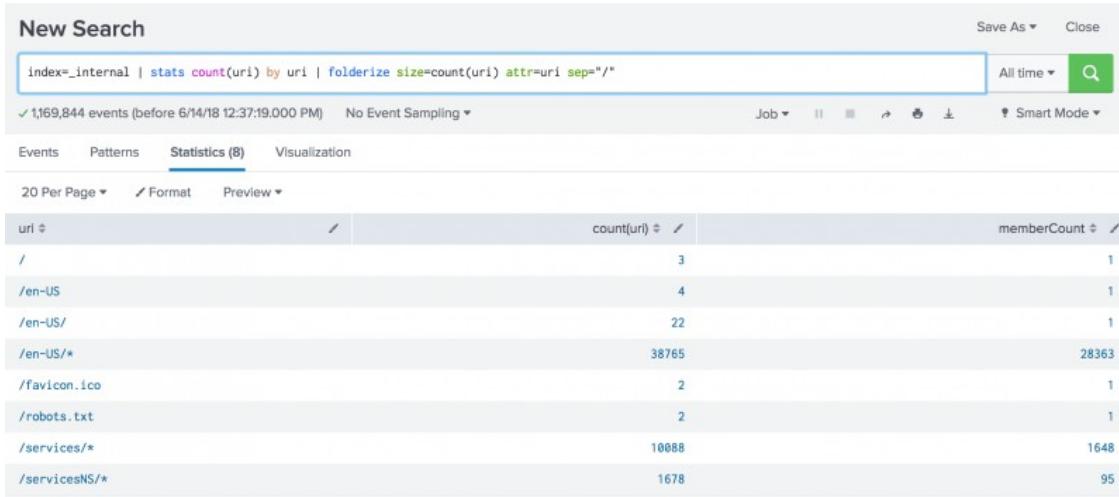
```
index=_internal | stats count(uri) by uri
```

The following image shows the results of the search run using the **All Time** time range. Many of the results start with /en-US/account. Because some of the URLs are very long, the image does not show the second column on the far right. That column is the `count(uri)` column created by the `stats` command.

Using the `folderize` command, you can summarize the URI values into more manageable groupings.

```
index=_internal | stats count(uri) by uri | folderize size=count(uri) attr=uri sep="/"
```

The following image shows the URIs grouped in the result set.



In this example, the `count(uri)` column is the count of the unique URIs that were returned from the `stats` command. The `memberCount` column shows the count of the URIs in each group. For example, the `/en-US/` URI was found 22 times in the events, as shown in the `count(uri)` column. When the `folderize` command arranges the URI into groups, there is only 1 member in the `/en-US/` group. Whereas the URIs that start with `/services/` occurred 10088 times in the events, but there are only 1648 unique members in the `/services/*` group.

foreach

Description

Use this command to run a subsearch that includes a template to iterate over the following elements:

- Each field in a wildcard field list
- Each value in a single **multivalue field**
- A single field representing a JSON array

Syntax

The required syntax is in **bold**.

```
foreach
mode=(multifield | multivalue | json_array)
<wildcard-field-list> | <field>
[<mode-options>]
<subsearch>
[<subsearch-options>]
```

Required arguments

wildcard-field-list

Syntax: <string> ...

Description: A space-delimited wildcard field name or list of wildcard field names that is used to iterate over one or more fields in a search. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard field such as value*. You can also specify a list of wildcard fields, such as hostA* hostB* hostC*.

You can use this argument only with the `multifield` mode.

field

Syntax: <string>

Description: A field name that is used when iterating over elements in a multivalue field or JSON array, using the `multivalue` mode or `json_array` mode in a search.

subsearch

Syntax: [subsearch]

Description: A subsearch that includes a template for replacing the values of the specified fields, which depend on whether you are using `multifield` mode, `multivalue` mode, or `json_array` mode. For each field that is matched, the templated subsearch replaces values as follows:

Option	Default template value	Replacement	Mode
fieldstr	<<FIELD>>	The whole field name.	<code>multifield</code>
matchstr	<<MATCHSTR>>	The part of the field name that matches the wildcard values in the wildcard field.	<code>multifield</code>
matchseg1	<<MATCHSEG1>>	The part of the field name that matches the first wildcard.	<code>multifield</code>
matchseg2	<<MATCHSEG2>>	The part of the field name that matches the second wildcard.	<code>multifield</code>
matchseg3	<<MATCHSEG3>>	The part of the field name that matches the third wildcard.	<code>multifield</code>
itemstr	<<ITEM>>	Matches each element in a multivalue field or JSON array.	<code>multivalue</code> or <code>json_array</code>

Optional arguments

mode

Syntax: mode=<mode-name>

Description: Tells the `foreach` command to iterate over multiple fields, a multivalue field, or a JSON array. If a mode is not specified, the `foreach` command defaults to the mode for multiple fields, which is the `multifield` mode.

You can specify one of the following modes for the `foreach` command:

Argument	Syntax	Description
multifield	mode=multifield	Iterates over a single supplied field name or a list of multiple field names, which can include wildcard characters. This is the default if the <code>mode</code> is not specified.
multivalue	mode=multivalue	Iterates over a single supplied multivalue field. Use this mode with multivalue fields.

Argument	Syntax	Description
json_array	mode=json_array	Iterates over a single supplied JSON array. Use this mode with JSON functions .

mode options

The mode options depend on the mode you use.

Use the following options to iterate over a field or list of fields using the `multifield` mode. These options are available only with the `multifield` mode.

Option	Syntax	Description
fieldstr	fieldstr=<string>	A customizable string that replaces the <code><<FIELD>></code> template value with the name that you specify. The value of the <code>fieldstr</code> option must match the <code><<FIELD>></code> template value. For example, if you change the default <code><<FIELD>></code> template value to <code>MYFIELD</code> , then you must also change the value of <code>fieldstr</code> option to <code>MYFIELD</code> .
matchstr	matchstr=<string>	<p>A customizable string that replaces the <code><<MATCHSTR>></code> template value with the segment of the field name that matches the wildcard(s) in each field in the list. The value of the <code>matchstr</code> option must match the <code><<MATCHSTR>></code> template value. For example, if you change the default <code><<MATCHSTR>></code> template value to <code>ID</code>, then you must also change the value of <code>matchstr</code> to <code>ID</code>.</p> <p>To avoid unpredictable results in searches, do not use the <code>matchstr</code> option with any <code>matchseg*</code> options.</p>
matchseg1	matchseg1=<string>	<p>A customizable string that replaces the <code><<MATCHSEG1>></code> template value with the segment of the field name that matches the first wildcard in each field in the list. The value of the <code>matchseg1</code> option must match the <code><<MATCHSEG1>></code> template value. For example, if you change the default <code><<MATCHSEG1>></code> template value to <code>PHONE</code>, then you must also change the value of <code>matchseg1</code> to <code>PHONE</code>.</p> <p>To avoid unpredictable results in searches, do not use the <code>matchseg1</code> option with the <code>matchstr</code> option.</p>
matchseg2	matchseg2=<string>	<p>A customizable string that replaces the <code><<MATCHSEG2>></code> template value with the segment of the field name that matches the second wildcard in each field in the list. The value of the <code><<matchseg2>></code> option must match the <code><<MATCHSEG2>></code> template value. For example, if you change the default <code><<MATCHSEG2>></code> template value to <code>PRICE</code>, then you must also change the value of <code><<matchseg2>></code> to <code>PRICE</code>.</p> <p>To avoid unpredictable results in searches, do not use the <code>matchseg2</code> option together with the <code>matchstr</code> option.</p>
matchseg3	matchseg3=<string>	<p>A customizable string that replaces the <code><<MATCHSEG3>></code> template value with the segment of the field name that matches the third wildcard in each field in the list. The value of the <code><<matchseg3>></code> option must match the <code><<MATCHSEG3>></code> template value. For example, if you change the default <code><<MATCHSEG3>></code> template value to <code>ADDRESS</code>, then you must also change the value of <code><<matchseg3>></code> to <code>ADDRESS</code>.</p> <p>To avoid unpredictable results in searches, do not use the <code>matchseg3</code> option together with the <code>matchstr</code> option.</p>

Use the following option to iterate over multivalue fields or JSON arrays using the the `multivalue` mode or the `json_array` mode. This option is available only with the `multivalue` mode or the `json_array` mode.

Option	Syntax	Description
itemstr	itemstr=<string>	<p>Replaces the <code><<ITEM>></code> template value with each element in a multivalue field or JSON array. The value of the <code>itemstr</code> option must match the <code><<ITEM>></code> template value. For example, if you change the default <code><<ITEM>></code> template value to <code><<ITERATOR>></code>, then you must also change the value of <code><<itemstr>></code> to <code><<ITERATOR>></code>.</p> <p>For example, consider the following search:</p> <pre> foreach mode=multivalue numbers [eval result = result + <<ITEM>>]</pre> <p>That search is equivalent to this search:</p> <pre> foreach mode=multivalue itemstr=<<ITERATOR>> numbers [eval result = result + <<ITERATOR>>]</pre>

subsearch options

Syntax: <option-name>

Description:

The subsearch options depend on the mode you use.

You can use the following optional template values in a subsearch when iterating over one or more fields. These template values are available only with the `multifield` mode.

Template value	Description
<code><<FIELD>></code>	A customizable string replacement for each field name in the field list. Each time you run a subsearch, this value is used to replace the whole field name in the <code>fieldstr</code> option for each field you specify. For example, if you change the value of <code><<FIELD>></code> to <code>MYFIELD</code> , then the value of the <code>fieldstr</code> option is also <code>MYFIELD</code> . Your search might look like this: ... foreach test* fieldstr=MYFIELD [eval total=total + MYFIELD].
<code><<MATCHSTR>></code>	A customizable string replacement that represents wildcards in each matching field name in the list. For example, if the wildcard field that is being matched is <code>test*</code> and the field name is <code>test8</code> , then the value of <code><<MATCHSTR>></code> is 8.
<code><<MATCHSEG1>></code>	To avoid unpredictable results in searches, do not use the <code><<MATCHSTR>></code> template value together with any <code><<MATCHSEG*>></code> template values.
<code><<MATCHSEG2>></code>	A customizable string replacement for the segment of the field name that matches the first segment before the first wildcard in each matching field name in the list.
	To avoid unpredictable results in searches, do not use the <code><<MATCHSEG1>></code> template value with the <code><<MATCHSTR>></code> template value.
	A customizable string replacement for the segment of the field name that matches the second segment before the second wildcard in each matching field name in the list.
	To avoid unpredictable results in searches, do not use the <code><<MATCHSEG2>></code> template value with

Template value	Description
	the <<MATCHSTR>> template value.
<<MATCHSEG3>>	A customizable string replacement for the segment of the field name that matches the third segment before the third wildcard in each matching field name in the list. To avoid unpredictable results in searches, do not use the <<MATCHSEG3>> template value with the <<MATCHSTR>> template value.

You can use the following optional template value in a subsearch to iterate over elements in a multivalue field or JSON array. This template value is available only with the `multivalue` mode or the `json_array` mode.

Template value	Description
<<ITEM>>	A customizable string that replaces the <<ITEM>> template value with some other string that is substituted with the contents of the multivalued field or JSON array being iterated over. Only a single <code>eval</code> statement is permitted in the search pipeline when using this template value with the <code>multivalue</code> mode or <code>json_array</code> mode.

Usage

The `foreach` command is a streaming command.

You can use the `foreach` command in the following ways:

- To obtain results across multiple fields in each result row. This is useful, for example, when you need to calculate the average or sum of a value across multiple columns in each row. If you want to iterate over one or more matching fields, use the `multifield` mode.
- To iterate over multiple values within a single row's field in multivalue fields or JSON arrays. This is useful, for example, when you need to concatenate strings or calculate the average or sum of a set of numbers in a single field across multiple columns in each row in a multivalue field or JSON array. If you want to iterate over a multivalue field, use the `multivalue` mode. If you want to iterate over a JSON array node, use the `json_array` mode.

The default <<ITEM>> template value should be used only when the mode is `multivalue` or `json_array`.

Iterating over multiple matching fields containing nonalphanumeric characters

If the field names contain characters other than alphanumeric characters, such as dashes, underscores, or periods, enclose the <<FIELD>> template value in single quotation marks in the right side of the `eval` command portion of the search to avoid unpredictable results. For example, the following search uses the default `foreach` multifield mode and adds the values from all of the fields that match `myfield_*`.

```
| makeresults | eval myfield_1 = 5, myfield_2 = 10 | foreach myfield_* [eval <<FIELD>> = '<<FIELD>>' + <<MATCHSTR>>]
```

The search results look something like this:

_time	myfield_1	myfield_2

2023-3-14 15:55:50	6	12
--------------------	---	----

The `<<FIELD>>` template value in the `foreach` subsearch is just a string replacement of the field named `myfield_*`. The eval expression does not recognize field names with nonalphanumeric characters unless the field names are surrounded by single quotation marks. For the eval expression to work, the `<<FIELD>>` template value must be surrounded by single quotation marks.

Support for multiple eval statements

If you need to include multiple `eval` statements with the `foreach` command, use the default multifield mode. Multiple `eval` statements are not supported in `foreach` searches that use multivalue mode or JSON array mode. As a result, your searches on multivalue fields or JSON arrays must contain only a single `eval` statement in the pipeline. However, your `eval` statement can include as many assignments as you want.

For example, the following multivalue search with multiple `eval` assignments completes successfully because there is only one `eval` statement, which means there aren't any piped commands following the `eval` command.

```
| makeresults | eval mv=mvappend("5", "15"), total = 0, count = 0 | foreach mode=multivalue mv [eval total = total + <<ITEM>>, count = count + 1]
```

The search results look something like this.

_time	count	mv	total
2022-03-29 19:52:38	2	5 15	20

Wildcards are not supported in multivalue fields or JSON arrays

Unlike multifield mode, the modes for multivalue fields and JSON arrays don't support wildcards in search expressions. Instead, these modes treat a wildcard as part of the field name. For example, the following search includes a field called `mv*`, which looks like a wildcarded field.

```
| makeresults | eval mv1=mvappend("1", "2"), mv2=mvappend("3", "4"), mv*=mvappend("100", "300"), total = 0 | foreach mode=multivalue mv* [eval total = total + <<ITEM>>]
```

However, multivalue and JSON array modes don't recognize the wildcard and add up all of the fields containing `mv`. As a result, the search results for multivalue mode look something like this:

_time	mv*	mv1	mv2	total
2022-4-20 15:55:50	100 300	1 2	3 4	400

Elements of the same type in multivalue fields or JSON arrays

Elements in a subsearch and the `eval` expression must be of the same type as either strings or numbers. For example, the following search correctly adds up the three numbers in the JSON array because all of the elements are numbers.

```
| foreach json_array(1, 2, 3) [eval total = total + <<ITEM>>]
```

However, the following search results in an error because adding a number to a string isn't allowed.

```
| foreach json_array(1, 2, "hello") [eval total = total + <<ITEM>>]
```

Examples

1. Generate a total for each row in search results

Suppose you have events that contain the following data:

categoryId	www1	www2
ACCESSORIES	1000	500
SIMULATION	3000	750
ARCADE	800	
STRATEGY	400	200

Use the `foreach` command with the default `multifield` mode to iterate over each field that starts with `www` and generate a total for each row in the search results.

```
... | eval total=0 | foreach www* [eval total=total + <<FIELD>>]
```

The results look like this:

categoryId	www1	www2	total
ACCESSORIES	1000	500	1500
SIMULATION	3000	750	3750
ARCADE	800	800	
STRATEGY	400	200	600

2. Add the values from all fields that start with similar names

The following search adds the values from all of the fields that start with similar names and match the wildcard field `test*`. It uses the `foreach` command with the default `multifield` mode

```
| makeresults | eval total=0, test1=1, test2=2, test3=3 | foreach test* [eval total=total + <<FIELD>>]
```

The results of the search look something like this.

_time	test1	test2	test3	total
2022-4-20 15:55:50	1	2	3	6

- This search creates one result using the `makeresults` command.
- The search then uses the `eval` command to create the fields `total`, `test1`, `test2`, and `test3` with corresponding values.
- The `foreach` command is used to perform the subsearch for every field that starts with "test". Each time the subsearch is run, the previous total is added to the value of the `test` field to calculate the new total. The final total after all of the `test` fields are processed is 6.

The following table shows how the subsearch iterates over each `test` field. The table shows the beginning value of the `total` field each time the subsearch is run and the calculated total based on the value for the `test` field.

Subsearch iteration	test field	total field start value	test field value	calculation of total field
1	test1	0	1	0+1=1
2	test2	1	2	1+2=3
3	test3	3	3	3+3=6

3. Iterate over fields using the eval and foreach commands

The `eval` command and `foreach` command can be used in similar ways. For example, this search uses the `eval` command:

```
| makeresults | eval name="name" | eval price="price" | eval category="category"
```

It is equivalent to this search that uses the `foreach` command with the default `multifield` mode:

```
| makeresults | foreach name price category [eval <> = "<>"]
```

The results of both searches look something like this:

_time	category	name	price
2022-4-20 15:55:50	category	name	price

4. Monitor license usage

Use the `foreach` command to monitor license usage.

First run the following search on the license master to return the daily license usage per source type in bytes:

```
index=_internal source=*license_usage.log type!="*Summary" earliest=-30d | timechart span=1d sum(b) AS daily_bytes by st
```

The search results for one user across several days looks something like this:

_time	csv	universal_data_json
2022-04-03	8308923	36069628
2022-04-04	7290647	48851560
2022-04-05	7676935	12542231
2022-04-06	3016517	17521059

You can also use the `foreach` command with the default `multifield` mode to calculate the daily license usage in gigabytes for each field:

```
index=_internal source=*license_usage.log type!="*Summary" earliest=-30d | timechart span=1d sum(b) AS daily_bytes by st | foreach * [eval <>='<>/1024/1024']
```

This time the search results look something like this:

_time	csv	universal_data_json
2022-04-03	0.2335968237849277853	0.6309081468478106
2022-04-04	0.9703636813411461080	0.4818287762321547
2022-04-05	0.3825212419915378210	0.9126501722671725
2022-04-06	0.0028093503788113594	0.0163177577778697

5. Use the MATCHSTR template value

In this example, the `<<FIELD>>` template value is a placeholder for `test`, and the `<<MATCHSTR>>` template value represents the wildcarded value that follows `test` in each field name in the `eval` expression. This example uses the default `multifield` mode.

```
| makeresults | eval test1 = 5, test2 = 10 | foreach test* [eval <<FIELD>> = <<FIELD>> + <<MATCHSTR>>]
```

The results look something like this:

_time	test1	test2
2022-03-28 15:43:39	6	12

The value of each field is added to the value that replaces the wildcard in the field name. For example, for the `test1` field, $5 + 1 = 6$.

6. Use the MATCHSEG1 and MATCHSEG2 template values

This example uses the default `multifield` mode. The `matchseg1` and `matchseg2` options are used to add each field value to the two values represented by the wildcard in the corresponding `<<MATCHSEG1>>` and `<<MATCHSEG2>>` template values.

```
| makeresults | eval test1ab2=5, test2ab3=10 | foreach test*ab* fieldstr=MYFIELD matchseg1=SEG1 matchseg2=SEG2 [eval MYFIELD = MYFIELD + SEG1 + SEG2]
```

Let's take a closer look at the syntax for the `test1ab2=5` `eval` expression:

- ◊ The wildcard field is `test*ab*`.
- ◊ The `<<FIELD>>` template value called `MYFIELD` is `test1ab2=5`.
- ◊ The field value is 5.
- ◊ The `<<MATCHSEG1>>` template value called `SEG1` is 1.
- ◊ The `<<MATCHSEG2>>` template value called `SEG2` is 2.

The results of the search look something like this:

_time	test1ab2	test2ab3
2022-03-28 17:10:56	8	15

The value of the `test1ab2` field in the search results is 8 because $5 + 1 + 2 = 8$.

7. Add values in a multivalue field

In this example using the `multivalue` mode, `<<ITEM>>` is a placeholder for each number in the multivalue field, which is added to the total.

```
| makeresults | eval mvfield=mvappend("1", "2", "3"), total=0 | foreach mode=multivalue mvfield [eval total = total + <<ITEM>>] | table mvfield, total
```

The results of the search look something like this.

_time	mvfield	total
2022-4-20 15:55:50	1	
	2	
	3	6

The previous search produces similar results as the following `eval` search, which also displays the total, but without listing each of the values that make up the total.

```
| makeresults | eval total = 0 | eval total = total + 1 | eval total = total + 2 | eval total = total + 3
```

The search results look like this:

_time	total
2022-4-20 15:55:50	6

8. Categorize employees by manager using multivalue fields

You can create lists of employee names and organize them by manager using the `eval` command or the `foreach` command. This is an example of a search on employees and their manager using the `eval` command:

```
| makeresults | eval manager="Rutherford", employees=mvappend("Alex", "Claudia", "David") | fields - _time
```

The results of the `eval` search look something like this.

employees	manager
Alex	
Claudia	Rutherford
David	

To create multivalue fields of employee names and organize them by manager, you can run a similar search using multivalue fields with the `foreach` command:

```
| makeresults | eval manager="Rutherford", employees=mvappend("Alex", "Claudia", "David"), employees_array=json_array() | foreach mode=multivalue employees [eval employees_array=json_append(employees_array, "", <<ITEM>>)] | fields - _time
```

The search results this time look like this:

employees	employees_array	manager
Alex		
Claudia	["Alex", "Claudia", "David"]	Rutherford
David		

9. Calculate grade averages using multivalue fields

To find the average of a set of student grades using the `multivalue` mode, you could run this search:

```
| makeresults | eval teacher="James", student_grades=mvappend("50", "100", "30"), sum = 0, count = 0 |  
foreach mode=multivalue student_grades [eval sum = sum + <<ITEM>>, count = count + 1] | eval average = sum /  
count
```

The search results look something like this:

_time	average	count	student_grades	sum	teacher
2022-03-21 16:02:30	60	3	50 100 30	180	James

10. Add values in a JSON array

If you want to do something simple like add up each element in a JSON array, you could run a search like this:

```
| makeresults | eval jsonfield=json_array(1, 2, 3), total=0 | foreach mode=json_array jsonfield [eval total  
= total + <<ITEM>>] | table jsonfield, total
```

The search results look like this:

jsonfield	total
[1, 2, 3]	6

11. Add values to a JSON array

Now let's take the names of the employees in a multivalue field and append them to a JSON array. In this search, the `employees_array` is empty.

```
| makeresults | eval manager="Rutherford", employees=mvappend("Alex", "Claudia", "David"),  
employees_array=json_array() | fields - _time
```

The search results look like this:

employees	employees_array	manager
Alex		
Claudia	[]	
David		Rutherford

To copy all the values from the multivalue field into `json_array()`, use `foreach` to iterate over the `employees` values and append each of the employee names to the array, like this search:

```
| makeresults | eval manager="Rutherford", employees=mvappend("Alex", "Claudia", "David"),  
employees_array=json_array() | foreach mode=multivalue employees [eval  
employees_array=json_append(employees_array, "", <<ITEM>>)] | fields - _time
```

Now the search results look like this:

employees	employees_array	manager
Alex		
Claudia	["Alex", "Claudia", "David"]	Rutherford
David		

The `foreach` command just copied over each of the employees' names to the JSON array.

12. Extracting values from a JSON array

What if you want to extract values for given key names from a JSON array and do something with them? For example, the following search extracts a list of employee IDs from a JSON array of employees and puts them in a new field called `ID_array` that you can use for other operations.

```
| makeresults | eval manager="Rutherford", employees=mvappend("Alex", "Claudia", "David"), ID_array=json_array(), IDs=json_object("Alex", 4125, "Claudia", 2538, "David", 3957) | foreach mode=multivalue employees [eval ID_array=json_append(ID_array, "", json_extract(IDs, <<ITEM>>))] | fields -_time
```

The results of this search look something like this:

ID_array	IDs	employees	manager
[4125,2538,3957]	{"Alex":4125,"Claudia":2538,"David":3957}	Alex Claudia David	Rutherford

13. Multiplying elements in a JSON array

You can use the `foreach` command to multiply numbers and append to a JSON array in searches like this:

```
| makeresults | eval price=json_array(1,2,3,4), double_price=json_array() | foreach mode=json_array price [eval double_price = json_append(double_price, "", <<ITEM>> * 2)]
```

The results look something like this:

_time	double_price	price
2022-03-21 16:24:49	[2,4,6,8]	[1,2,3,4]

This search doubles each value in the array in `price` and then adds the values to a new array called `double_price`.

14. Calculating weights

To find the weights of values in a JSON array called `grades`, you could run a search like this:

```
| makeresults | eval grades=json_array(1,2,3,4), weight=json_array() | eval sum = 0 | foreach mode=json_array grades [eval sum = sum + <<ITEM>>] | foreach mode=json_array grades [eval weight = json_append(weight, "", <<ITEM>> / sum)]
```

The search results look something like this:

_time	grades	sum	weight

2022-03-31 12:58:16	[1,2,3,4]	10	[0.1,0.2,0.3,0.4]
---------------------	-----------	----	-------------------

See also

Commands

[eval](#), [map](#)

Related information

Evaluate and manipulate fields with multiple values
[JSON functions](#)

format

Description

This command is used implicitly by subsearches. This command takes the results of a **subsearch**, formats the results into a single result and places that result into a new field called `search`.

The `format` command performs similar functions as the [return](#) command.

Syntax

The required syntax is in **bold**.

```
format
[mvsep=<mv separator>]
[maxresults=<int>]
["<row prefix>" "<column prefix>" "<column separator>" "<column end>" "<row separator>" "<row end>"]
[emptystr=<string>]
```

If you want to specify a row or column options, you must specify all of the row and column options.

Required arguments

None.

Optional arguments

`mvsep`

Syntax: `mvsep=<string>`
Description: The separator to use for multivalue fields.
Default: OR

`maxresults`

Syntax: `maxresults=<int>`
Description: The maximum results to return.
Default: 0, which means no limitation on the number of results returned.

`<row prefix>`

Syntax: "<string>"

Description: The value to use for the row prefix.

Default: The open parenthesis character "("

<column prefix>

Syntax: "<string>"

Description: The value to use for the column prefix.

Default: The open parenthesis character "("

<column separator>

Syntax: "<string>"

Description: The value to use for the column separator.

Default: AND

<column end>

Syntax: "<string>"

Description: The value to use for the column end.

Default: The close parenthesis character ")"

<row separator>

Syntax: "<string>"

Description: The value to use for the row separator.

Default: OR

<row end>

Syntax: "<string>"

Description: The value to use for the column end.

Default: The close parenthesis character ")"

emptystr

Syntax: emptystr=<string>"

Description: The value that the `format` command outputs instead of the default empty string `NOT()` if the results generated up to that point are empty and no fields or values other than internal fields are returned. You can set this argument to a custom string that is displayed instead of the default empty string whenever your search results are empty.

Default: NOT()

Usage

By default, when you do not specify any of the optional row and column arguments, the output of the `format` command defaults to: " (" " (" "AND" " ") " "OR" " ") ".

Specifying row and column arguments

There are several reasons to specify the row and column arguments:

Subsearches

There is an implicit `format` at the end of a subsearch that uses the default values for column and row arguments. For example, you can specify OR for the column separator by including the `format` command at the end of the subsearch.

Export the search to a different system

Specify the row and column arguments when you need to export the search to another system that requires different formatting.

Examples

1. Example with no optional parameters

Suppose that you have results that look like this:

source	sourcetype	host
syslog.log	syslog	my_laptop
bob-syslog.log	syslog	bobs_laptop
laura-syslog.log	syslog	lauras_laptop

The following search returns the top 2 results, and creates a search based on the host, source, and sourcetype fields. The default format settings are used.

```
... | head 2 | fields source, sourcetype, host | format
```

This search returns the syntax for a search that is based on the field values in the top 2 results. The syntax is placed into a new field called **search**.

source	sourcetype	host	search
			((host="mylaptop" AND source="syslog.log" AND sourcetype="syslog") OR (host="bobs_laptop" AND source="bob-syslog.log" AND sourcetype="syslog"))

2. Example using the optional parameters

You want to produce output that is formatted to use on an external system.

```
... | format "[" "[" "&&" "]" "||" "]"
```

Using the data in Example 1, the result is:

source	sourcetype	host	search
			[[host="mylaptop" && source="syslog.log" && sourcetype="syslog"] [host="bobs_laptop" && source="bob-syslog.log" && sourcetype="syslog"]]

3. Multivalue separator example

The following search uses the `eval` command to create a field called "foo" that contains one value "eventtype,log_level". The `makemv` command is used to make the foo field a multivalue field and specifies the comma as the delimiter between the values. The search then outputs only the foo field and formats that field.

```
index=_internal | head 1 | eval foo="eventtype,log_level" | makemv delim="," foo | fields foo | format mvsep="mvseparator" "[" "[" "AND" "] " "AND" "}" "
```

This results in the following output:

foo	search
	{ [(foo="eventtype" mvseparator foo="log_level")] }

4. Use emptystr to indicate empty results

When a search generates empty results, the `format` command returns internal fields and the contents of `emptystr`. You can change the value of `emptystr` from the default to a custom string. For example, the results in the following search are empty, so `format` returns a customized string "Error Found" in a new field called `search`.

```
| makeresults count=1 | format emptystr="Error Found"
```

The results look something like this.

search
Error Found

If your search doesn't include `emptystr` like the following example, the `format` command displays the default empty string to indicate that the results are empty.

```
| makeresults count=1 | format
```

The results look something like this.

search
NOT ()

5. Use emptystr in a subsearch as a failsafe

Customizing your empty string as shown in the last example is one way to use `emptystr`. However, it is more typical to use the `format` command as a subsearch that is operating as a search filter, and then use `emptystr` as a failsafe in case your search returns empty results. For example, perhaps your index isn't generating results because one of the fields you're specifying in the subsearch doesn't exist or there's a typo or some other error in your search. You can include the `emptystr` argument and set it to a default source type that you know is always present, such as `splunkd`. Then, instead of returning nothing, your search will return some results that you can use for further filtering.

You can use the following sample search to make sure you get results even if your search contains errors.

```
index=_internal sourcetype= [search index=does_not_exist | head 1 | fields sourcetype | format emptystr="splunkd"]
```

The results look something like this.

i	Time	Event
>	11/16/21 3:11:33.745 PM	11-16-2021 15:11:33.745 -0800 INFO Metrics - group=thruput, name=thruput, instantaneous_kbps=4.984, instantaneous_eps=20.935, average_kbps=1.667, total_k_processed=182447.000, kb=154.505, ev=649 host = PF32198Dsource = C:\Program Files\Splunk\var\log\splunk\metrics.logsourcetype = splunkd
>	11/16/21	11-16-2021 15:11:33.745 -0800 INFO Metrics - group=thruput, name=syslog_output, instantaneous_kbps=0.000, instantaneous_eps=0.000, average_kbps=0.000,

i	Time	Event
	3:11:33.745 PM	total_k_processed=0.000, kb=0.000, ev=0 host = PF32198Dsource = C:\Program Files\Splunk\var\log\splunk\metrics.logsourcetype = splunkd
>	11/16/21 3:11:33.745 PM	11-16-2021 15:11:33.745 -0800 INFO Metrics - group=thrput, name=index_thrput, instantaneous_kbps=4.971, instantaneous_eps=19.355, average_kbps=1.667, total_k_processed=182424.000, kb=154.094, ev=600 host = PF32198Dsource = C:\Program Files\Splunk\var\log\splunk\metrics.logsourcetype = splunkd
>	11/16/21 3:11:33.745 PM	11-16-2021 15:11:33.745 -0800 INFO Metrics - group=queue, name=winparsing, max_size_kb=500, current_size_kb=0, current_size=0, largest_size=0, smallest_size=0 host = PF32198Dsource = C:\Program Files\Splunk\var\log\splunk\metrics.logsourcetype = splunkd

See also

[search](#), [return](#)

from

Description

The `from` command retrieves data from a dataset, such as a data model dataset, a CSV lookup, a KV Store lookup, a saved search, or a table dataset.

Design a search that uses the `from` command to reference a dataset. Optionally add additional SPL such as lookups, eval expressions, and transforming commands to the search. Save the result as a report, alert, or dashboard panel. If you use Splunk Cloud Platform, or use Splunk Enterprise and have installed the Splunk Datasets Add-on, you can also save the search as a table dataset.

See the **Usage** section.

Syntax

The required syntax is in **bold**.

```
| from
<dataset_type>:<dataset_name> | <dataset_type> <dataset_name>
```

You can specify a colon (:) or a space between `<dataset_type>` and `<dataset_name>`.

Required arguments

`<dataset_type>`

Syntax: `<dataset_type>`

Description: The type of dataset. Valid values are: `datamodel`, `lookup`, and `savedsearch`.

The `datamodel` dataset type can be either a data model dataset or a table dataset. You create data model datasets with the Data Model Editor. You can create table datasets with the Table Editor if you use Splunk Cloud Platform, or use Splunk Enterprise and have installed the Splunk Datasets Add-on.

The `lookup` dataset type can be either a CSV lookup or a KV Store lookup.

The `savedsearch` dataset type is a saved search. You can use `from` to reference any saved search as a dataset. See [About datasets in the Knowledge Manager Manual](#).

<dataset_name>

Syntax: <dataset_name>

Description: The name of the dataset that you want to retrieve data from. If the `dataset_type` is a data model, the syntax is `<datamodel_name>. <dataset_name>`. If the name of the dataset contains spaces, enclose the dataset name in quotation marks.

Example: If the data model name is `internal_server`, and the dataset name is `splunkdaccess`, specify `internal_server.splunkdaccess` for the `dataset_name`.

In older versions of the Splunk software, the term "data model object" was used. That term has been replaced with "data model dataset".

Optional arguments

None.

Usage

The `from` command is a generating command. It can be either report-generating or event-generating depending on the search or knowledge object that is referenced by the command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search. However, you can use the `from` command inside the `append` command.

When you use the `from` command, you must reference an existing dataset. You can reference any dataset listed in the Datasets listing page, such as data model datasets, CSV lookup files, CSV lookup definitions, and table datasets. You can also reference saved searches and KV Store lookup definitions. See [View and manage datasets in the Knowledge Manager Manual](#).

Knowledge object dependencies

When you create a knowledge object such as a report, alert, dashboard panel, or table dataset, that knowledge object has a dependency on the referenced dataset. This is referred to as a dataset extension. When you make a change to the original dataset, such as removing or adding fields, that change propagates down to the reports, alerts, dashboard panels, and tables that have been extended from that original dataset. See [Dataset extension in the Knowledge Manager Manual](#).

When field filtering is disabled for a data model

When you search the contents of a data model using the `from` command, by default the search returns a strictly-filtered set of fields. It returns only default fields and fields that are explicitly identified in the constraint search that defines the data model.

If you have edit access to your local `datamodel.conf` file, you can disable field filtering for specific data models by adding the `strict_fields=false` setting to their stanzas. When you do this, `| from` searches of data models with that setting return all fields related to the data model, including fields inherited from parent data models, fields extracted at search time, calculated fields, and fields derived from lookups.

Examples

1. Search a data model

Search a data model that contains internal server log events for REST API calls. In this example, `internal_server` is the data model name and `splunkdaccess` is the dataset inside the `internal_server` data model.

```
| from datamodel:internal_server.splunkdaccess
```

2. Search a lookup file

Search a lookup file that contains geographic attributes for each country, such as continent, two-letter ISO code, and subregion.

```
| from lookup geo_attr_countries.csv
```

3. Retrieve data by using a lookup file

Search the contents of the KV store collection `kvstorecoll` that have a `CustID` value greater than 500 and a `CustName` value that begins with the letter P. The collection is referenced in a lookup table called `kvstorecoll_lookup`. Using the `stats` command, provide a count of the events received from the table.

```
| from lookup:kvstorecoll_lookup | where (CustID>500) AND (CustName="P*") | stats count
```

4. Retrieve data using a saved search

This search retrieves the timestamp and client IP from the saved search called `mysecurityquery`.

```
| from savedsearch:mysecurityquery | fields _time clientip ...
```

The search results look something like this.

i	Time	Event	
✓	8/28/22 6:20:56.000 PM	182.236.164.11 -- [28/Aug/2022:18:20:56] "GET /cart.do?action=addtocart&itemId=EST-15&productId=BS-AG-G09&JSESSIONID=SD6SL8FF10ADFF53101 HTTP/1.1" 200 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 506	
Event Actions ▾			
Type	<input checked="" type="checkbox"/>	Field	Value
Event	<input type="checkbox"/>	clientip	182.236.164.11
Time	_time	2022-08-28T18:20:56.000-07:00	
➤	8/28/22 6:20:55.000 PM	182.236.164.11 -- [28/Aug/2022:18:20:55] "POST /oldlink?itemId=EST-18&JSESSIONID=SD6SL8FF10ADFF53101 HTTP/1.1" 408 893 "http://www.intosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 134	
➤	8/28/22 6:20:54.000 PM	182.236.164.11 -- [28/Aug/2022:18:20:54] "GET /category.screen?categoryId=ACCESSORIES&JSESSIONID=SD6SL8FF10ADFF53101 HTTP/1.1" 200 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 648	

Even if the saved search is scheduled, this search is rerun, which can be expensive and lead to concurrency issues if more searches are run at the same time than the system can support. Alternatively, you can use the `loadjob` command instead of the `from` command in conjunction with a scheduled search if you are concerned about the number and frequency of searches that your users run.

5. Specify a dataset name that contains spaces

When the name of a dataset includes spaces, enclose the dataset name in quotation marks.

```
| from savedsearch "Top five sourcetypes"
```

See also

Commands

[datamodel](#)
[inputlookup](#)
[inputcsv](#)
[lookup](#)
[loadjob](#)

fromjson

Description

Converts JSON-formatted objects into multivalue fields. If you give the `fromjson` command a single field name that points to proper JSON objects, `fromjson` returns keys as fields and key values as field values.

Syntax

Required syntax is in **bold**.

```
| fromjson<string>
[ prefix=<string>]
```

Optional arguments

prefix

Syntax: `prefix=<string>`

Description: Prepends a string to the fields that `fromjson` extracts from a JSON-formatted object. For example, including `prefix=my_` in the search adds `my_` to the beginning of field names in the results.

Default: none

Usage

The `fromjson` command is a **streaming command**, which means that it turns JSON-formatted objects into fields as each JSON object is received. See [Types of commands](#).

Examples

1. Expand a JSON object to create new fields

Use the `fromjson` command to expand a JSON-formatted object and return the values in the search result. This example creates two new fields called `name` and `age`, and outputs the corresponding values in the search results.

```
| makeresults | eval object=json_object("name", "Albert", "age", 63) | fromjson object
```

The results look something like this.

_time	age	name	object
2020-11-09 17:01:22	63	Albert	{"name": "Albert", "age": 63}

2. Prepend the name of extracted fields

You can use the optional argument `prefix` to prepend a string to fields extracted from a JSON-formatted object. This example creates two new fields called `json_name` and `json_age`.

```
| makeresults | eval object=json_object("name", "Albert", "age", 63) | fromjson object prefix=my_
```

The results look something like this.

_time	my_age	my_name	object
2020-11-09 17:01:22	63	Albert	{"name": "Albert", "age": 63}

3. Expand nested JSON objects

When you use `fromjson` to expand JSON-formatted objects into multivalue fields, you can retain the formatting of JSON objects by nesting them within the main object. In the following example, the object called `json_obj` with the key-value pair "school" and "city", is nested within another JSON object called `object`.

```
| makeresults | eval object=json_object("age", 19, "name", "Sally", "new", false(), "classes", json_array("math", "history", "science"), "another_json_obj", json_object("school", "city"), "null", null) | fromjson object
```

The results look something like this.

_time	age	another_json_obj	classes	name	new	object
2020-11-09 17:01:22	19	{"school": "city"}	math history science	Sally	false	{"age": 19, "name": "Sally", "new": false, "classes": ["math", "history", "science"], "another_json_obj": {"school": "city"}}

See also

Commands

[tojson](#)

Evaluation functions

gauge

Description

Use the `gauge` command to transform your search results into a format that can be used with the gauge charts. Gauge charts are a visualization of a single aggregated metric, such as a count or a sum.

The output of the `gauge` command is a single numerical value stored in a field called `x`. You can specify a range to display in the gauge or use the default range of 0 to 100.

For more information about using the `gauge` command with the gauge chart types, see [Using gauges in the Gauges section in Dashboards and Visualizations](#).

Syntax

```
gauge <value> [<range_val1> <range_val2> ...]
```

Required arguments

`value`

Syntax: `field_name | <num>`

Description: A numeric field or literal number to use as the current value of the gauge. If you specify a numeric field, the `gauge` command uses the first value in that field as the value for the gauge.

Optional arguments

`range values`

Syntax: `<range_val1> <range_val2> ...`

Description: A space-separated list of two or more numeric fields or numbers to use as the overall numeric range displayed in the gauge. Each range value can be a numeric field name or a literal number. If you specify a field name, the first value in that field is used as the range value. The total range of the gauge is from the first `range_val` to the last `range_val`. See [Usage](#).

Default range: 0 to 100

Usage

You can create gauge charts without using the `gauge` command as long as your search results in a single value. The advantage of using the `gauge` command is that you can specify a set of range values instead of using the default range values of 0 to 100.

Specifying ranges

If you specify range values, you must specify at least two values. The gauge begins at the first value and ends at the last value that you specify.

If you specify more than two `range_val` arguments, the intermediate range values are used to split the total range into subranges. Each subrange displays in different color, which creates a visual distinction.

The range values are returned as a series of fields called `y1`, `y2`, and so on.

If you do not specify range values, the range defaults to a low value of 0 and a high value of 100.

If a single range value is specified, it is ignored.

Gauge colors

With a gauge chart, a single numerical value is mapped against a set of colors. These colors can have particular business meaning or business logic. As the value changes over time, the gauge marker changes position within this range.

The color ranges in the gauge chart are based on the range values that you specify with the `gauge` command. When you specify range values, you define the overall numerical range represented by the gauge. You can define the size of the colored bands within that range. If you want to use the color bands, add four range values to the search string. These range values indicate the beginning and end of the range. These range values also indicate the relative sizes of the color bands within this range.

Examples

1. Create a gauge with multiple ranges

Count the number of events and display the count on a gauge with four ranges, from 0-750, 750-1000, 1000-1250, and 1250-1500.

Start by generating the results table using this search. Run the search using the **Last 15 minutes** time range.

```
index=_internal | stats count as myCount | gauge myCount 750 1000 1250 1500
```

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

x	y1	y2	y3	y4
3321	750	1000	1250	1500

Click on the **Visualizations** tab. There are three types of gauges that you can choose from: radial, filler, and marker. The following image shows the radial gauge that is created based on the search results.



For more information about using the `gauge` command with the gauge chart type, see the Gauges section in *Dashboard and Visualizations*.

See also

Commands

[eval](#)
[stats](#)

gentimes

Description

The `gentimes` command is useful in conjunction with the `map` command.

Generates timestamp results starting with the exact time specified as start time. Each result describes an adjacent, non-overlapping time range as indicated by the increment value. This terminates when enough results are generated to pass the endtime value.

The `gentimes` command generates events up to the end time, but not including the end time.

Syntax

```
| gentimes start=<timestamp> [end=<timestamp>] [increment=<increment>]
```

Required arguments

start

Syntax: `start=<timestamp>`

Description: Specify as start time.

<timestamp>

Syntax: `MM/DD/YYYY[:HH:MM:SS] | <int>`

Description: Indicate the timeframe, using either a timestamp or an integer value. For example: 10/1/2020 for October 1, 2020, 4/1/2021:12:34:56 for April 1, 2021 at 12:34:56, or -5 for five days ago.

Optional arguments

end

Syntax: `end=<timestamp>`

Description: Specify an end time.

Default: midnight, prior to the current time in local time

increment

Syntax: `increment=<int>(s | m | h | d)`

Description: Specify a time period to increment from the start time to the end time. Supported increments are seconds, minutes, hours, and days.

Default: 1d

Usage

The `gentimes` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

The `gentimes` command returns four fields.

Field	Description
starttime	The starting time range in UNIX time.
starthuman	The human readable time range in the format DDD MMM DD HH:MM:SS YYYY. For example Sun Apr 4 00:00:00 2021.
endtime	The ending time range in UNIX time.
endhuman	The human readable time range in the format DDD MMM DD HH:MM:SS YYYY. For example Fri Apr 16 23:59:59 2021.

To specify future dates, you must include the `end` argument.

Examples

1. Generate daily time ranges by specifying dates

Generates daily time ranges from April 4 to April 7 in 2021. This search generates events up to the end time, but not including the end time. This search generates three intervals covering one day periods aligning with the calendar days April 4, 5, and 6, during 2021. The `gentimes` command generates events up to the end time, but not including the end time.

```
| gentimes start=4/4/21 end=4/7/21
```

The results look something like this:

starttime	starthuman	endtime	endhuman
1617519600	Sun Apr 4 00:00:00 2021	1617605999	Sun Apr 4 23:59:59 2021
1617606000	Mon Apr 5 00:00:00 2021	1617692399	Mon Apr 5 23:59:59 2021
1617692400	Tue Apr 6 00:00:00 2021	1617778799	Tue Apr 6 23:59:59 2021

2. Generate daily time ranges by specifying relative times

Generate daily time ranges from 30 days ago until 27 days ago.

```
| gentimes start=-30 end=-27
```

3. Generate hourly time ranges

Generate hourly time ranges from December 1 to December 5 in 2021.

```
| gentimes start=12/1/21 end=12/5/21 increment=1h
```

4. Generate time ranges by only specifying a start date

Generate daily time ranges from April 25 to today.

```
| gentimes start=4/25/22
```

5. Generate weekly time ranges

Although the week increment is not supported, you can generate a weekly increment by specifying `increment=7d`.

This example generates weekly time ranges from December 1, 2021 to April 30, 2022.

```
| gentimes start=12/1/21 end=4/30/22 increment=7d
```

See also

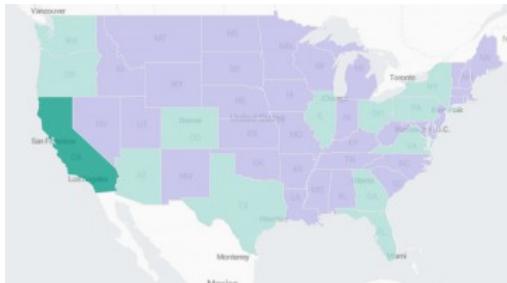
Commands

[makeresults](#)
[map](#)

geom

Description

The `geom` command adds a field, named **geom**, to each result. This field contains geographic data structures for polygon geometry in JSON. These geographic data structures are used to create choropleth map visualizations.



For more information about choropleth maps, see Mapping data in the *Dashboards and Visualizations* manual.

Syntax

```
geom [<featureCollection>] [allFeatures=<boolean>] [featureIdField=<string>] [gen=<double>] [min_x=<double>]  
[min_y=<double>] [max_x=<double>] [max_y=<double>]
```

Required arguments

None.

Optional arguments

featureCollection

Syntax: <geo_lookup>

Description: Specifies the geographic lookup file that you want to use. Two geographic lookup files are included by default with Splunk software: geo_us_states and geo_countries. You can install your own geographic lookups from KMZ or KML files. See [Usage](#) for more information.

allFeatures

Syntax: allFeatures=<bool>

Description: Specifies that the output include every geometric feature in the feature collection. When a shape has no values, any aggregate fields, such as average or count, display zero when this argument is used. Additional rows are appended for each feature that is not already present in the search results when this argument is used. See [Examples](#).

Default: false

featureIdField

Syntax: featureIdField=<field>

Description: If the event contains the featureId in a field named something other than "featureId", use this option to specify the field name.

gen

Syntax: gen=<double>

Description: Specifies generalization, in the units of the data. For example, gen=0.1 generalizes, or reduces the size of, the geometry by running the Douglas-Puett Ramer algorithm on the polygons with a parameter of 0.1 degrees.

Default: 0.1

min_x

Syntax: min_x=<double>

Description: The X coordinate for the bottom-left corner of the bounding box for the geometric shape. The range for the coordinate is -180 to 180. See [Usage](#) for more information.

Default: -180

min_y

Syntax: min_y=<double>

Description: The Y coordinate for the bottom-left corner of the bounding box for the geometric shape. The range for the coordinate is -90 to 90.

Default: -90

max_x

Syntax: max_x=<double>

Description: The X coordinate for the upper-right corner of the bounding box for the geometric shape. The range for the coordinate is -180 to 180.

Default: 180

max_y

Syntax: max_y=<double>

Description: The Y coordinate for the upper-right corner of the bounding box for the geometric shape. The range is -90 to 90.

Default: 90

Usage

Specifying a lookup

To use your own lookup file in Splunk Enterprise, you can define the lookup in Splunk Web or edit the `transforms.conf` file. If you use Splunk Cloud Platform, use Splunk Web to define lookups.

Define a geospatial lookup in Splunk Web

1. To create a geospatial lookup in Splunk Web, you use the **Lookups** option in the **Settings** menu. You must add the lookup file, create a lookup definition, and can set the lookup to work automatically. See Define a geospatial lookup in Splunk Web in the *Knowledge Manager Manual*.

Configure a geospatial lookup in transforms.conf

1. Edit the `%SPLUNK_HOME%\etc\system\local\transforms.conf` file, or create a new file named `transforms.conf` in the `%SPLUNK_HOME%\etc\system\local` directory, if the file does not already exist. See How to edit a configuration file in the *Admin Manual*.
2. Specify the name of the lookup stanza in the `transforms.conf` file for the `featureCollection` argument.
3. Set `external_type=geo` in the stanza. See Configure geospatial lookups in the *Knowledge Manager Manual*.

Specifying no optional arguments

When no arguments are specified, the `geom` command looks for a field named `featureCollection` and a field named `featureIdField` in the event. These fields are present in the default output from a geoindex lookup.

Clipping the geometry

The `min_x`, `min_y`, `max_x`, and `max_y` arguments are used to clip the geometry. Use these arguments to define a bounding box for the geometric shape. You can specify the minimum rectangle corner (`min_x`, `min_y`) and the maximum rectangle corner (`max_x`, `max_y`). By specifying the coordinates, you are returning only the data within those coordinates.

Testing lookup files

You can use the `inputlookup` command to verify that the geometric features on the map are correct. The syntax is `| inputlookup <your_lookup>`.

For example, to verify that the geometric features in built-in `geo_us_states` lookup appear correctly on the choropleth map:

1. Run the following search:

```
| inputlookup geo_us_states
```

2. On the **Visualizations** tab, change to a Choropleth Map.
3. zoom in to see the geometric features. In this example, the states in the United States.

Testing geometric features

You can create an arbitrary result to test the geometric features.

To show how the output appears with the `allFeatures` argument, the following search creates a simple set of fields and values.

```
| stats count | eval featureId="California" | eval count=10000 | geom geo_us_states allFeatures=true
```

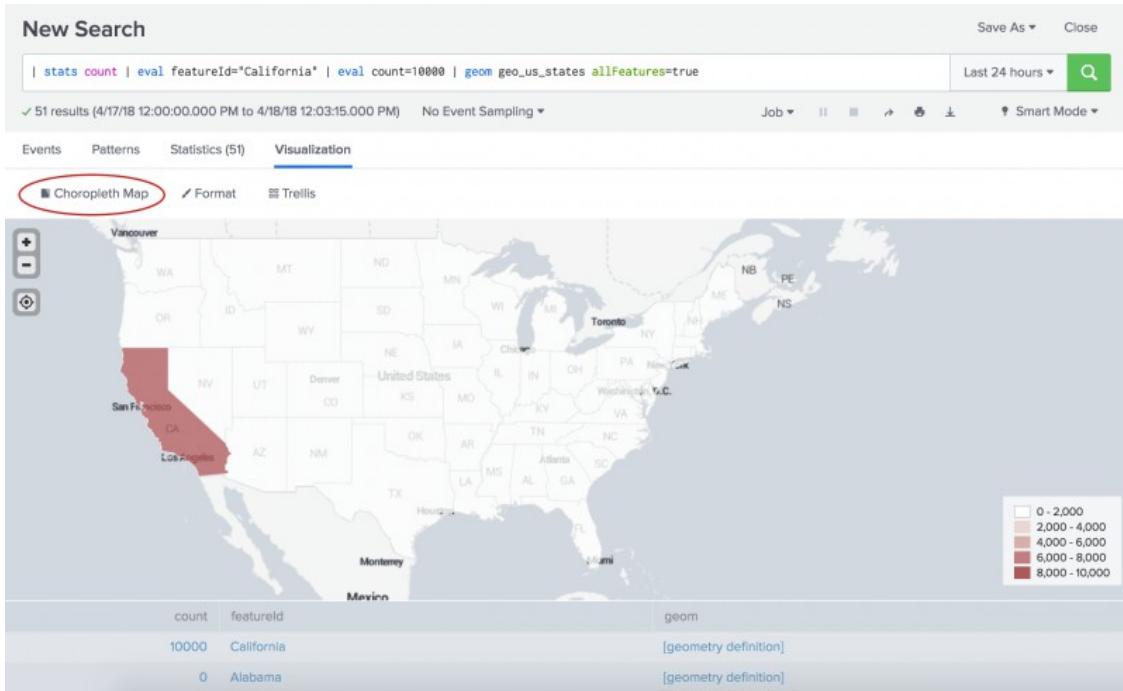
- The search uses the `stats` command, specifying the `count` field. A single result is created that has a value of zero (0) in the `count` field.
- The `eval` command is used to add the `featureId` field with value of California to the result.
- Another `eval` command is used to specify the value 10000 for the `count` field. You now have a single result with two fields, `count` and `featureId`.
- When the `geom` command is added, two additional fields are added, `featureCollection` and `geom`.

The following image shows the results of the search on the **Statistics** tab.

The screenshot shows the Splunk interface with the "New Search" bar at the top containing the search command: `| stats count | eval featureId="California" | eval count=10000 | geom geo_us_states allFeatures=true`. Below the search bar are buttons for "Save As" and "Close". A search history dropdown shows "Last 24 hours" and a magnifying glass icon. The main area displays search results for 51 results from April 17, 2018, to April 18, 2018. The results are filtered by "Events" and "Patterns" and are currently displayed in "Statistics (51)" mode. The results table includes columns for count, featureCollection, featureId, and geom. The first result for California is shown in detail, displaying its coordinates as a multi-polygon shape. The second result for Alabama is also partially visible.

count	featureCollection	featureId	geom
10000	geo_us_states	California	["type": "MultiPolygon", "coordinates": [[[[-118.60337829589844, 33.47809982299805], [-118.60337829589844, 33.47809982299805], [-118.60553741455078, 33.03099822998047], [-118.60553741455078, 33.03099822998047], [-119.57894134521484, 33.278629302978516], [-119.57894134521484, 33.278629302978516], [-119.91915130615234, 34.07727813720703], [-119.91915130615234, 34.07727813720703], [-120.24739074707031, 34.00191116333008], [-120.24739074707031, 34.00191116333008], [-120.46257781982422, 34.042625427246094], [-120.46257781982422, 34.042625427246094], [-122.37787628173828, 37.83064651489258], [-122.37787628173828, 37.83064651489258], [-122.44631958007812, 37.861045837402344], [-122.44631958007812, 37.861045837402344], [-124.4085985351562, 40.44319157714844], [-124.4085985351562, 40.44319157714844], [-124.21160125732422, 41.99845886230469], [-119.9991683959961, 41.99454116821289], [-120.00101470947266, 38.9995275390625], [-114.63348388671875, 35.00185775756836], [-114.52685546875, 32.75709533691406], [-120.62257385253906, 34.55401611328125], [-124.4085985351562, 40.44319157714844]]]]]
0	geo_us_states	Alabama	["type": "MultiPolygon", "coordinates": [[[[-88.31002807617188, 30.233232498168945], [-88.31002807617188, 30.233232498168945], [-88.47322845458984, 31.893856048583984], [-88.20295715332031, 35.008026123046875], [-85.60516357421875, 34.984676361083984], [-85.00250244140625, 31.000682830810547], [-88.02840423583984, 30.22132278442383], [-88.47322845458984, 31.893856048583984]]]]]

The following image shows the results of the search on the **Visualization** tab. Make sure that the map is a **Choropleth Map**. This image is zoomed in to show more detail.



Examples

1. Use the default settings

When no arguments are provided, the `geom` command looks for a field named **featureCollection** and a field named **featureId** in the event. These fields are present in the default output from a geospatial lookup.

```
... | geom
```

2. Use the built-in geospatial lookup

This example uses the built-in `geo_us_states` lookup file for the **featureCollection**.

```
... | geom geo_us_states
```

3. Specify a field that contains the `featureId`

This example uses the built-in `geo_us_states` lookup and specifies `state` as the **featureIdField**. In most geospatial lookup files, the feature IDs are stored in a field called `featureId`. Use the `featureIdField` argument when the event contains the feature IDs in a field named something other than "featureId".

```
... | geom geo_us_states featureIdField="state"
```

4. Show all geometric features in the output

The following example specifies that the output include every geometric feature in the feature collection. If no value is present for a geometric feature, zero is the default value. Using the `allFeatures` argument causes the choropleth map visualization to render all of the shapes.

```
... | geom geo_us_states allFeatures=true
```

5. Use the built-in countries lookup

The following example uses the built-in geo_countries lookup. This search uses the `lookup` command to specify shorter field names for the latitude and longitude fields. The `stats` command is used to count the feature IDs and renames the `featureIdField` field as `country`. The `geom` command generates the information for the choropleth map using the renamed field `country`.

```
... | lookup geo_countries latitude AS lat, longitude AS long | stats count BY featureIdField AS country | geom geo_countries featureIdField="country"
```

6. Specify the bounding box for the geometric shape

This example uses the `geom` command attributes that enable you to clip the geometry by specifying a bounding box.

```
... | geom geo_us_states featureIdField="state" gen=0.1 min_x=-130.5 min_y=37.6 max_x=-130.1 max_y=37.7
```

See also

Mapping data in the *Dashboards and Visualizations* manual.

geomfilter

Description

Use the `geomfilter` command to specify points of a bounding box for clipping choropleth maps.

For more information about choropleth maps, see "Mapping data" in the *Dashboards and Visualizations* Manual.

Syntax

```
geomfilter [min_x=<float>] [min_y=<float>] [max_x=<float>] [max_y=<float>]
```

Optional arguments

`min_x`

Syntax: `min_x=<float>`

Description: The x coordinate of the bounding box's bottom-left corner, in the range [-180, 180].

Default: -180

`min_y`

Syntax: `min_y=<float>`

Description: The y coordinate of the bounding box's bottom-left corner, in the range [-90, 90].

Default: -90

`max_x`

Syntax: `max_x=<float>`

Description: The x coordinate of the bounding box's up-right corner, in the range [-180, 180].

Default: 180

`max_y`

Syntax: max_y=<float>

Description: The y coordinate of the bounding box's up-right corner, in the range [-90, 90].

Default: max_y=90

Usage

The geomfilter command accepts two points that specify a bounding box for clipping choropleth maps. Points that fall outside of the bounding box will be filtered out.

Examples

Example 1: This example uses the default bounding box, which will clip the entire map.

```
... | geomfilter
```

Example 2: This example clips half of the whole map.

```
... | geomfilter min_x=-90 min_y=-90 max_x=90 max_y=90
```

See also

[geom](#)

geostats

Description

Use the `geostats` command to generate statistics to display geographic data and summarize the data on maps.

The command generates statistics which are clustered into geographical bins to be rendered on a world map. The events are clustered based on latitude and longitude fields in the events. Statistics are then evaluated on the generated clusters. The statistics can be grouped or split by fields using a `BY` clause.

For map rendering and zooming efficiency, the `geostats` command generates clustered statistics at a variety of zoom levels in one search, the visualization selecting among them. The quantity of zoom levels is controlled by the `binspanlat`, `binspanlong`, and `maxzoomlevel` options. The initial granularity is selected by the `binspanlat` and the `binspanlong`. At each level of zoom, the number of bins is doubled in both dimensions for a total of 4 times as many bins for each zoom in.

Syntax

The required syntax is in **bold**.

```
geostats
[ translatetoxy=<bool> ]
[ latfield=<string> ]
```

```
[ longfield=<string> ]
[ globallimit=<int> ]
[ locallimit=<int> ]
[ outputlatfield=<string> ]
[ outputlongfield=<string> ]
[ binspanlat=<float> binspanlong=<float> ]
[ maxzoomlevel=<int> ]
<stats-agg-term>...
[ <by-clause> ]
```

Required arguments

stats-agg-term

Syntax: <stats-func> (<evalued-field> | <wc-field>) [AS <wc-field>]

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names. For more information on eval expressions, see Types of eval expressions in the *Search Manual*.

Optional arguments

binspanlat

Syntax: binspanlat=<float>

Description: The size of the bins in latitude degrees at the lowest zoom level. If you set binspanlat lower than the default value, the visualizations on the map might not render.

Default: 22.5. If the default values for binspanlat and binspanlong are used, a grid size of 8x8 is generated.

binspanlong

Syntax: binspanlong=<float>

Description: The size of the bins in longitude degrees at the lowest zoom level. If you set binspanlong lower than 33, the visualizations on the map might not render.

Default: 45.0. If the default values for binspanlat and binspanlong are used, a grid size of 8x8 is generated.

by-clause

Syntax: BY <field>

Description: The name of the field to group by.

globallimit

Syntax: globallimit=<int>

Description: Controls the number of named categories to add to each pie chart. There is one additional category called "OTHER" under which all other split-by values are grouped. Setting globallimit=0 removes all limits and all categories are rendered. Currently the grouping into "OTHER" only works intuitively for count and additive statistics.

Default: 10

locallimit

Syntax: locallimit=<int>

Description: Specifies the limit for series filtering. When you set locallimit=N, the top N values are filtered based on the sum of each series. If locallimit=0, no filtering occurs.

Default: 10

latfield

Syntax: latfield=<field>

Description: Specify a field from the pre-search that represents the latitude coordinates to use in your analysis.

Defaults: lat

longfield

Syntax: longfield=<field>

Description: Specify a field from the pre-search that represents the longitude coordinates to use in your analysis.

Default: lon

maxzoomlevel

Syntax: maxzoomlevel=<int>

Description: The maximum number of levels to create in the quadtree.

Default: 9. Specifies that 10 zoom levels are created, 0-9.

outputlatfield

Syntax: outputlatfield=<string>

Description: Specify a name for the latitude field in your geostats output data.

Default: latitude

outputlongfield

Syntax: outputlongfield=<string>

Description: Specify a name for the longitude field in your geostats output data.

Default: longitude

translatetox

Syntax: translatetox=<bool>

Description: If true, geostats produces one result per each locationally binned location. This mode is appropriate for rendering on a map. If false, geostats produces one result per category (or tuple of a multiply split dataset) per locationally binned location. Essentially this causes the data to be broken down by category. This mode cannot be rendered on a map.

Default: true

Stats function options

stats-func

Syntax: The syntax depends on the function that you use. See [Usage](#).

Description: Statistical and charting functions that you can use with the `geostats` command. Each time you invoke the `geostats` command, you can use one or more functions.

Usage

To display the information on a map, you must run a reporting search with the `geostats` command.

If you are using a `lookup` command before the `geostats` command, see [Optimizing your lookup search](#).

Supported functions

You can use a wide range of functions with the `geostats` command. For general information about using functions, see [Statistical and charting functions](#).

- For a list of statistical functions by category, see [Function list by category](#)

- For an alphabetical list of statistical functions, see [Alphabetical list of functions](#)

Memory and geostats search performance

A pair of `limits.conf` settings strike a balance between the performance of `geostats` searches and the amount of memory they use during the search process, in RAM and on disk. If your `geostats` searches are consistently slow to complete you can adjust these settings to improve their performance, but at the cost of increased search-time memory usage, which can lead to search failures.

For more information, see Memory and stats search performance in the *Search Manual*.

Basic examples

1. Use the default settings and calculate the count

Cluster events by default latitude and longitude fields "lat" and "lon" respectively. Calculate the count of the events.

```
... | geostats count
```

2. Specify the latfield and longfield and calculate the average of a field

Compute the average rating for each gender after clustering/grouping the events by "eventlat" and "eventlong" values.

```
... | geostats latfield=eventlat longfield=eventlong avg(rating) by gender
```

Extended examples

3. Count each product sold by a vendor and display the information on a map

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

In addition, this example uses several lookup files that you must download (`prices.csv.zip` and `vendors.csv.zip`) and unzip the files. You must complete the steps in the Enabling field lookups section of the tutorial for both the `prices.csv` and the `vendors.csv` files. The steps in the tutorial are specific to the `prices.csv` file. For the `vendors.csv` file, use the name **vendors_lookup** for the lookup definition. Skip the step in the tutorial that makes the lookups automatic.

This search uses the `stats` command to narrow down the number of events that the `lookup` and `geostats` commands need to process.

Use the following search to count each product sold by a vendor and display the information on a map.

```
sourcetype=vendor_sales | stats count by Code VendorID | lookup prices_lookup Code OUTPUTNEW product_name | table product_name VendorID | lookup vendors_lookup VendorID | geostats latfield=VendorLatitude longfield=VendorLongitude count by product_name
```

- In this example, `sourcetype=vendor_sales` is associated with a log file that is included in the Search Tutorial sample data. This log file contains vendor information that looks like this:

```
[10/Apr/2018:18:24:02]  VendorID=5036  Code=B  AcctID=6024298300471575
```

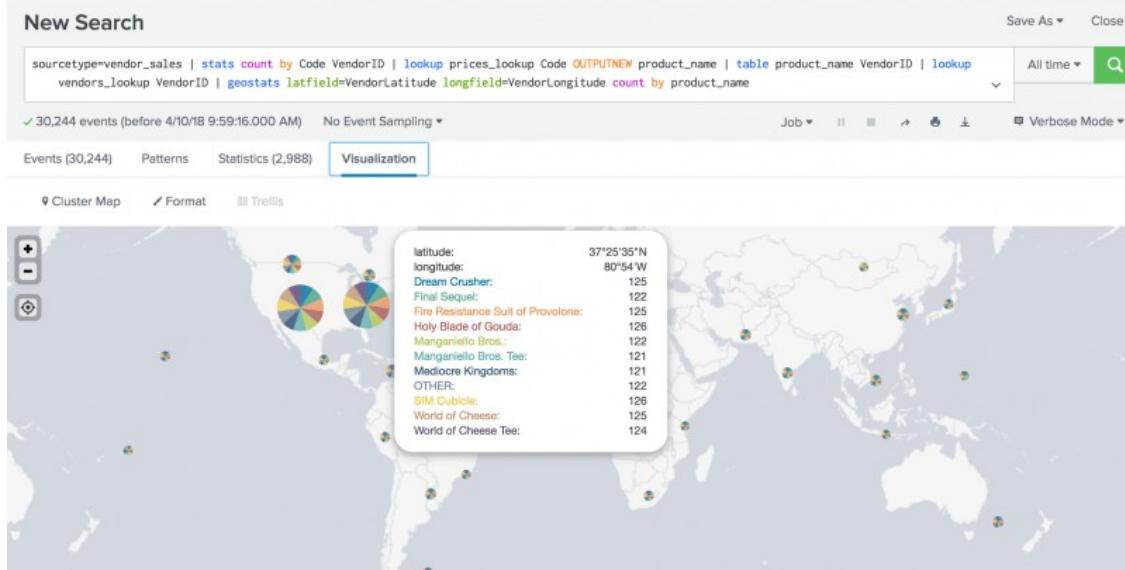
- The `vendors_lookup` is used to output all the fields in `vendors.csv` file that match to the VendorID in the `vendor_sales.log` file. The fields in the `vendors.csv` file are : Vendor, VendorCity, VendorID, VendorLatitude, VendorLongitude, VendorStateProvince, and VendorCountry.
- The `prices_lookup` is used to match the Code field in each event to a product_name in the table.

In this search, the CSV files are uploaded and the lookups are defined but are not automatic.

This search produces a table displayed on the **Statistics** tab:

New Search															
sourcetype=vendor_sales stats count by Code VendorID lookup prices_lookup Code OUTPUTNEW product_name table product_name VendorID lookup vendors_lookup VendorID geostats latfield=VendorLatitude longfield=VendorLongitude count by product_name															
✓ 30,244 events (before 4/10/18 9:59:16.000 AM) No Event Sampling ▾ Job ▾ All time ▾ Verbose Mode ▾															
Events (30,244) Patterns Statistics (2,988) Visualization															
100 Per Page ▾ Format Preview ▾															
geobin	Dream Crusher	Final Sequel	Fire Resistance Suit of Provalone	Holy Blade of Gouda	Manganelli Bros.	Manganelli Bros. Tee	Mediocre Kingdoms	OTHER	SIM Cubicle	World of Cheese	World of Cheese Tee	latitude	longitude		
bin_id_zl_0_y_1_x_2	1	1	1	1	1	1	1	1	1	1	1	-51.70778	-57.82654		
bin_id_zl_0_y_2_x_2	7	7	7	7	7	7	7	7	6	7	7	-29.44057	-56.45329		
bin_id_zl_0_y_2_x_3	2	2	1	2	2	2	2	2	2	1	2	-22.88780	-43.21717		
bin_id_zl_0_y_2_x_4	3	3	3	3	2	3	3	3	3	3	3	-30.02566	25.54097		
bin_id_zl_0_y_2_x_7	6	6	6	6	3	4	6	6	6	6	5	-38.13657	157.60132		
bin_id_zl_0_y_3_x_0	1	1	1	1	1	1	1	1	1	1	1	-14.33100	-170.71001		

Click the **Visualization** tab. The results are plotted on a world map. There is a pie chart for each vendor in the results. The larger the pie chart, the larger the count value.



In this screen shot, the mouse pointer is over the pie chart for a region in the northeastern part of the United States. An popup information box displays the latitude and longitude for the vendor, as well as a count of each product that the vendor sold.

You can zoom in to see more details on the map.

See also

Commands

[iplocation](#)
[stats](#)
[xseries](#)

Reference information

Mapping data in *Dashboards and Visualizations*

head

Description

Returns the first N number of specified results in search order. This means the most recent N events for a historical search, or the first N captured events for a real-time search. The search results are limited to the first results in search order.

There are two types of limits that can be applied: an absolute number of results, or an expression where all results are returned until the expression becomes false.

Syntax

The required syntax is in **bold**.

```
head
[<N> | (<eval-expression>)]
[limit=<int>]
[null=<bool>]
[keeplast=<bool>]
```

Required arguments

None.

If no options or limits are specified, the `head` command returns the first 10 results.

Optional arguments

<N>

Syntax: <int>

Description: The number of results to return.

Default: 10

limit

Syntax: limit=<int>

Description: Another way to specify the number of results to return.

Default: 10

eval-expression

Syntax: <eval-compare-exp> | <eval-bool-exp>

Description: A valid <eval-expression> that evaluates to a Boolean. The search returns results until this expression evaluates to false. For more information, see the [evaluation functions](#) in the *Search Reference*.

keeplast

Syntax: keeplast=<bool>

Description: You must specify a eval-expression to use the keeplast argument. Controls whether the last result in the result set is retained. The last result returned is the result that caused the eval-expression to evaluate to false or NULL. Set keeplast to true to retain the last result in the result set. Set keeplast to false to discard the last result.

Default: false

null

Syntax: null=<bool>

Description: You must specify an <eval-expression> for the null argument to have any effect. Controls how an <eval-expression> that evaluates to NULL is handled. For example, if the <eval-expression> is `(x > 10)` and a value in field x does not exist, the <eval-expression> evaluates to NULL instead of true or false.

- ◊ If null=true, the results of the head command include events for which <eval-expression> evaluates to NULL in the output. The head command continues to process the remaining events.
- ◊ If null=false, the head command treats the <eval-expression> that evaluates to NULL as if the <eval-expression> evaluated to false. The head command stops processing events. If keeplast=true, the event for which the <eval-expression> evaluated to NULL is also included in the output.

Default: false

Usage

The head command is a [centralized streaming command](#). See [Command types](#).

Setting limits

If a numeric limit such as a numeric literal or the argument limit=<int> is used, the head command returns the first N results where N is the selected number. Using both the numeric limit and limit=<int> results in an error.

Using an <eval-expression>

If an <eval-expression> is used, all initial results are returned until the first result where the expression evaluates to false. The result where the <eval-expression> evaluates to false is kept or dropped based on the keeplast argument.

If both a numeric limit and an <eval-expression> are used, the smaller of the two constraints applies. For example, the following search returns up to the first 10 results, because the <eval-expression> is always true.

```
... |head limit=10 (1==1)
```

However, this search returns no results because the <eval-expression> is always false.

```
... |head limit=10 (0==1)
```

Basic examples

1. Return a specific number of results

Return the first 20 results.

```
... | head 20
```

2. Return results based on a specified limit

Return events until the time span of the data is ≥ 100 seconds

```
... | streamstats range(_time) as timerange | head (timerange<100)
```

Extended example

1. Using the `keeplast` and `null` arguments

The following example shows the search results when an <eval-expression> evaluates to NULL, and the impact of the `keeplast` and `null` arguments on those results.

Let's start with creating a set of events. The `eval` command replaces the value 3 with NULL in the count field.

```
| makeresults count=7 | streamstats count | eval count=if(count=3,null(), count)
```

The results look something like this:

_time	count
2020-05-18 12:46:51	1
2020-05-18 12:46:51	2
2020-05-18 12:46:51	
2020-05-18 12:46:51	4
2020-05-18 12:46:51	5
2020-05-18 12:46:51	6
2020-05-18 12:46:51	7

When `null` is set to `true`, the `head` command continues to process the results. In this example the command processes the results, ignoring NULL values, as long as the count is less than 5. Because `keeplast=true` the event that stopped the processing, count 5, is also included in the output.

```
| makeresults count=7 | streamstats count | eval count=if(count=3,null(), count) | head count<5  
keeplast=true null=true
```

The results look something like this:

_time	count
2020-05-18 12:46:51	1
2020-05-18 12:46:51	2
2020-05-18 12:46:51	
2020-05-18 12:46:51	4
2020-05-18 12:46:51	5

2020-05-18 12:46:51	1
2020-05-18 12:46:51	2
2020-05-18 12:46:51	
2020-05-18 12:46:51	4
2020-05-18 12:46:51	5

When `null` is set to `false`, the `head` command stops processing the results when it encounters a NULL value. The events with count 1 and 2 are returned. Because `keeplast=true` the event with the NULL value that stopped the processing, the third event, is also included in the output.

```
| makeresults count=7 | streamstats count | eval count=if(count=3,null(), count) | head count<5  
keeplast=true null=false
```

The results look something like this:

_time	count
2020-05-18 12:46:51	1
2020-05-18 12:46:51	2
2020-05-18 12:46:51	

See also

Commands

[reverse](#)
[tail](#)

highlight

Description

Highlights specified terms in the events list. Matches a string or list of strings and highlights them in the display in **Splunk Web**. The matching is not case sensitive.

Syntax

`highlight <string>...`

Required arguments

<string>

Syntax: <string> ...

Description: A space-separated list of strings to highlight in the results. The list you specify is not case-sensitive. Any combination of uppercase and lowercase letters that match the string are highlighted.

Usage

The `highlight` command is a distributable streaming command. See [Command types](#).

The string that you specify must be a field value. The string cannot be a field name.

You must use the `highlight` command in a search that keeps the raw events and displays output on the Events tab. You cannot use the `highlight` command with commands, such as `stats` which produce calculated or generated results.

Examples

Example 1:

Highlight the terms "login" and "logout".

```
... | highlight login,logout
```

Example 2:

Highlight the phrase "Access Denied".

```
... | highlight "access denied"
```

See also

[rangemap](#)

history

Description

Use this command to view your search history in the current application. This search history is presented as a set of events or as a table.

Syntax

```
| history [events=<bool>]
```

Required arguments

None.

Optional arguments

events

Syntax: `events=<bool>`

Description: When you specify `events=true`, the search history is returned as events. This invokes the event-oriented UI which allows for convenient highlighting, or field-inspection. When you specify `events=false`, the search history is returned in a table format for more convenient aggregate viewing.

Default: false

Fields returned when `events=false`.

Output field	Description
<code>_time</code>	The time that the search was started.
<code>api_et</code>	The earliest time of the API call, which is the earliest time for which events were requested.
<code>api_lt</code>	The latest time of the API call, which is the latest time for which events were requested.
<code>event_count</code>	If the search retrieved or generated events, the count of events returned with the search.
<code>exec_time</code>	The execution time of the search in integer quantity of seconds into the Unix epoch.
<code>is_realtime</code>	Indicates whether the search was real-time (1) or historical (0).
<code>result_count</code>	If the search is a transforming search, the count of results for the search.
<code>scan_count</code>	The number of events retrieved from a Splunk index at a low level.
<code>search</code>	The search string.
<code>search_et</code>	The earliest time set for the search to run.
<code>search_lt</code>	The latest time set for the search to run.
<code>sid</code>	The search job ID.
<code>splunk_server</code>	The host name of the machine where the search was run.
<code>status</code>	The status of the search.
<code>total_run_time</code>	The total time it took to run the search in seconds.

Usage

The `history` command is a generating command and should be the first command in the search. Generating commands use a leading pipe character.

The `history` command returns your search history only from the application where you run the command.

Examples

Return search history in a table

Return a table of the search history. You do not have to specify `events=false`, since that is the default setting.

```
| history
```

Splunk > App: Search & Reporting >

Search Datasets Reports Alerts Dashboards

Search & Reporting

New Search

1 | history

Last 24 hours

63 results (1/22/17 2:00:00:000 PM to 1/23/17 2:48:13.000 PM) No Event Sampling

Events Patterns Statistics (63) Visualization

20 Per Page

_time	api_et	api_lt	event_count	exec_time	is_realtime	result_count	savedsearch_name	scan_count	search
2017-01-23 14:48:04.358						1			metadata type search totalCount
2017-01-23 14:47:57.620						0			search sourcetype=a head 5 sort streamstats s ASimpleSumf clientip table bytes, ASimpl
2017-01-23 14:47:44.582			0	1485211664		0	333		0 history sea history* AND search="*net search="*load savedsearch, NOT search=' search="*fron search="*eve summarize=f index=_**" d head 100000

Return search history as events

Return the search history as a set of events.

```
| history events=true
```

Splunk > App: Search & Reporting >

Search Datasets Reports Alerts Dashboards

Search & Reporting

New Search

1 | history events=true

Last 24 hours

51 events (1/22/17 2:00:00:000 PM to 1/23/17 2:44:52,000 PM) No Event Sampling

Events Patterns Statistics Visualization

List 50 Per Page 1

#	Time	Event
>	1/23/17 2:34:01.232 PM	history
>	1/23/17 2:35:55.782 PM	history events=1
>	1/23/17 2:33:49.398 PM	history events=0
>	1/23/17 2:33:40.243 PM	metadata type=sourcetypes search totalCount > 0
>	1/23/17 2:33:34.717 PM	search sourcetype=access_* status=200 chart count AS views count(eval(action="addtocart")) AS addtocart count(eval(action="purchase")) AS purchases by productName rename productName AS "Product Name", views AS "Views", addtocart AS "Adds to Cart", purchases AS "Purchases"
>	1/23/17 2:33:22.325 PM	history search NOT search=" history*" AND NOT search="*metadata*" AND NOT search="*loadjob*" AND NOT savedsearch_name="" AND NOT search="search" AND NOT search="*from sid*" AND NOT search=" eventcount summarize=false index=_**" dedup search head 1 00000
>	1/23/17 2:33:19.884 PM	metadata type=sourcetypes search totalCount > 0

See also

Commands

[search](#)

iconify

Description

Causes Splunk Web to display an icon for each different value in the list of fields that you specify.

The `iconify` command adds a field named `_icon` to each event. This field is the hash value for the event. Within Splunk Web, a different icon for each unique value in the field is displayed in the events list. If multiple fields are listed, the UI displays a different icon for each unique combination of the field values.

Syntax

`iconify <field-list>`

Required arguments

`field-list`

Syntax: `<field>...`

Description: Comma or space-delimited list of fields. You cannot specify a wildcard character in the field list.

Usage

The `iconify` command is a distributable streaming command. See [Command types](#).

Examples

1. Display a different icon for each eventtype

```
... | iconify eventtype
```

2. Display a different icon for unique pairs of field values

Display a different icon for unique pair of `clientip` and `method` values.

```
... | iconify clientip method
```

Here is how Splunk Web displays the results in your **Events List**:

```

7 5/30/10 10.31.10.21 - - [30/May/2010:23:55:27] "GET / HTTP/1.1" 200 9619 "-" "Feedfetcher-Google;
(+http://www.google.com/feedfetcher.html; feed-id=18085321451258128740)"
"10.31.10.21.1269586527948788"
clientip=10.31.10.21 | method=GET

8 5/30/10 10.135.143.166 - - [30/May/2010:23:55:01] "GET /view/SP-CAAACGY HTTP/1.1" 200 11729
"http://5207724e63.example.org/" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.2.2) Gecko/20100316 YFF35 Firefox/3.6.2" "216.145.54.158.1265888041846647"
clientip=10.135.143.166 | method=GET

9 5/30/10 10.135.143.166 - - [30/May/2010:23:54:20] "GET /download HTTP/1.1" 200 7501 "-" "Mozilla/5.0
(Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.2.2) Gecko/20100316 YFF35 Firefox/3.6.2"
"216.145.54.158.1265888041846647"
clientip=10.135.143.166 | method=GET

```

See also

[highlight](#)

inputcsv

Description

For Splunk Enterprise deployments, loads search results from the specified .csv file, which is not modified. The filename must refer to a relative path in \$SPLUNK_HOME/var/run/splunk/csv. If `dispatch=true`, the path must be in \$SPLUNK_HOME/var/run/splunk/dispatch/<job id>.

If the specified file does not exist and the filename does not have an extension, then the Splunk software assumes it has a filename with a `.csv` extension.

If you run into an issue with the `inputcsv` command resulting in an error, ensure that your CSV file ends with a BLANK LINE.

Syntax

The required syntax is in **bold**.

```

| inputcsv
[dispatch=<bool>]
[append=<bool>]
[strict=<bool>]
[start=<int>]
[max=<int>]
[events=<bool>]
<filename>
[WHERE <search-query>]

```

Required arguments

`filename`

Syntax: <filename>

Description: Specify the name of the .csv file, located in \$SPLUNK_HOME/var/run/splunk/csv.

Optional arguments

dispatch

Syntax: dispatch=<bool>

Description: When set to true, this argument indicates that the filename is a .csv file in the dispatch directory.

The relative path is \$SPLUNK_HOME/var/run/splunk/dispatch/<job id>/.

Default: false

append

Syntax: append=<bool>

Description: Specifies whether the data from the .csv file is appended to the current set of results (true) or replaces the current set of results (false).

Default: false

strict

Syntax: strict=<bool>

Description: When set to true this argument forces the search to fail completely if `inputcsv` raises an error. This happens even when the errors apply to a subsearch. When set to false, many `inputcsv` error conditions return warning messages but do not otherwise cause the search to fail. Certain error conditions cause the search to fail even when `strict=false`.

Default: false

events

Syntax: events=<bool>

Description: Specifies whether the data in the CSV file are treated as events or as a table of search results. By default `events=false` returns the data in a table with field names as column headings. The table appears on the Statistics tab. If you set `events=true`, the imported CSV data must have the `_time` and `_raw` fields. The data is treated as events, which appear on the Events tab.

Default: false

max

Syntax: max=<int>

Description: Controls the maximum number of events to be read from the file. If `max` is not specified, there is no limit to the number of events that can be read.

Default: 1000000000 (1 billion)

start

Syntax: start=<int>

Description: Controls the 0-based offset of the first event to be read.

Default: 0

WHERE

Syntax: WHERE <search-criteria>

Description: Use this clause to improve search performance by prefiltering data returned from the CSV file. Supports a limited set of search query operators: =, !=, <, >, <=, >=, AND, OR, NOT. Any combination of these operators is permitted. Also supports wildcard string searches.

Usage

The `inputcsv` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Appending or replacing results

If the `append` argument is set to `true`, you can use the `inputcsv` command to append the data from the CSV file to the current set of search results. With `append=true`, you use the `inputcsv` command later in your search, after the search has returned a set of results. See [Examples](#).

The `append` argument is set to `false` by default. If the `append` argument is not specified or is set to `false`, the `inputcsv` command must be the first command in the search. Data is loaded from the specified CSV file into the search.

Working with large CSV files

The `WHERE` clause allows you to narrow the scope of the search of the `inputcsv` file. It restricts the `inputcsv` to a smaller number of rows, which can improve search efficiency when you are working with significantly large CSV files.

Distributed deployments

The `inputcsv` command is not compatible with **search head pooling** and **search head clustering**.

The command saves the `*.csv` file on the local search head in the `$SPLUNK_HOME/var/run/splunk/` directory. The `*.csv` files are not replicated on the other search heads.

Strict error handling

Use the `strict` argument to make `inputcsv` searches fail whenever they encounter an error condition. You can set this at the system level for all `inputcsv` and `inputlookup` searches by changing `input_errors_fatal` in `limits.conf`

If you use Splunk Cloud Platform, file a Support ticket to change the `input_errors_fatal` setting.

Use the `strict` argument to override the `input_errors_fatal` setting for an `inputcsv` search.

Examples

1. Load results that contain a specific string

This example loads search results from the `$SPLUNK_HOME/var/run/splunk/csv/all.csv` file. Those that contain the string `error` are saved to the `$SPLUNK_HOME/var/run/splunk/csv/error.csv` file.

```
| inputcsv all.csv | search error | outputcsv errors.csv
```

2. Load a specific range of results

This example loads results 101 to 600 from either the `bar` file, if exists, or from the `bar.csv` file.

```
| inputcsv start=100 max=500 bar
```

3. Specifying which results to load with operators and expressions

You can use comparison operators and Boolean expression to specify which results to load. This example loads all of the events from the CSV file `$SPLUNK_HOME/var/run/splunk/csv/students.csv` and then filters out the events that do not

match the WHERE clause, where the values in the `age` field are greater than 13, less than 19, but not 16. The search returns a count of the remaining search results.

```
| inputcsv students.csv WHERE (age>=13 age<=19) AND NOT age=16 | stats count
```

4. Append data from a CSV file to search results

You can use the `append` argument to append data from a CSV file to a set of search results. In this example the combined data is then output back to the same CSV file.

```
error earliest=-d@d | inputcsv append=true all_errors.csv | outputcsv all_errors.csv
```

5. Appending multiple CSV files

You can also append the search results of one CSV file to another CSV file by using the `append` command and a subsearch. This example uses the `eval` command to add a field to each set of data to denote which CSV file the data originated from.

```
| inputcsv file1.csv | eval source="file1" | append [inputcsv file2.csv | eval source="file2"]
```

See also

[outputcsv](#)

inputintelligence

The `inputintelligence` command is used with Splunk Enterprise Security.

For information about this command, see Use generic intelligence in search with `inputintelligence` in *Administer Splunk Enterprise Security*.

inputlookup

Description

Use the `inputlookup` command to search the contents of a lookup table. The lookup table can be a CSV lookup or a KV store lookup.

Syntax

The required syntax is in **bold**.

```
| inputlookup
[append=<bool>]
[strict=<bool>]
[start=<int>]
[max=<int>]
<filename> | <tablename>
[WHERE <search-query>]
```

Required arguments

You must specify either a <filename> or a <tablename>.

<filename>

Syntax: <string>

Description: The name of the lookup file must end with `.csv` or `.csv.gz`. If the lookup does not exist, a warning message is displayed (but no syntax error is generated).

<tablename>

Syntax: <string>

Description: The name of the lookup table as specified by a stanza name in the `transforms.conf` file. The lookup table can be configured for any lookup type (CSV, external, or KV store).

Optional arguments

append

Syntax: append=<bool>

Description: If set to `true`, the data returned from the lookup file is appended to the current set of results rather than replacing it. Defaults to `false`.

strict

Syntax: strict=<bool>

Description: When set to `true` this argument forces the search to fail completely if `inputlookup` raises an error. This happens even when the errors apply to a subsearch. When set to `false`, many `inputlookup` error conditions return warning messages but do not otherwise cause the search to fail. Certain error conditions cause the search to fail even when `strict=false`.

Default: false

max

Syntax max=<int>

Description: Specify the maximum number of events to be read from the file. Defaults to 1000000000.

start

Syntax: start=<int>

Description: Specify the 0-based offset of the first event to read. If `start=0`, it begins with the first event. If `start=4`, it begins with the fifth event. Defaults to 0.

WHERE clause

Syntax: WHERE <search-query>

Description: Use this clause to improve search performance by prefiltering data returned from the lookup table. Supports a limited set of search query operators: `=`, `!=`, `<`, `>`, `<=`, `>=`, AND, OR, NOT. Any combination of these operators is permitted. Also supports wildcard string searches.

Usage

The `inputlookup` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search. The `inputlookup` command can be first command in a search or in a subsearch.

The lookup can be a file name that ends with `.csv` or `.csv.gz`, or a lookup table definition in **Settings > Lookups > Lookup definitions**.

Appending or replacing results

When using the `inputlookup` command in a subsearch, if `append=true`, data from the lookup file or KV store collection is appended to the search results from the main search. When `append=false` the main search results are replaced with the results from the lookup search.

Working with large CSV lookup tables

The WHERE clause allows you to narrow the scope of the query that `inputlookup` makes against the lookup table. It restricts `inputlookup` to a smaller number of lookup table rows, which can improve search efficiency when you are working with significantly large lookup tables.

Testing geometric lookup files

You can use the `inputlookup` command to verify that the geometric features on the map are correct. The syntax is `| inputlookup <your_lookup>`.

1. For example, to verify that the geometric features in built-in `geo_us_states` lookup appear correctly on the choropleth map, run the following search:

```
| inputlookup geo_us_states
```

2. On the **Visualizations** tab, zoom in to see the geometric features. In this example, the states in the United States.

Strict error handling

Use the `strict` argument to make `inputlookup` searches fail whenever they encounter an error condition. You can set this at the system level for all `inputcsv` and `inputlookup` searches by changing `input_errors_fatal` in `limits.conf`.

If you use Splunk Cloud Platform, file a Support ticket to change the `input_errors_fatal` setting.

Use the `strict` argument to override the `input_errors_fatal` setting for an `inputlookup` search.

Additional information

For more information about creating lookups, see *About lookups* in the *Knowledge Manager Manual*.

For more information about the App Key Value store, see *About KV store* in the *Admin Manual*.

Examples

1. Read in a lookup table

Read in a `usertogroup` lookup table that is defined in the `transforms.conf` file.

```
| inputlookup usertogroup
```

2. Append lookup table fields to the current search results

Using a subsearch, read in the `usertogroup` lookup table that is defined by a stanza in the `transforms.conf` file. Append the fields to the results in the main search.

```
... [| inputlookup append=t usertogroup]
```

3. Read in a lookup table in a CSV file

Search the `users.csv` lookup file, which is in the `$SPLUNK_HOME/etc/system/lookups` or `$SPLUNK_HOME/etc/apps/<app_name>/lookups` directory.

```
| inputlookup users.csv
```

4. Read in a lookup table from a KV store collection

Search the contents of the KV store collection `kvstorecoll` that have a `CustID` value greater than 500 and a `CustName` value that begins with the letter P. The collection is referenced in a lookup table called `kvstorecoll_lookup`. Provide a count of the events received from the table.

```
| inputlookup kvstorecoll_lookup where (CustID>500) AND (CustName="P*") | stats count
```

In this example, the lookup definition explicitly defines the `CustID` field as a type of "number". If the field type is not explicitly defined, the `where` clause does not work. Defining field types is optional.

5. View the internal key ID values for the KV store collection

Example 5: View internal key ID values for the KV store collection `kvstorecoll`, using the lookup table `kvstorecoll_lookup`. The internal key ID is a unique identifier for each record in the collection. This example uses the `eval` and `table` commands.

```
| inputlookup kvstorecoll_lookup | eval CustKey = _key | table CustKey, CustName, CustStreet, CustCity, CustState, CustZip
```

6. Update field values for a single KV store collection record

Update field values for a single KV store collection record. This example uses the `inputlookup`, `outputlookup`, and `eval` commands. The record is indicated by its internal key ID (the `_key` field) and this search updates the record with a new customer name and customer city. The record belongs to the KV store collection `kvstorecoll`, which is accessed through the lookup table `kvstorecoll_lookup`.

```
| inputlookup kvstorecoll_lookup | search _key=544948df3ec32d7a4c1d9755 | eval CustName="Claudia Garcia" | eval CustCity="San Francisco" | outputlookup kvstorecoll_lookup append=true key_field=_key
```

7. Write the contents of a CSV file to a KV store collection

Write the contents of a CSV file to the KV store collection `kvstorecoll` using the lookup table `kvstorecoll_lookup`. The CSV file is in the `$SPLUNK_HOME/etc/system/lookups` or `$SPLUNK_HOME/etc/apps/<app_name>/lookups` directory.

```
| inputlookup customers.csv | outputlookup kvstorecoll_lookup
```

See also

Commands

[inputcsv](#)
[join](#)
[lookup](#)
[outputlookup](#)

iplocation

Description

The `iplocation` command extracts location information from IP addresses by using 3rd-party databases. This command supports IPv4 and IPv6 addresses and subnets that use CIDR notation.

The IP address that you specify in the `ip-address-fieldname` argument, is looked up in a database. Fields from that database that contain location information are added to each event. The setting of the `allfields` argument determines which fields are added to the events.

Because all the information might not be available for each IP address, an event can have empty field values.

For IP addresses which do not have a location, such as internal addresses, no fields are added.

Syntax

The required syntax is in **bold**.

```
iplocation
[prefix=<string>]
[allfields=<bool>]
[lang=<string>]
<ip-address-fieldname>
```

Required arguments

`ip-address-fieldname`

Syntax: <field>

Description: Specify an IP address field, such as `clientip`.

Optional arguments

`allfields`

Syntax: `allfields=<bool>`

Description: Specifies whether to add all of the fields from the database to the search results. If set to `true`, this argument adds the fields `City`, `Continent`, `Country`, `MetroCode`, `Region`, `Timezone`, `_time`, `lat` (`latitude`), and `lon` (`longitude`).

Default: `false`. Only the `City`, `Country`, `Region`, `_time`, `lat`, and `lon` fields are added to the search results.

`lang`

Syntax: `lang=<string>`

Description: Render the resulting strings in different languages. For example, use `lang=es` for Spanish. The set of languages depends on the geoip database that is used. To specify more than one language, separate them with a comma. This also indicates the priority in descending order. Specify `lang=code` to return the fields as two letter ISO abbreviations.

prefix

Syntax: prefix=<string>

Description: Specify a string to prefix the field name. With this argument you can add a prefix to the added field names to avoid name collisions with existing fields. For example, if you specify `prefix=iploc_` the field names that are added to the events become `iploc_City`, `iploc_County`, `iploc_lat`, and so forth.

Default: NULL/empty string

Usage

The `iplocation` command is a distributable streaming command. See [Command types](#).

The Splunk software ships with a copy of the `dbip-city-lite.mmdb` IP geolocation database file. This file is located in the `$SPLUNK_HOME/share/` directory.

Updating the IP geolocation database file

Through Splunk Web, you can update the `.mmdb` file that ships with the Splunk software. The file you update it with can be a copy of one of the following two files. Only those two files are supported. To use these two files, you must have a license for the GeoIP2 City database.

File name	Description
<code>GeoLite2-City.mmdb</code>	This is a free IP geolocation database that is updated on its download page on a weekly basis.
<code>GeoIP2-City.mmdb</code>	This is a paid version of the GeoLite2-City IP geolocation database that is more accurate than the free version.

Replacing your `mmdb` file with one of these two files reintroduces the `Timezone` field that is absent in the default `.mmdb` file, but does not reintroduce the `MetroCode` field.

Prerequisites

You must have a **role** with the `upload_mmdb_files` **capability**.

Steps

1. Go online and find a download page for the binary `.tar.gz` versions of the `GeoLite2-City` or the `GeoIP2-City` database files.
2. Download the binary `.tar.gz` version of the file (`GeoLite2-City` or `GeoIP2-City`) that is most appropriate for your needs.
3. Expand the binary `.tar.gz` version of the file.
The `.tar.gz` file expands into a folder which contains the `GeoLite2-City.mmdb` file, or the `GeoIP2-City.mmdb` file, depending on the download you selected.
4. In Splunk Web, go to **Settings > Lookups > GeoIP lookups file**.
5. On the GeoIP lookups file page, click **Choose file**. Select the `.mmdb` file.
6. Click **Save**.

The page displays a success message when the upload completes.

An .mmdb file that you upload through this method is treated as a lookup table by the Splunk software. This means it is picked up by knowledge bundle replication in distributed search environments, but that also means it can increase the size of knowledge bundles. See Knowledge bundle replication overview in the *Distributed Search* manual.

If you upload your own .mmdb file in Splunk Web and later decide you want to revert back to the .mmdb file that was shipped with the Splunk software, go to Settings > Lookups > GeolP lookups file and delete your uploaded file.

Impact of upgrading Splunk software

When you upgrade your Splunk platform, the .mmdb file in the \$SPLUNK_HOME/share/ directory is replaced by the version of the file that ships with the Splunk software. You can avoid this by storing the .mmdb file in a different file path.

Storing the .mmdb file in a different file path

To store the GeoLite2-City.mmdb or GeoIP2-City.mmdb file in a different file path you must update the path directly in the `limits.conf` file. This is not possible in Splunk Cloud Platform, only Splunk Enterprise.

The Splunk Web .mmdb file upload feature takes precedence over manual updates to the .mmdb file path in `limits.conf`.

Prerequisites

- Only users with file system access, such as system administrators, can specify a different file path to the .mmdb file in the `limits.conf` file.
- Review the steps in How to edit a configuration file in the *Admin Manual*.
- You can have configuration files with the same name in your default, local, and app directories. Read Where you can place (or find) your modified configuration files in the *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make the changes in the local directory.

Steps

1. Open the local `limits.conf` file for the Search app. For example, `$SPLUNK_HOME/etc/apps/search/local`.
2. Add the `[iplocation]` stanza.
3. Add the `db_path` setting and specify the absolute path to the .mmdb file. The `db_path` setting does not support standard Splunk environment variables such as `$SPLUNK_HOME`.
For example: `db_path = /Applications/Splunk/mmdb/GeoLite2-City.mmdb` specifies a new directory called `mmdb`.
4. Ensure a copy of the .mmdb file is stored in the `../Applications/Splunk/mmdb/` directory.
5. Because you are editing the path to the .mmdb file, you should restart the Splunk server.

Storing the .mmdb file with a different name

Alternatively, you can add the updated .mmdb to the \$SPLUNK_HOME/share/ directory using a different name and then specify that name in the `db_path` setting. For example: `db_path = /Applications/Splunk/share/GeoLite2-City_paid.mmdb`.

The .mmdb file and distributed deployments

The `iplocation` command is a distributable **streaming command**, which means that it can be processed on the indexers. The `$SPLUNK_HOME/share/` directory is not part of the **knowledge bundle**. If you update the `.mmdb` file in the `$SPLUNK_HOME/share/` directory, the updated file is not automatically sent to the indexers in a distributed deployment. To add the `.mmdb` file to the indexers, use the tools that you typically use to push files to the indexers.

Examples

1. Add location information to web access events

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 **All time** when you run the search.

Add location information to web access events. By default, the `iplocation` command adds the `City`, `Country`, `lat`, `lon`, and `Region` fields to the results.

```
sourcetype=access_* | iplocation clientip
```

2. Search for client errors and return the first 20 results

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 **All time** when you run the search.

Search for client errors in web access events, returning only the first 20 results. Add location information and return a table with the IP address, City, and Country for each client error.

```
sourcetype=access_* status>=400 | head 20 | iplocation clientip | table clientip, status, City, Country
```

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

clientip	status	City	Country
182.236.164.11	408	Zhengzhou	China
198.35.1.75	500	Princeton	United States
198.35.1.75	404	Princeton	United States
198.35.1.75	406	Princeton	United States
198.35.1.75	500	Princeton	United States
221.204.246.72	503	Taiyuan	China
1.192.86.205	503	Amesbury	United States
91.205.189.15	406		
216.221.226.11	505	Redwood City	United States
216.221.226.11	404	Redwood City	United States
195.2.240.99	400		Russia

3. Add a prefix to the fields added by the iplocation command

Prefix the fields added by the `iplocation` command with `iploc_`. Add all of the fields in the `.mmdb` database file to the results.

```
sourcetype = access_* | iplocation prefix=iploc_ allfields=true clientip | fields iploc_*
```

4. Generate a choropleth map using IP addresses

Generate a choropleth map of your data like the one below using the `iplocation` command. See [Use IP addresses to generate a choropleth map in Dashboards and Visualizations](#).



5. Identify IPv6 address locations

The `iplocation` command supports IPv6 lookup through IP geolocation functionality. In the following example, `iplocation` looks up the specified IP address in the default geolocation database file to determine where it is located.

```
| makeresults | eval myip="2001:4860:4860::8888" | iplocation myip
```

Search finds the location of the IP address and displays the following results.

City	Country	Region	_time	lat	lon	myip
	United States		2021-11-22 13:37:07	37.75100	-97.82200	2001:4860:4860::8888

See also

Commands

[lookup](#)
[search](#)

Functions

[cidrmatch](#)

join

Description

You can use the `join` command to combine the results of a main search (left-side dataset) with the results of either another dataset or a subsearch (right-side dataset). You can also combine a search result set to itself using the `selfjoin` command.

The left-side dataset is the set of results from a search that is piped into the `join` command and then merged on the right side with either a dataset or the results from a subsearch. The left-side dataset is sometimes referred to as the source data.

The following search example joins the source data from the search pipeline with a subsearch on the right side. Rows from each dataset are merged into a single row if the `where` predicate is satisfied.

```
<left-dataset>
| join left=L right=R where L.pid = R.pid [subsearch]
```

A maximum of 50,000 rows in the right-side dataset can be joined with the left-side dataset. This maximum default is set to limit the impact of the `join` command on performance and resource consumption.

If you are familiar with SQL but new to SPL, see [Splunk SPL for SQL users](#).

Alternative commands

For flexibility and performance, consider using one of the following commands if you do not require join semantics. These commands provide event grouping and correlations using time and geographic location, transactions, subsearches, field lookups, and joins.

Command	Use
<code>append</code>	To append the results of a subsearch to the results of your current search. The events from both result sets are retained. <ul style="list-style-type: none">• Use only with historical data. The <code>append</code> command does not produce correct results if used in a real-time search.• If you use <code>append</code> to combine the events, use a <code>stats</code> command to group the events in a meaningful way. You cannot use a <code>transaction</code> command after you use an <code>append</code> command.
<code>appendcols</code>	Appends the fields of the subsearch results with the input search result fields. The first subsearch result is merged with the first main result, the second subsearch result is merged with the second main result, and so on.
<code>lookup</code>	Use when one of the result sets or source files remains static or rarely changes. For example, a file from an external system such as a CSV file. The <code>lookup</code> cannot be a subsearch.
<code>search</code>	In the most simple scenarios, you might need to search only for sources using the OR operator and then use a <code>stats</code> or <code>transaction</code> command to perform the grouping operation on the events.
<code>stats</code>	To group events by a field and perform a statistical function on the events. For example to determine the average duration of events by host name. <ul style="list-style-type: none">• To use <code>stats</code>, the field must have a unique identifier.• To view the raw event data, use the <code>transaction</code> command instead.

Command	Use
<code>transaction</code>	<p>Use <code>transaction</code> in the following situations.</p> <ul style="list-style-type: none"> • To group events by using the <code>eval</code> command with a conditional expression, such as <code>if</code>, <code>case</code>, or <code>match</code>. • To group events by using a recycled field value, such as an ID or IP address. • To group events by using a pattern, such as a start or end time for the event. • To break up groups larger than a certain duration. For example, when a transaction does not explicitly end with a message and you want to specify a maximum span of time after the start of the transaction. • To display the raw event data for the grouped events.

For information about when to use a join, see the flowchart in About event grouping and correlation in the *Search Manual*.

Syntax

The required syntax is in **bold**.

```
join
[<join-options>...]
[<field-list>] | [left=<left-alias>] [right=<right-alias>] where <left-alias>.<field>=<right-alias>.<field>
[<left-alias>.<field>=<right-alias>.<field>]...
<dataset-type>:<dataset-name> | <subsearch>
```

Required arguments

dataset-type

Syntax: `datamodel` | `savedsearch` | `inputlookup`

Description: The type of dataset that you want to use to join with the source data. The dataset must be a dataset that you created or are authorized to use. You can specify `datamodel`, `savedsearch`, or `inputlookup`. The dataset type must precede the dataset name. For example, `savedsearch:<dataset-name>`.

You can use either `<dataset-type>:<dataset-name>` or `<subsearch>` with the `join` command, but not both.

dataset-name

Syntax: `<dataset-name>`

Description: The name of the dataset that you want to use to join with the source data. The dataset must be a dataset that you created or are authorized to use. The dataset name must follow the dataset type. For example, if the dataset name is `january` and the dataset type is `datamodel`, you specify `datamodel:january`.

You can use either `<dataset-type>:<dataset-name>` or `<subsearch>` with the `join` command, but not both.

subsearch

Syntax: [<subsearch>]

Description: A secondary search or dataset that specifies the source of the events that you want to join to. The subsearch must be enclosed in square brackets. The results of the subsearch should not exceed available memory.

You can use either `<dataset-type>:<dataset-name>` or `<subsearch>` in a search, but not both. When `[<subsearch>]` is used in a search by itself with no join keys, the Splunk software autodetects common fields and combines the search results before the `join` command with the results of the subsearch.

Optional arguments

join-options

Syntax: type=(inner | outer | left) | usetime=<bool> | earlier=<bool> | overwrite=<bool> | max=<int>

Description: Arguments to the join command. Use either `outer` or `left` to specify a left outer join. See [Descriptions for the join-options argument](#) in this topic.

field-list

Syntax: <field> <field> ...

Description: Specify the list of fields to use for the join. For example, to join fields `ProductA`, `ProductB`, and `ProductC`, you would specify `| join ProductA ProductB ProductC....`. If <field-list> is specified, one or more of the fields must be common to each dataset. If no fields are specified, all of the fields that are common to both datasets are used.

left alias

Syntax: left=<left-alias>

Description: The alias to use with the left-side dataset, the source data, to avoid naming collisions. Must be combined with the right alias and `where` clause, or the alias is ignored.

The left alias must be used together with the right alias.

right alias

Syntax: right=<right-alias>

Description: The alias to use with the right-side dataset to avoid naming collisions. Must be combined with the left alias and the `where` clause, or the alias is ignored.

The right alias must be used together with the left alias.

where clause

Syntax: where <left-alias>.<field>=<right-alias>.<field>...

Description: Identifies the names of the fields in the left-side dataset and the right-side dataset that you want to join on. You must specify the left and right aliases and the field name. Fields that are joined from the left and right datasets do not have to have the same names. For example: `where L.host=R.user` matches events in the `host` field from the left dataset with events in the `user` field from the right dataset.

The `where` clause must be used with the right and left aliases and field name.

You can specify the aliases and fields in a `where` clause on either side of the equal sign. For example:

```
where <left-alias>.<left-field>=<right-alias>.<right-field>
```

or

```
where <right-alias>.<right-field>=<left-alias>.<left-field>
```

Descriptions for the join-options argument

type

Syntax: type=inner | outer | left

Description: Indicates the type of join to perform. The difference between an `inner` and a `left` (or `outer`) join is how the events are treated in the main search that do not match any of the events in the subsearch. In both `inner` and `left` joins, events that match are joined. The results of an `inner` join do not include events from the main

search that have no matches in the subsearch. The results of a `left` (or `outer`) join includes all of the events in the main search and only those values in the subsearch have matching field values.

Default: `inner`

`usetime`

Syntax: `usetime=<bool>`

Description: A Boolean value that indicates whether to use time to limit the matches in the subsearch results. Used with the `earlier` option to limit the subsearch results to matches that are earlier or later than the main search results.

If you use the `join` command with `usetime=true` and `type=left`, the search results are similar to those of an inner join. This is because there might be non-matching results when using the left join that are the same as those produced by an inner join.

Default: `false`

`earlier`

Syntax: `earlier=<bool>`

Description: If `usetime=true` and `earlier=true`, the main search results are matched only against earlier results from the subsearch. If `earlier=false`, the main search results are matched only against later results from the subsearch. Results that occur at the same time (second) are not eliminated by either value.

Default: `true`

`overwrite`

Syntax: `overwrite=<bool>`

Description: If fields in the main search results and subsearch results have the same name, indicates whether fields from the subsearch results overwrite the fields from the main search results.

Default: `true`

`max`

Syntax: `max=<int>`

Description: Specifies the maximum number of subsearch results that each main search result can join with. If set to `max=0`, there is no limit.

Default: `1`

Usage

The `join` command is a centralized streaming command when there is a defined set of fields to join to. Otherwise the command is a dataset processing command. See [Command types](#).

A subsearch can be initiated through a search command such as the `join` command. See [Initiating subsearches with search commands](#) in the Splunk Cloud Platform *Search Manual*.

Limitations on subsearches in joins

Use the `join` command when the results of the subsearch are relatively small, for example 50,000 rows or less. To minimize the impact of this command on performance and resource consumption, Splunk software imposes some default limitations on the subsearch.

Limitations on the subsearch for the `join` command are specified in the `limits.conf` file. The limitations include a maximum of 50,000 rows in the subsearch to join against and the maximum search time for the subsearch. See [Subsearches](#) in the *Search Manual*.

Splunk Cloud Platform

To change the limits.conf settings `subsearch_maxout` or `subsearch_maxtime`, use one of the following methods:

- ◊ The Configure limits page in Splunk Web. For more information, see Configure limits using Splunk Web in the *Splunk Cloud Platform Admin Manual*.
- ◊ The Admin Config Service (ACS) API. For more information, see Manage limits.conf configurations in Splunk Cloud Platform in the Splunk Cloud Platform *Admin Config Service Manual*.
- ◊ The Admin Config Service (ACS) command line interface (CLI). For more information, see Administer Splunk Cloud Platform using the ACS CLI in the Splunk Cloud Platform *Admin Config Service Manual*.

Splunk Enterprise

To change the `subsearch_maxout` or `subsearch_maxtime` settings in your limits.conf file for `join` command subsearches, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in How to edit a configuration file in the Splunk Enterprise *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local limits.conf file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[join]` stanza, add the line `subsearch_maxout = <value>` or `subsearch_maxtime = <value>`.

One-to-many and many-to-many relationships

To return matches for one-to-many, many-to-one, or many-to-many relationships, include the `max` argument in your `join` syntax and set the value to 0. By default `max=1`, which means that the subsearch returns only the first result from the subsearch. Setting the value to a higher number or to 0, which is unlimited, returns multiple results from the subsearch.

Basic examples

1. A basic join

Combine the results from a main search with the results from a subsearch `search vendors`. The result sets are joined on the `product_id` field, which is common to both sources.

```
... | join product_id [search vendors]
```

2. Returning all subsearch rows

By default, only the first row of the subsearch that matches a row of the main search is returned. To return all of the matching subsearch rows, include the `max=<int>` argument and set the value to 0. This argument joins each matching subsearch row with the corresponding main search row.

```
... | join product_id max=0 [search vendors]
```

3. Join datasets on fields that have the same name

Combine the results from a search with the `vendors` dataset. The data is joined on the `product_id` field, which is common to both datasets.

```
... | join left=L right=R where L.product_id=R.product_id [search vendors]
```

4. Join datasets on fields that have different names

Combine the results from a search with the `vendors` dataset. The data is joined on a product ID field, which have different field names in each dataset. The field in the left-side dataset is `product_id`. The field in the right-side dataset is `pid`.

```
... | join left=L right=R where L.product_id=R.pid [search vendors]
```

4. Use words instead of letters as aliases

You can use words for the aliases to help identify the datasets involved in the join. This example uses `products` and `vendors` for the aliases.

```
... | join left=products right=vendors where products.product_id=vendors.pid [search vendors]
```

Extended examples

1. Specifying dataset aliases with a saved search dataset

This example joins each matching right-side dataset row with the corresponding source data row. This example uses `products`, which is a `savedsearch` type of dataset, for the right-side dataset. The field names in the left-side dataset and the right-side dataset are different. This search returns all of the matching rows in the left and right datasets by including `max=0` in the search.

```
... | join max=0 left=L right=R where L.vendor_id=R.vid savedsearch:products
```

2. Use aliasing with commands following the join

Commands following the join can take advantage of the aliasing provided through the `join` command. For example, you can use the aliasing in another command like `stats` as shown in the following example.

```
... | join left=L right=R where L.product_id=R.pid [search vendors] | stats count by L.product_id
```

3. Using a join to display resource usage information

The dashboards and alerts in the distributed management console shows you performance information about your Splunk deployment. The **Resource Usage: Instance** dashboard contains a table that shows the machine, number of cores, physical memory capacity, operating system, and CPU architecture.

To display the information in the table, use the following search. This search includes the `join` command. The search uses the information in the `dmc_assets` table to look up the instance name and machine name. The search then uses the `serverName` field to join the information with information from the `/services/server/info` REST endpoint. The `/services/server/info` is the URI path to the Splunk REST API endpoint that provides hardware and operating system information for the machine. The `$splunk_server$` part of the search is a dashboard token variable.

```
| inputlookup dmc_assets
```

```

| search serverName = $splunk_server$
| stats first(serverName) AS serverName, first(host) AS host, first(machine) AS machine
| join type=left serverName
  [ | rest splunk_server=$splunk_server$ /services/server/info
    | fields serverName, numberOfCores, physicalMemoryMB, os_name, cpu_arch]
| fields machine AS Machine, numberOfCores AS "Number of Cores",
  physicalMemoryMB AS "Physical Memory Capacity (MB)", os_name AS "Operating System",
  cpu_arch AS "CPU Architecture"

```

See also

[selfjoin](#), [append](#), [set](#), [appendcols](#)

kmeans

Description

Partitions the events into k clusters, with each cluster defined by its mean value. Each event belongs to the cluster with the nearest mean value. Performs k-means clustering on the list of fields that you specify. If no fields are specified, performs the clustering on all numeric fields. Events in the same cluster are moved next to each other. You have the option to display the cluster number for each event.

Syntax

`kmeans [kmeans-options...] [field-list]`

Required arguments

None.

Optional arguments

field-list

Syntax: <field> ...

Description: Specify a space separated list of the exact fields to use for the join.

Default: If no fields are specified, uses all numerical fields that are common to both result sets. Skips events with non-numerical fields.

kmeans-options

Syntax: <reps> | <iters> | <t> | <k> | <cnumfield> | <distype> | <showcentroid>

Description: Options for the `kmeans` command.

kmeans options

reps

Syntax: reps=<int>

Description: Specify the number of times to repeat kmeans using random starting clusters.

Default: 10

iters

Syntax: maxiters=<int>

Description: Specify the maximum number of iterations allowed before failing to converge.

Default: 10000

t

Syntax: t=<num>

Description: Specify the algorithm convergence tolerance.

Default: 0

k

Syntax: k=<int> | <int>-<int>

Description: Specify as a scalar integer value or a range of integers. When provided as single number, selects the number of clusters to use. This produces events annotated by the cluster label. When expressed as a range, clustering is done for each of the cluster counts in the range and a summary of the results is produced. These results express the size of the clusters, and a 'distortion' field which represents how well the data fits those selected clusters. Values must be greater than 1 and less than maxkvalue (see Limits section).

Default: k=2

cnumfield

Syntax: cfield=<field>

Description: Names the field to annotate the results with the cluster number for each event.

Default: CLUSTERNUM

distype

Syntax: dt= (l1 | l1norm | cityblock | cb) | (l2 | l2norm | sq | sgeuclidean) | (cos | cosine)

Description: Specify the distance metric to use. The `l1`, `l1norm`, and `cb` distance metrics are synonyms for `cityblock`. The `l2`, `l2norm`, and `sq` distance metrics are synonyms for `sgeuclidean` or `sqEuclidean`. The `cos` distance metric is a synonym for `cosine`.

Default: sgeuclidean

showcentroid

Syntax: showcentroid= true | false

Description: Specify whether to expose the centroid centers in the search results (showcentroid=true) or not.

Default: true

Usage

Limits

The number of clusters to collect the values into -- k -- is not permitted to exceed maxkvalue. The maxkvalue is specified in the `limits.conf` file, in the `[kmeans]` stanza. The maxkvalue default is 1000.

When a range is given for the `k` option, the total distance between the beginning and ending cluster counts is not permitted to exceed maxrange. The maxrange is specified in the `limits.conf` file, in the `[kmeans]` stanza. The maxrange default is 100.

The above limits are designed to avoid the computation work becoming unreasonably expensive.

The total number of values which are clustered by the algorithm (typically the number of input results) is limited by the `maxdatapoints` parameter in the `[kmeans]` stanza of `limits.conf`. If this limit is exceeded at runtime, a warning message displays in Splunk Web. This defaults to 100000000 or 100 million. This `maxdatapoints` limit is designed to avoid exhausting memory.

Examples

Example 1: Group search results into 4 clusters based on the values of the "date_hour" and "date_minute" fields.

```
... | kmeans k=4 date_hour date_minute
```

Example 2: Group results into 2 clusters based on the values of all numerical fields.

```
... | kmeans
```

See also

[anomalies](#), [anomalousvalue](#), [cluster](#), [outlier](#),

kvform

Description

Extracts key-value pairs from events based on a form template that describes how to extract the values.

For Splunk Cloud Platform, you must create a private app to extract key-value pairs from events. If you are a Splunk Cloud administrator with experience creating private apps, see [Manage private apps in your Splunk Cloud Platform deployment](#) in the [Splunk Cloud Admin Manual](#). If you have not created private apps, contact your Splunk account representative for help with this customization.

Syntax

```
kvform [form=<string>] [field=<field>]
```

Optional arguments

form

Syntax: form=<string>

Description: Specify a .form file located in a \$SPLUNK_HOME/etc/apps/*/forms/ directory.

field

Syntax: field=<field_name>

Description: Uses the field name to look for .form files that correspond to the field values for that field name. For example, your Splunk deployment uses the splunkd and mongod sourcetypes. If you specify field=sourcetype, the kvform command looks for the splunkd.form and mongod.form in the \$SPLUNK_HOME/etc/apps/*/forms/ directory.

Default: sourcetype

Usage

Before you can use the kvform command, you must:

- Create the forms directory in the appropriate application path. For example
\$SPLUNK_HOME/etc/apps/<app_name>/forms.

- Create the `.form` files and add the files to the `forms` directory.

Format for the `.form` files

A `.form` file is essentially a text file of all static parts of a form. It might be interspersed with named references to regular expressions of the type found in the `transforms.conf` file.

An example `.form` file might look like this:

```
Students Name: [[string:student_name]]
Age: [[int:age]] Zip: [[int:zip]]
```

Specifying a form

If the `form` argument is specified, the `kvform` command uses the `<form_name>.form` file found in the Splunk configuration `forms` directory. For example, if `form=sales_order`, the `kvform` command looks for a `sales_order.form` file in the `$SPLUNK_HOME/etc/apps/<app_name>/forms` directory for all apps. All the events processed are matched against the form, trying to extract values.

Specifying a field

If you specify the `field` argument, the `kvform` command looks for forms in the `forms` directory that correspond to the values for that field. For example, if you specify `field=error_code`, and an event has the field value `error_code=404`, the command looks for a form called `404.form` in the `$SPLUNK_HOME/etc/apps/<app_name>/forms` directory.

Default value

If no `form` or `field` argument is specified, the `kvform` command uses the default value for the `field` argument, which is `sourcetype`. The `kvform` command looks for `<sourcetype_value>.form` files to extract values.

Examples

1. Extract values using a specific form

Use a specific form to extract values from.

```
... | kvform form=sales_order
```

2. Extract values using a field name

Specify `field=sourcetype` to extract values from forms such as `splunkd.form` and `mongod.form`. If there is a form for a source type, values are extracted from that form. If one of the source types is `access_combined` but there is no `access_combined.form` file, that source type is ignored.

```
... | kvform field=sourcetype
```

3. Extract values using the `eventtype` field

```
... | kvform field=eventtype
```

See also

Commands

[extract](#)
[multikv](#)
[rex](#)
[xmlkv](#)

loadjob

Description

Loads events or results of a previously completed search job. The artifacts to load are identified either by the search job id <sid> or a scheduled search name and the time range of the current search. If a saved search name is provided and multiple artifacts are found within that range, the latest artifacts are loaded.

You cannot run the loadjob command on real-time searches.

Syntax

The required syntax is in **bold**.

```
| loadjob
  (<sid> | <savedsearch>)
    [<result_event>]
    [<delegate>]
    [<artifact_offset>]
    [<ignore_running>]
```

Required arguments

You must specify either `sid` or `savedsearch`.

`sid`

Syntax: <string>

Description: The search ID of the job whose artifacts need to be loaded, for example: `1233886270.2`. You can locate the `sid` through the Job Inspector or the `addinfo` command.

`savedsearch`

Syntax: `savedsearch=<user-string>:<app-string>:<search-name-string>`

Description: The unique identifier of a saved search whose artifacts need to be loaded. A saved search is uniquely identified by the triplet {user, app, savedsearch name}, for example: `savedsearch="admin:search:my Saved Search"`. There is no method to specify a wildcard or match-all behavior. All portions of the triplet must be provided.

Optional arguments

`artifact_offset`

Syntax: `artifact_offset=<int>`

Description: Selects a search artifact other than the most recent matching one. For example, if `artifact_offset=1`, the second most recent artifact will be used. If `artifact_offset=2`, the third most recent artifact will be used. If `artifact_offset=0`, selects the most recent. A value that selects past all available artifacts will result in an error.

Default: 0

delegate

Syntax: `job_delegate=<string>`

Description: When specifying a saved search, this option selects jobs that were started by the given user. Scheduled jobs will be run by the delegate "scheduler". Dashboard-embedded searches are run in accordance with the saved search's `dispatchAs` parameter, typically the owner of the search.

Defaults: scheduler

ignore_running

Syntax: `ignore_running=<bool>`

Description: Skip over artifacts whose search is still running.

Default: true

result_event

Syntax: `events=<bool>`

Description: `events=true` loads events, while `events=false` loads results.

Default: false

Usage

The `loadjob` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

The `loadjob` command can be used for a variety of purposes, but one of the most useful is to run a fairly expensive search that calculates statistics. You can use `loadjob` searches to display those statistics for further aggregation, categorization, field selection and other manipulations for charting and display.

After a search job has completed and the results are cached, you can use this command to access or load the results.

Search head clusters

A search head cluster can run the `loadjob` command only on scheduled saved searches. A search head cluster runs searches on results or artifacts that the search head cluster replicates.

For more information on artifact replication, see Search head clustering architecture in the *Distributed Search* manual.

Controlling truncation in search results

To improve the speed of searches, Splunk software truncates search results by default. If you want your search results to include full result sets and search performance is not a concern, you can use the `read_final_results_from_timeliner` setting in the `limits.conf` file to control whether results are truncated when running the `loadjob` command.

When `read_final_results_from_timeliner` is set to true, which is the default, the `loadjob` search returns the sample of the final results, not the full result set. For example, if the full result set is 10,000 results, the search might return only 1,000 results. When `read_final_results_from_timeliner` is set to false, the `loadjob` search returns the full set of search

results. For example, if the full result set is 10,000 results, the search returns 10,000 results.

Splunk Cloud Platform

To change the `read_final_results_from_timeliner` setting in your `limits.conf` file, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `read_final_results_from_timeliner` setting, follow these steps.

Prerequisites

- Only users with file system access, such as system administrators, can edit configuration files.
- Review the steps in How to edit a configuration file in the *Splunk Enterprise Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. In the `[search]` stanza, add the line `read_final_results_from_timeliner = true` to truncate search results, or `read_final_results_from_timeliner = false` to output the full set of search results.

Examples

1. Load the results of a saved search

Loads the results of the latest scheduled execution of saved search `MySavedSearch` in the 'search' application owned by the user `admin`.

```
| loadjob savedsearch="admin:search:MySavedSearch"
```

2. Specifying a saved search with a space in the name

Loads the results of the latest scheduled execution of saved search `Potential Threats` in the 'search' application owned by the user `maria`.

```
| loadjob savedsearch="maria:search:Potential Threats"
```

3. Load the results from a specific search job

Loads the events that were generated by the search job with `id=1233886270.2`.

```
| loadjob 1233886270.2 events=true
```

See also

Commands

[addinfo](#)
[inputcsv](#)
[savedsearch](#)

Related information

Manage search jobs

localize

Description

The `localize` command generates results that represent a list of time contiguous event regions. An event region is a period of time in which consecutive events are separated, at most, by the `maxpause` time value. The regions found can be expanded using the `timeafter` and `timebefore` arguments.

The regions discovered by the `localize` command are meant to be fed into the `map` command. The `map` command uses a different region for each iteration.

Syntax

`localize [<maxpause> [<timeafter> [<timebefore>]]`

Optional arguments

maxpause

Syntax: `maxpause=<int>(s|m|h|d)`

Description: Specify the maximum (inclusive) time between two consecutive events in a contiguous time region.

Default: 1m

timeafter

Syntax: `timeafter=<int>(s|m|h|d)`

Description: Specify the amount of time to add to the output endtime field (expand the time region forward in time).

Default: 30s

timebefore

Syntax: `timebefore=<int>(s|m|h|d)`

Description: Specify the amount of time to subtract from the output starttime field (expand the time region backwards in time).

Default: 30s

Usage

Expanding event ranges

You can expand the event range after the last event or before the first event in the region. These expansions are done arbitrarily, possibly causing overlaps in the regions if the values are larger than `maxpause`.

Event region order

The regions are returned in search order, or descending time for historical searches and data-arrival order for realtime search. The time of each region is the initial pre-expanded start-time.

Other information returned by the localize command

The `localize` command also reports:

- The number of events in the range
- The range duration in seconds
- The region density defined as the `number of events in range divided by <range duration - events per second`.

Examples

1. Search the time range of each previous result for the term "failure"

```
... | localize maxpause=5m | map search="search failure starttimeu=$starttime$ endtimeu=$endtime$" | transaction uid,qid
```

2: Finds suitable regions around where "error" occurs

Searching for "error" and calling the `localize` command finds suitable regions around where error occurs and passes each on to the search inside of the `map` command. Each iteration works with a specific time range to find potential transactions.

```
error | localize | map search="search starttimeu:::$starttime$ endtimeu:::$endtime$ | transaction uid,qid  
maxspan=1h"
```

See also

[map](#), [transaction](#)

localop

Description

Prevents subsequent commands from being executed on remote peers. Tells the search to run subsequent commands locally, instead.

The `localop` command forces subsequent commands to be part of the reduce step of the mapreduce process.

Syntax

`localop`

Examples

Example 1:

The `iplocation` command in this case will never be run on remote peers. All events from remote peers that originate from the initial search, which was for the terms FOO and BAR, are forwarded to the search head. The search head is where the `iplocation` command is run.

```
FOO BAR | localop | iplocation clientip
```

lookup

Description

Use the `lookup` command to invoke field value lookups.

For information about the types of lookups you can define, see *About lookups* in the *Knowledge Manager Manual*.

The `lookup` command supports IPv4 and IPv6 addresses and subnets that use CIDR notation.

Syntax

The required syntax is in **bold**.

```
lookup
[local=<bool>]
[update=<bool>]
<lookup-table-name>
( <lookup-field> [AS <event-field>] )...
[ OUTPUT | OUTPUTNEW (<lookup-destfield> [AS <event-destfield>] )... ]
```

Note: The `lookup` command can accept multiple lookup and event `fields` and `destfields`. For example:

```
... | lookup <lookup-table-name> <lookup-field1> AS <event-field1>, <lookup-field2> AS <event-field2>
OUTPUTNEW <lookup-destfield1> AS <event-destfield1>, <lookup-destfield2> AS <event-destfield2>
```

Required arguments

<lookup-table-name>

Syntax: <string>

Description: Can be either the name of a CSV file that you want to use as the lookup, or the name of a stanza in the `transforms.conf` file that specifies the location of the lookup table file.

Optional arguments

local

Syntax: local=<bool>

Description: If `local=true`, forces the lookup to run on the search head and not on any remote peers.

Default: false

`update`

Syntax: `update=<bool>`

Description: If the lookup table is modified on disk while the search is running, real-time searches do not automatically reflect the update. To do this, specify `update=true`. This does not apply to searches that are not real-time searches. This implies that `local=false`.

Default: `false`

`<lookup-field>`

Syntax: `<string>`

Description: Refers to a field in the lookup table to match against the events. You can specify multiple `<lookup-field>` values.

`<event-field>`

Syntax: `<string>`

Description: Refers to a field in the events from which to acquire the value to match in the lookup table. You can specify multiple `<event-field>` values.

Default: The value of the `<lookup-field>`.

`<lookup-destfield>`

Syntax: `<string>`

Description: Refers to a field in the lookup table to be copied into the events. You can specify multiple `<lookup-destfield>` values.

`<event-destfield>`

Syntax: `<string>`

Description: A field in the events. You can specify multiple `<event-destfield>` values.

Default: The value of the `<lookup-destfield>` argument.

Usage

The `lookup` command is a distributable streaming command when `local=false`, which is the default setting. See [Command types](#).

When using the `lookup` command, if an `OUTPUT` or `OUTPUTNEW` clause is not specified, all of the fields in the lookup table that are not the match fields are used as output fields. If the `OUTPUT` clause is specified, the output lookup fields overwrite existing fields. If the `OUTPUTNEW` clause is specified, the lookup is not performed for events in which the output fields already exist.

Avoid `lookup` reference cycles

When you set up the `OUTPUT` or `OUTPUTNEW` clause for your `lookup`, avoid accidentally creating `lookup` reference cycles, where you intentionally or accidentally reuse the same field names among the match fields and the output fields of a `lookup` search.

For example, if you run a `lookup` search where `type` is both the match field and the output field, you are creating a `lookup` reference cycle. You can accidentally create a `lookup` reference cycle when you fail to specify an `OUTPUT` or `OUTPUTNEW` clause for `lookup`.

For more information about `lookup` reference cycles see Define an automatic `lookup` in Splunk Web in the *Knowledge Manager Manual*.

Optimizing your lookup search

If you are using the `lookup` command in the same pipeline as a **transforming command**, and it is possible to retain the field you will lookup on after the transforming command, do the lookup after the transforming command. For example, run:

```
sourcetype=access_* | stats count by status | lookup status_desc status OUTPUT description
```

and not:

```
sourcetype=access_* | lookup status_desc status OUTPUT description | stats count by description
```

The lookup in the first search is faster because it only needs to match the results of the `stats` command and not all the Web access events.

Running lookup in federated searches

If you use `lookup` in **federated searches**, do not set `local=true`. This setting prevents the federated lookup search from being processed on the remote search heads of the federated providers, which causes the federated search to return incorrect results.

If you are running federated searches over standard mode federated providers, it is also important that the related lookup knowledge objects are duplicated on the local and remote sides of the search. For example, if you are running a federated search which performs a CSV file lookup across your deployment and two remote standard mode federated providers, the CSV file and the CSV lookup definition on your local federated search head must be duplicated on the remote search heads of the standard mode federated providers. See [Custom knowledge object coordination for standard mode federated providers](#) in the *Search Manual*.

For an overview of federated search and federated search terminology, see [About federated search](#) in the *Search Manual*.

Basic example

1. Lookup users and return the corresponding group the user belongs to

Suppose you have a lookup table specified in a stanza named `usertogroup` in the `transforms.conf` file. This lookup table contains (at least) two fields, `user` and `group`. Your events contain a field called `local_user`. For each event, the following search checks to see if the value in the field `local_user` has a corresponding value in the `user` field in the lookup table. For any entries that match, the value of the `group` field in the lookup table is written to the field `user_group` in the event.

```
... | lookup usertogroup user as local_user OUTPUT group as user_group
```

Extended example

1. Lookup price and vendor information and return the count for each product sold by a vendor

This example uses the `tutorialdata.zip` file from the Search Tutorial. You can download this file and follow the instructions to [upload the tutorial data](#) into your Splunk deployment. Additionally, this example uses the `prices.csv` and the `vendors.csv` files. To follow along with this example in your Splunk deployment, download these CSV files and complete the steps in the Use field lookups section of the Search Tutorial for both the `prices.csv` and the `vendors.csv` files. When you create the lookup definition for the `vendors.csv` file, name the lookup `vendors_lookup`. You can skip the step in the tutorial that makes the lookups automatic.

This example calculates the count of each product sold by each vendor.

The `prices.csv` file contains the product names, price, and code. For example:

productId	product_name	price	sale_price	Code
DB-SG-G01	Mediocre Kingdoms	24.99	19.99	A
DC-SG-G02	Dream Crusher	39.99	24.99	B
FS-SG-G03	Final Sequel	24.99	16.99	C
WC-SH-G04	World of Cheese	24.99	19.99	D

The `vendors.csv` file contains vendor information, such as vendor name, city, and ID. For example:

Vendor	VendorCity	VendorID	VendorLatitude	VendorLongitude	VendorStateProvince	VendorCountry	Weight
Anchorage Gaming	Anchorage	1001	61.17440033	-149.9960022	Alaska	United States	3
Games of Salt Lake	Salt Lake City	1002	40.78839874	-111.9779968	Utah	United States	3
New Jack Games	New York	1003	40.63980103	-73.77890015	New York	United States	4
Seals Gaming	San Francisco	1004	37.61899948	-122.375	California	United States	5

The search will query the `vendor_sales.log` file, which is part of the `tutorialdata.zip` file. The `vendor_sales.log` file contains the VendorID, Code, and AcctID fields. For example:

Entries in the <code>vendor_sales.log</code> file
[13/Mar/2018:18:24:02] VendorID=5036 Code=B AcctID=6024298300471575
[13/Mar/2018:18:23:46] VendorID=7026 Code=C AcctID=8702194102896748
[13/Mar/2018:18:23:31] VendorID=1043 Code=B AcctID=2063718909897951
[13/Mar/2018:18:22:59] VendorID=1243 Code=F AcctID=8768831614147676

The following search calculates the count of each product sold by each vendor and uses the time range `All time`.

```
sourcetype=vendor_* | stats count by Code VendorID | lookup prices_lookup Code OUTPUTNEW product_name
```

- The `stats` command calculates the `count` by `Code` and `VendorID`.
- The `lookup` command uses the `prices_lookup` to match the `Code` field in each event and return the `product_name`.

The search results are displayed on displayed on the **Statistics** tab.

New Search

sourcetype=vendor_* | stats count by Code VendorID | lookup prices_lookup Code OUTPUTNEW product_name

All time

✓ 30,244 events (before 3/14/18 4:08:00.000 PM) No Event Sampling

Events Patterns Statistics (6,001) Visualization

20 Per Page

Code	VendorID	count	product_name
A	1001	4	Mediocre Kingdoms
A	1002	1	Mediocre Kingdoms
A	1003	5	Mediocre Kingdoms
A	1004	8	Mediocre Kingdoms
A	1005	4	Mediocre Kingdoms
A	1006	5	Mediocre Kingdoms
A	1007	2	Mediocre Kingdoms
A	1008	5	Mediocre Kingdoms
A	1009	4	Mediocre Kingdoms
A	1010	5	Mediocre Kingdoms

You can extend the search to display more information about the vendor by using the vendors_lookup.

Use the `table` command to return only the fields that you need. In this example you want the `product_name`, `VendorID`, and `count` fields. Use the `vendors_lookup` file to output all the fields in the `vendors.csv` file that match the `VendorID` in each event.

```
sourcetype=vendor_* | stats count by Code VendorID | lookup prices_lookup Code OUTPUTNEW product_name | table product_name VendorID count | lookup vendors_lookup VendorID
```

The revised search results are displayed on the **Statistics** tab.

New Search

sourcetype=vendor_* | stats count by Code VendorID | lookup prices_lookup Code OUTPUTNEW product_name | table product_name VendorID count | lookup vendors_lookup VendorID

All time

✓ 30,244 events (before 3/14/18 4:17:22.000 PM) No Event Sampling

Events Patterns Statistics (6,001) Visualization

20 Per Page

product_name	VendorID	count	Vendor	VendorCity	VendorCountry	VendorLatitude	VendorLongitude	VendorStateProvince	Weight
Mediocre Kingdoms	1001	4	Anchorage Gaming	Anchorage	United States	61.17440033	-149.9960022	Alaska	3
Mediocre Kingdoms	1002	1	Games of Salt Lake City	Salt Lake City	United States	40.78839874	-111.9779968	Utah	3
Mediocre Kingdoms	1003	5	New Jack Games	New York	United States	40.63980103	-73.77890015	New York	4
Mediocre Kingdoms	1004	8	Seals Gaming	San Francisco	United States	37.61899948	-122.375	California	5
Mediocre Kingdoms	1005	4	Lost Angels Games	Los Angeles	United States	33.94250107	-118.4079971	California	5
Mediocre	1006	5	Flyin Hawaiian	Honolulu	United States	21.31870079	-157.9219971	Hawaii	3

To expand the search to display the results on a map, see the [geostats](#) command.

2. IPv6 CIDR match in Splunk Web

In this example, CSV lookups are used to determine whether a specified IPv6 address is in a CIDR subnet. You can follow along with the example by performing these steps in Splunk Web. See Define a CSV lookup in Splunk Web.

Prerequisites

- ◆ Your role must have the upload_lookup_files capability to upload lookup table files in Splunk Web. See Define roles with capabilities in Splunk Enterprise "Securing the Splunk Platform".
- ◆ A CSV lookup table file called ipv6test.csv that contains the following text.

```
ip,expected  
2001:0db8:ffff:ffff:ffff:ffff:ff00/120,true
```

The `ip` field in the lookup table contains the subnet value, not the IP address.

Steps

You have to define a CSV lookup before you can match an IP address to a subnet.

1. Select **Settings > Lookups** to go to the Lookups manager page.
2. Click **Add new** next to **Lookup table files**.
3. Select a **Destination app** from the drop-down list.
4. Click **Choose File** to look for the ipv6test.csv file to upload.
5. Enter ipv6test.csv as the destination filename. This is the name the lookup table file will have on the Splunk server.
6. Click **Save**.
7. In the Lookup table list, click **Permissions** in the Sharing column of the ipv6test lookup you want to share.
8. In the Permissions dialog box, under **Object should appear in**, select **All apps** to share globally. If you want the lookup to be specific to this app only, select **This app only**.
9. Click **Save**.
10. Select **Settings > Lookups**.
11. Click **Add new** next to **Lookup definitions**.
12. Select a **Destination app** from the drop-down list.
13. Give your lookup definition a unique **Name**, like ipv6test.
14. Select **File-based** as the lookup **Type**.
15. Select ipv6test.csv as the **Lookup file** from the drop-down list.
16. Select the **Advanced options** check box.
17. Enter a **Match type** of **CIDR(ip)**.
18. Click **Save**.
19. In the Lookup definitions list, click **Permissions** in the Sharing column of the ipv6test lookup definition you want to share.
20. In the Permissions dialog box, under **Object should appear in**, select **All apps** to share globally. If you want the lookup to be specific to this app only, select **This app only**.

Permissions for lookup table files must be at the same level or higher than those of the lookup definitions that use those files.

21. Click **Save**.
22. In the Search app, run the following search to match the IP address to the subnet.

```
| makeresults | eval ip="2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99" | lookup ipv6test ip OUTPUT  
expected
```

The IP address is in the subnet, so the search displays `true` in the `expected` field. The search results look something like this.

time	expected	ip
2020-11-19 16:43:31	true	2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99

See also

Commands

[appendcols](#)
[inputlookup](#)
[outputlookup](#)
[iplocation](#)
[search](#)

Functions

[cidrmatch](#)

Related information

About lookups in the *Knowledge Manager Manual*

makecontinuous

Description

Makes a field on the x-axis numerically continuous by adding empty buckets for periods where there is no data and quantifying the periods where there is data. This x-axis field can then be invoked by the `chart` and `timechart` commands.

Syntax

```
makecontinuous [<field>] <bins-options>...
```

Required arguments

<bins-options>

Datatype: bins | span | start-end

Description: Discretization options. See "Bins options" for details.

Optional arguments

<field>

Datatype: <field>

Description: Specify a field name.

Bins options

bins

Syntax: bins=<int>

Description: Sets the maximum number of bins to discretize into.

span

Syntax: <log-span> | <span-length>

Description: Sets the size of each bin, using a span length based on time or log-based span.

<start-end>

Syntax: end=<num> | start=<num>

Description: Sets the minimum and maximum extents for numerical bins. Data outside of the [start, end] range is discarded.

Span options

<log-span>

Syntax: [<num>]log[<num>]

Description: Sets to log-based span. The first number is a coefficient. The second number is the base. If the first number is supplied, it must be a real number ≥ 1.0 and <base>. Base, if supplied, must be real number > 1.0 , meaning it must be strictly greater than 1.

span-length

Syntax: [<timescale>]

Description: A span length based on time.

Syntax: <int>

Description: The span of each bin. If using a timescale, this is used as a time range. If not, this is an absolute bin "length."

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month> | <subseconds>

Description: Time scale units.

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

Usage

The `makecontinuous` command is a **transforming command**. See [Command types](#).

Examples

Example 1:

Make `_time` continuous with a span of 10 minutes.

```
... | makecontinuous _time span=10m
```

See also

[chart](#), [timechart](#)

makemv

Description

Converts a single valued field into a multivalue field by splitting the values on a string delimiter or by using a regular expression. The delimiter can be a multicharacter delimiter.

The `makemv` command does not apply to internal fields.

See Use default fields in the *Knowledge Manager Manual*.

Syntax

```
makemv [delim=<string> | tokenizer=<string>] [allowempty=<bool>] [setsv=<bool>] <field>
```

Required arguments

`field`

Syntax: `<field>`

Description: The name of a field to generate the multivalues from.

Optional arguments

`delim`

Syntax: `delim=<string>`

Description: A string value used as a delimiter. Splits the values in `field` on every occurrence of this delimiter.

Default: A single space (" ").

`tokenizer`

Syntax: `tokenizer=<string>`

Description: A regular expression with a capturing group that is repeat-matched against the values in the field. For each match, the first capturing group is used as a value in the newly created multivalue field.

allowempty

Syntax: allowempty=<bool>

Description: Specifies whether to permit empty string values in the multivalue field. When using `delim=true`, repeats of the delimiter string produce empty string values in the multivalue field. For example if `delim=","` and `field="a,,b"`, by default does not produce any value for the empty string. When using the `tokenizer` argument, zero length matches produce empty string values. By default they produce no values.

Default: false

setsv

Syntax: setsv=<bool>

Description: If true, the `makemv` command combines the decided values of the field into a single value, which is set on the same field. (The simultaneous existence of a multivalue and a single value for the same field is a problematic aspect of this flag.)

Default: false

Usage

The `makemv` command is a distributable streaming command. See [Command types](#).

You can use [evaluation functions](#) and [statistical functions](#) on multivalue fields or to return multivalue fields.

Examples

1. Use a comma to separate field values

For sendmail search results, separate the values of "senders" into multiple values. Display the top values.

```
eventtype="sendmail" | makemv delim="," senders | top senders
```

2. Use a colon delimiter and allow empty values

Separate the value of "product_info" into multiple values.

```
... | makemv delim=":" allowempty=true product_info
```

3. Use a regular expression to separate values

The following search creates a result and adds three values to the `my_multival` field. The `makemv` command is used to separate the values in the field by using a regular expression.

```
| makeresults | eval my_multival="one,two,three" | makemv tokenizer="([^\,]+),?" my_multival
```

See also

Commands:

[mvcombine](#)
[mvexpand](#)
[nomv](#)

Functions:

[Multivalue eval functions](#)
[Multivalue stats and chart functions](#)

makeresults

Description

Generates the specified number of search results in temporary memory.

If you do not specify any of the optional arguments, this command runs on the local machine and generates one result with only the `_time` field.

Syntax

The required syntax is in **bold**.

```
| makeresults
[count=<num>]
[annotate=<bool>]
[splunk-server=<string>]
[splunk-server-group=<string>...]
[<format>]
[data=<string>]
```

Required arguments

None.

Optional arguments

`count`

Syntax: `count=<num>`

Description: The number of results to generate. If you do not specify the `annotate` argument, the results have only the `_time` field.

Default: 1

`annotate`

Syntax: `annotate=<bool>`

Description: If `annotate=true`, generates results with the fields shown in the table below.

If `annotate=false`, generates results with only the `_time` field.

Default: false

Fields generated with `annotate=true`

Field	Value
<code>_raw</code>	None.
<code>_time</code>	Date and time that you run the <code>makeresults</code> command.
<code>host</code>	None.

Field	Value
source	None.
sourcetype	None.
splunk_server	The name of the server that the <code>makeresults</code> command is run on.
splunk_server_group	None.

You can use these fields to compute aggregate statistics.

splunk-server

Syntax: `splunk_server=<string>`

Description: Use to generate results on one specific server. Use 'local' to refer to the search head.

Default: local. See the [Usage](#) section.

splunk-server-group

Syntax: `(splunk_server_group=<string>)...`

Description: Use to generate results on a specific server group or groups. You can specify more than one `<splunk_server_group>`.

Default: none. See the [Usage](#) section.

You can use the `format` and `data` arguments to convert CSV- or JSON-formatted data into Splunk events. If you specify these arguments, `makeresults` ignores other arguments such as `count` or `annotate`.

<format>

Syntax: `csv | json`

Description: Specifies the format of the inline data supplied by the `data` argument. If you provide a `format` argument, `makeresults` expects a corresponding `data` argument with inline data that fits the specified format. See [Usage](#).

data

Syntax: `data=<string>`

Description: A collection of inline data that `makeresults` converts into events. If you provide a `data` argument, `makeresults` expects this data to follow the format specified by a corresponding `format` argument. See [Usage](#).

Usage

The `makeresults` command is a report-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

The search results created by the `makeresults` command are created in temporary memory and are not saved to disk or indexed.

You can use this command with the [eval command](#) to generate an empty result for the eval command to operate on. See the [Examples](#) section.

Order-sensitive processors might fail if the internal `_time` field is absent.

Specifying server and server groups

If you use Splunk Cloud Platform, omit any server or server group argument.

If you are using Splunk Enterprise, by default results are generated only on the originating search head, which is equivalent to specifying `splunk_server=local`. If you provide a specific `splunk_server` or `splunk_server_group`, then the number of results you specify with the `count` argument are generated on the all servers or server groups that you specify.

If you specify a server, the results are generated for that server, regardless of the server group that the server is associated with.

If you specify a count of 5 and you target 3 servers, then you will generate 15 total results. If `annotate=true`, the names for each server appear in the `splunk_server` column. This column will show that each server produced 5 results.

Generating results from inline CSV- or JSON-formatted data

Use the `format` and `data` arguments in conjunction to generate events from CSV- or JSON-formatted data.

Inline JSON data must be provided as a series of JSON objects, all within a single JSON array. `makeresults` generates a separate event for each JSON object. The keys of that object become fields, and the object values become field values. Each key must be bracketed in escape quotes. The entire JSON array must be placed within single quotation marks ('').

Here is an example of JSON formatted data:

```
| makeresults format=json data="[{\"name\":\"John\", \"age\":35}, {\"name\":\"Sarah\", \"age\":39}]"
```

Inline data in CSV format consists of a set of lines. The first line contains the schema, or headers, for the CSV table. This first line consists of a comma-separated list of strings, and each string corresponds to a field name. The schema ends when a newline character is reached. Each line following the schema line contains comma-separated field values, and each of these subsequent lines is translated by `makeresults` into an individual event. Use newlines to indicate the end of one event and the beginning of another.

Here is an example of CSV-formatted data:

```
| makeresults format=csv data="name, age John,35 Sarah,39"
```

Inline datasets cannot exceed a threshold of 30,000 characters.

If `makeresults` cannot parse the data for the specified format, it returns an error.

Basic examples

1. Create a result as an input into the eval command

Sometimes you want to use the `eval` command as the first command in a search. However, the `eval` command expects events as inputs. You can create a dummy event at the beginning of a search by using the `makeresults` command. You can then use the `eval` command in your search.

```
| makeresults | eval newfield="some value"
```

The results look something like this:

_time	newfield
2020-01-09 14:35:58	some value

2. Determine if the modified time of an event is greater than the relative time

For events that contain the field `scheduled_time` in UNIX time, determine if the scheduled time is greater than the relative time. The relative time is 1 minute before now. This search uses a subsearch that starts with the `makeresults` command.

```
index=_internal sourcetype=scheduler ( scheduled_time > [ makeresults | eval it=relative_time(now(), "-m") | return $it ] )
```

Extended examples

1. Create daily results for testing

You can use the `makeresults` command to create a series of results to test your search syntax. For example, the following search creates a set of five results:

```
| makeresults count=5
```

The results look something like this:

_time
2020-01-09 14:35:58
2020-01-09 14:35:58
2020-01-09 14:35:58
2020-01-09 14:35:58
2020-01-09 14:35:58

Each result has the same timestamp which, by itself, is not very useful. But with a few additions, you can create a set of unique dates. Start by adding the `streamstats` command to count your results:

```
| makeresults count=5 | streamstats count
```

The results look something like this:

_time	count
2020-01-09 14:35:58	1
2020-01-09 14:35:58	2
2020-01-09 14:35:58	3
2020-01-09 14:35:58	4
2020-01-09 14:35:58	5

You can now use that count to create different dates in the `_time` field, using the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*86400)
```

The calculation multiplies the value in the `count` field by the number of seconds in a day. The result is subtracted from the original `_time` field to get new dates equivalent to 24 hours ago, 48 hours ago, and so forth. The seconds in the date are

different because `_time` is calculated the moment you run the search.

The results look something like this:

<code>_time</code>	<code>count</code>
2020-01-08 14:45:24	1
2020-01-07 14:45:24	2
2020-01-06 14:45:24	3
2020-01-05 14:45:24	4
2020-01-04 14:45:24	5

The dates start from the day before the original date, 2020-01-09, and go back five days.

Need more than five results? Simply change the `count` value in the `makeresults` command.

2. Create hourly results for testing

You can create a series of hours instead of a series of days for testing. Use 3600, the number of seconds in an hour, instead of 86400 in the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600)
```

The results look something like this:

<code>_time</code>	<code>count</code>
2020-01-09 15:35:14	1
2020-01-09 14:35:14	2
2020-01-09 13:35:14	3
2020-01-09 12:35:14	4
2020-01-09 11:35:14	5

Notice that the hours in the timestamp are 1 hour apart.

3. Add a field with string values

You can specify a list of values for a field. But to have the values appear in separate results, you need to make the list a multivalue field and then expand that multivalued list into separate results. Use this search, substituting your strings for buttercup and her friends:

```
| makeresults | eval test="buttercup rarity tenderhoof dash mcintosh fleetfoot mistmane" | makemv delim=" " test | mvexpand test
```

The results look something like this:

<code>_time</code>	<code>test</code>
2020-01-09 16:35:14	buttercup
2020-01-09 16:35:14	rarity

_time	test
2020-01-09 16:35:14	tenderhoof
2020-01-09 16:35:14	dash
2020-01-09 16:35:14	mcintosh
2020-01-09 16:35:14	fleetfoot
2020-01-09 16:35:14	mistmane

4. Create a set of events with multiple fields

Let's start by creating a set of four events. One of the events contains a null value in the `age` field.

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle")
```

- The `streamstats` command is used to create the `count` field. The `streamstats` command calculates a cumulative count for each event, at the time the event is processed.
- The `eval` command is used to create two new fields, `age` and `city`. The `eval` command uses the value in the `count` field.
- The `case` function takes pairs of arguments, such as `count=1, 25`. The first argument is a Boolean expression. When that expression is TRUE, the corresponding second argument is returned.

The results of the search look like this:

_time	age	city	count
2020-02-05 18:32:07	25	San Francisco	1
2020-02-05 18:32:07	39	Seattle	2
2020-02-05 18:32:07	31	San Francisco	3
2020-02-05 18:32:07		Seattle	4

In this example, the `eventstats` command generates the average age for each city. The generated averages are placed into a new field called `avg(age)`.

The following search is the same as the previous search, with the `eventstats` command added at the end:

```
| makeresults count=4 | streamstats count | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle") | eventstats avg(age) BY city
```

- For `San Francisco`, the average age is $28 = (25 + 31) / 2$.
- For `Seattle`, there is only one event with a value. The average is $39 = 39 / 1$. The `eventstats` command places that average in every event for Seattle, including events that did not contain a value for `age`.

The results of the search look like this:

_time	age	avg(age)	city	count
2020-02-05 18:32:07	25	28	San Francisco	1

_time	age	avg(age)	city	count
2020-02-05 18:32:07	39	39	Seattle	2
2020-02-05 18:32:07	31	28	San Francisco	3
2020-02-05 18:32:07		39	Seattle	4

5. Add a field with a set of random numbers

If you need to test something with a set of numbers, you have two options:

- You can add a field with a set of numbers that you specify. This is similar to adding a field with a set of string values, which is shown in the previous example.
- You can add a field with a set of randomly generated numbers by using the `random` function, as shown below:

```
| makeresults count=5 | streamstats count | eval test=random()/random()
```

The results look something like this:

_time	count	test
2020-01-08 14:45:24	1	5.371091109260495
2020-01-07 14:45:24	2	0.4563314783228324
2020-01-06 14:45:24	3	0.804991002129475
2020-01-05 14:45:24	4	1.4946919835236068
2020-01-04 14:45:24	5	24.193952675772845

Use the `round` function to round the numbers up. For example, this search rounds the numbers up to four digits to the right of the decimal:

```
... | eval test=round(random()/random(), 4)
```

The results look something like this:

_time	count	test
2020-01-08 14:45:24	1	5.3711
2020-01-07 14:45:24	2	0.4563
2020-01-06 14:45:24	3	0.8050
2020-01-05 14:45:24	4	1.4947
2020-01-04 14:45:24	5	24.1940

6. Generate a table of results from JSON-formatted data

This `makeresults` search provides a JSON array of objects with the names and ages of a set of individuals.

```
| makeresults format=json data='[{"name": "Larson", "age": 32}, {"name": "Nyeti", "age": 44}, {"name": "Vero", "age": 22}]'
```

`makeresults` transforms this JSON object array into a result table where the keys have been turned into fields and the values have been transformed into field values. The results look something like this:

The results look something like this:

_raw	_time	age	name
{"name":"Larson","age":32}	2021-09-13 22:27:41	32	Larson
{"name":"Nyeti","age":44}	2021-09-13 22:27:41	44	Nyeti
{"name":"Vero","age":22}	2021-09-13 22:27:41	22	Vero

7. Generate a table of results from CSV-formatted data

This `makeresults` search provides an inline collection of CSV-formatted data. It is a table containing the names and ages of a set of individuals. You can add the `fields` command to reorder the fields so they do not appear in alphabetical order.

```
| makeresults format=csv data="name, age Sujata,61 Linus,29 Karina,33" | fields name, age
```

The results look something like this:

name	age
Sujata	61
Linus	29
Karina	33

See also

Commands

[gentimes](#)

map

Description

The `map` command is a looping operator that runs a search repeatedly for each input event or result. You can run the `map` command on a saved search or an ad hoc search.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in *Securing the Splunk Platform*.

Syntax

The required syntax is in **bold**.

```
map  
(<searchoption> | <savedsplunkoption>)  
[maxsearches=int]
```

Required arguments

You must specify either <savedsplunkoption> or <searchoption>.

<savedsplunkoption>

Syntax: <string>

Description: The name of a saved search to run for each input result.

Default: No default.

<searchoption>

Syntax: search=<string>"

Description: An ad hoc search to run for each input result. For example:

```
... | map search="search index=_internal earliest=$myearliest$ latest=$mylatest$".
```

Default: No default.

Optional arguments

maxsearches

Syntax: maxsearches=<int>

Description: The maximum number of searches to run. A message is generated if there are more search results than the maximum number that you specify. Zero (0) does not equate to unlimited searches.

Default: 10

Usage

The `map` command is a dataset processing command. See [Command types](#).

A subsearch can be initiated through a search command such as the `map` command. See [Initiating subsearches with search commands](#) in the Splunk Cloud Platform *Search Manual*.

Known limitations

You cannot use the `map` command after an `append` or `appendpipe` command in your search pipeline.

Variable for field names

When using a saved search or a literal search, the `map` command supports the substitution of \$variable\$ strings that match field names in the input results. A search with a string like \$count\$, for example, will replace the variable with the value of the `count` field in the input search result.

When using the `map` command in a dashboard <form>, use double dollar signs (\$\$) to specify a variable string. For example, \$\$count\$\$. See [Dashboards and forms](#).

Search ID field

The `map` command also supports a search ID field, provided as `$_serial_id$`. The search ID field will have a number that increases incrementally each time that the search is run. In other words, the first run search will have the ID value 1, and the second 2, and so on.

Basic examples

1. Invoke the map command with a saved search

```
error | localize | map mytimebased_savedsearch
```

2. Map the start and end time values

```
... | map search="search starttimeu::$start$ endtimeu::$end$" maxsearches=10
```

Extended examples

1. Use a Sudo event to locate the user logins

This example illustrates how to find a Sudo event and then use the `map` command to trace back to the computer and the time that users logged on before the Sudo event. Start with the following search for the Sudo event.

```
sourcetype=syslog sudo | stats count by user host
```

This search returns a table of results.

User	Host	Count
userA	serverA	1
userB	serverA	3
userA	serverB	2

Pipe these results into the `map` command, substituting the username.

```
sourcetype=syslog sudo | stats count by user host | map search="search index=ad_summary username=$user$type_logon=ad_last_logon"
```

It takes each of the three results from the previous search and searches in the `ad_summary` index for the logon event for the user. The results are returned as a table.

_time	computername	computertime	username	usertime
10/12/16 8:31:35.00 AM	Workstation\$	10/12/2016 08:25:42	userA	10/12/2016 08:31:35 AM

(Thanks to Splunk user Alacer cogitatus for this example.)

See also

Commands

[gentimes](#)
[search](#)

mcollect

Description

Converts events into metric data points and inserts the metric data points into a metric index on the search head. A metric index must be present on the search head for `mcollect` to work properly, unless you are forwarding data to the indexer.

If you are forwarding data to the indexer, your data will be inserted on the indexer instead of the search head.

You can use the `mcollect` command only if your role has the `run_mcollect` capability. See Define roles on the Splunk platform with capabilities in *Securing Splunk Enterprise*.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

The required syntax is in **bold**.

```
| mcollect index=<string>
| [ file=<string> ]
| [ split=<true | false | allnums> ]
| [ spool=<bool> ]
| [ prefix_field=<string> ]
| [ host=<string> ]
| [ source=<string> ]
| [ sourcetype=<string> ]
| [ marker=<string> ]
| [ <field-list> ]
```

Required arguments

index

Syntax: `index=<string>`

Description: Name of the metric index where the collected metric data is added.

field-list

Syntax: `<field>, ...`

Description: A list of dimension fields. Required if `split=true`. Optional if `split=false` or `split=allnums`. If unspecified, which implies that `split=false`, `mcollect` treats all fields as dimensions for the data point except for the `metric_name`, `prefix_field`, and all internal fields.

Default: No default value

Optional arguments

file

Syntax: `file=<string>`

Description: The file name where you want the collected metric data to be written. Only applicable when `spool=false`. You can use a timestamp or a random number for the file name by specifying either `file=$timestamp$` or `file=$random$`.

Default: \$random\$_metrics.csv

split

Syntax: split=<true | false | allnums>

Description: Determines how `mcollect` identifies the measures in an event. See [How to use the split argument](#).

Default: false

spool

Syntax: spool=<bool>

Description: If set to true, the metrics data file is written to the Splunk spool directory,

`$SPLUNK_HOME/var/spool/splunk`, where the file is indexed. Once the file is indexed, it is removed. If set to false, the file is written to the `$SPLUNK_HOME/var/run/splunk` directory. The file remains in this directory unless further automation or administration is done.

Default: true

prefix_field

Syntax: prefix_field=<string>

Description: Only applicable when `split=true`. If specified, any data point with that field missing is ignored. Otherwise, the field value is prefixed to the metric name. See [Set a prefix field](#)

Default: No default value

host

Syntax: host=<string>

Description: The name of the host that you want to specify for the collected metrics data. Only applicable when `spool=true`.

Default: No default value

source

Syntax: source=<string>

Description: The name of the source that you want to specify for the collected metrics data.

Default: If the search is scheduled, the name of the search. If the search is ad-hoc, the name of the file that is written to the `var/spool/splunk` directory containing the search results.

sourcetype

Syntax: sourcetype=<string>

Description: The name of the source type that is specified for the collected metrics data. The Splunk platform does not calculate license usage for data indexed with `mcollect_stash`, the default source type. If you change the value of this setting to a different source type, the Splunk platform calculates license usage for any data indexed by the `mcollect` command.

Default: `mcollect_stash`

Do not change this setting without assistance from Splunk Professional Services or Splunk Support. Changing the source type requires a change to the `props.conf` file.

marker

Syntax: marker=<string>

Description: A string of one or more comma-separated key/value pairs that `mcollect` adds as dimensions to the metric data points it generates, for the purpose of searching on those metric data points later. For example, you could add the name of the `mcollect` search that you are running, like this:

`marker=savedsearch=firewall_top_src_ip`. This allows you to run searches later that isolate the metric data points created by that `mcollect` search, simply by adding `savedsearch=firewall_top_src_ip` to the search string.

Usage

You use the `mcollect` command to convert events into metric data points to be stored in a metric index on the search head. The metrics data uses a specific format for the metrics fields. See Metrics data format in *Metrics*.

The `mcollect` command causes new data to be written to a metric index for every run of the search.

All metrics search commands are case sensitive. This means, for example, that `mcollect` treats as the following as three distinct values of `metric_name`: `cap.gear`, `CAP.GEAR`, and `Cap.Gear`.

The Splunk platform cannot index metric data points that contain `metric_name` fields which are empty or composed entirely of white spaces.

If you are upgrading to version 8.0.0

After you upgrade your search head and indexer clusters to version 8.0.x of Splunk Enterprise, edit `limits.conf` on each search head cluster and set the `always_use_single_value_output` setting under the `[mcollect]` stanza to `false`. This lets these nodes use the "multiple measures per metric data point" schema when you convert logs to metrics with the `mcollect` command or use metrics rollups. This schema increases your data storage capacity and improves metrics search performance.

How to use the `split` argument

The `split` argument determines how `mcollect` identifies the measurement fields in your search. It defaults to `false`.

When `split=false`, your search needs to explicitly identify its measurement fields. If necessary it can use `rename` or `eval` conversions to do this.

- If you have single-metric events, your `mcollect` search must produce results with a `metric_name` field that provides the name of the measure, and a `_value` field that provides the measure's numeric value.
- If you have multiple-metric events, your `mcollect` search must produce results that follow this syntax:
`metric_name:<metric_name>=<numeric_value>`. `mcollect` treats each of these fields as a measurement. `mcollect` treats the remaining fields as dimensions.

When you set `split=true`, you use `field-list` to identify the dimensions in your search. `mcollect` converts any field that is not in the `field-list` into a measurement. The only exceptions are internal fields beginning with an underscore and the `prefix_field`, if you have set one.

When you set `split=allnums`, `mcollect` treats all numeric fields as metric measures and all non-numeric fields as dimensions. You can optionally use `field-list` to declare that `mcollect` should treat certain numeric fields in the events as dimensions.

Set a prefix field

Use the `prefix_field` argument to apply a prefix to the metric fields in your event data.

For example, if you have the following data:

```
type=cpu usage=0.78 idle=0.22
```

You have two metric fields, `usage` and `idle`.

Say you include the following in an `mcollect` search of that data:

```
...split=true prefix_field=type...
```

Because you have set `split = true` the Splunk software automatically converts those fields into measures, because they are not otherwise identified in a `<field-list>`. Then it applies the value of the specified `prefix_field` as a prefix to the metric field names. In this case, because you have specified the `type` field as the prefix field, its value, `cpu`, becomes the metric name prefix. The results look like this:

metric_name:cpu.usage	metric_name:cpu.idle
0.78	0.22

Time

If the `_time` field is present in the results, the Splunk software uses it as the timestamp of the metric data point. If the `_time` field is not present, the current time is used.

field-list

If `field-list` is not specified, `mcollect` treats all fields as dimensions for the metric data points it generates, except for the `prefix_field` and internal fields (fields with an underscore '_' prefix). If `field-list` is specified, the list must appear at the end of the `mcollect` command arguments. If `field-list` is specified, all fields are treated as metric values, except for the fields in `field-list`, the `prefix-field`, and internal fields.

The name of each metric value is the field name prefixed with the `prefix_field` value.

Effectively, one metric data point is returned for each qualifying field that contains a numerical value. If one search result contains multiple qualifying metric name/value pairs, the result is split into multiple metric data points.

Examples

The following examples show how to use the `mcollect` command to convert events into multiple-value metric data points.

1: Generate metric data points that break out jobs and latency metrics by user

The following example specifies the metrics that should appear in the resulting metric data points, and splits them by user. Note that it does not use the `split` argument, so the search has to use a `rename` conversion to explicitly identify the measurements that will appear in the data points.

```
index="_audit" search_id info total_run_time | stats count(search_id) as jobs avg(total_run_time) as latency  
by user | rename jobs as metric_name:jobs latency as metric_name:latency | mcollect index=mcollect_test
```

Here are example results of that search:

_time	user	metric_name:jobs	metric_name:latency
1563318689	admin	25	3.810555555555575
1563318689	splunk-system-user	129	0.2951162790697676

2: Generate metric data points that break out event counts and total runtimes by user

This search sets `split=true` so it automatically converts fields not otherwise identified as dimensions by the `<field-list>` into metrics. The search identifies `user` as a dimension.

```
index="_audit" info=completed | stats max(total_run_time) as runtime max(event_count) as events by user | mcollect index=mcollect_test split=t user
```

Here are example results of that search:

_time	user	metric_name:runtim	metric_name:events
1563318968	admin	0.29	293
1563318968	splunk-system-user	0.04	3

See also

Commands

[collect](#)
[meventcollect](#)

metadata

Description

The `metadata` command returns a list of sources, sourcetypes, or hosts from a specified index or distributed search peer. The `metadata` command returns information accumulated over time. You can view a snapshot of an index over a specific timeframe, such as the last 7 days, by using the time range picker.

See [Usage](#).

Syntax

```
| metadata type=<metadata-type> [<index-specifier>]... [splunk_server=<wc-string>] [splunk_server_group=<wc-string>]...
```

Required arguments

type

Syntax: `type= hosts | sources | sourcetypes`

Description: The type of metadata to return. This must be one of the three literal strings: hosts, sources, or sourcetypes.

Optional arguments

index-specifier

Syntax: `index=<index_name>`

Description: Specifies the index from which to return results. You can specify more than one index. Wildcard characters (*) can be used. To match non-internal indexes, use `index=*`. To match internal indexes, use `index=_*`.

Example: `| metadata type=hosts index=cs* index=na* index=ap* index=eu*`

Default: The default index, which is usually the **main** index.

splunk_server

Syntax: `splunk_server=<wc-string>`

Description: Specifies the distributed search peer from which to return results.

If you are using Splunk Cloud Platform, omit this parameter.

If you are using Splunk Enterprise, you can specify only one `splunk_server` argument. However, you can use a wildcard when you specify the server name to indicate multiple servers. For example, you can specify `splunk_server=peer01` or `splunk_server=peer*`. Use `local` to refer to the search head.

Default: All configured search peers return information

splunk_server_group

Syntax: `splunk_server_group=<wc-string>...`

Description: Limits the results to one or more server groups. If you are using Splunk Cloud, omit this parameter. You can specify a wildcard character in the string to indicate multiple server groups.

Usage

The `metadata` command is a report-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Although the `metadata` command fetches data from all peers, any command run after it runs only on the search head.

The command shows the first, last, and most recent events that were seen for each value of the specified `metadata` type. For example, if you search for:

```
| metadata type=hosts
```

Your results should look something like this:

	firstTime	host	lastTime	recentTime	totalCount	type
1	1293005265	apache3.splunk.com	1293609574	1293609574	27888	hosts
2	1293005220	apache2.splunk.com	1293609574	1293609574	27705	hosts
3	1293005265	apache1.splunk.com	1293609555	1293609555	9199	hosts
4	1293005241	mysql.splunk.com	1293492848	1293492848	180	hosts

- The `firstTime` field is the timestamp for the first time that the indexer saw an event from this host.
- The `lastTime` field is the timestamp for the last time that the indexer saw an event from this host.
- The `recentTime` field is the `indexetime` for the most recent time that the index saw an event from this host. In other words, this is the time of the last update.
- The `totalCount` field is the total number of events seen from this host.
- The `type` field is the specified type of metadata to display. Because this search specifies `type=hosts`, there is also a `host` column.

In most cases, when the data is streaming live, the `lastTime` and `recentTime` field values are equal. If the data is historical, however, the values might be different.

In small testing environments, the data is complete. However, in environments with large numbers of values for each category, the data might not be complete. This is intentional and allows the `metadata` command to operate within reasonable time and memory usage.

Real-time searches

Running the `metadata` command in a real-time search that returns a large number of results will very quickly consume all the available memory on the Splunk server. Use caution when you use the `metadata` command in real-time searches.

Time ranges

Set the time range using the Time Range Picker. You cannot use the `earliest` or `latest` time range modifiers in the search string. Time range modifiers must be set before the first piped command and generating commands in general do not allow anything to be specified before the first pipe.

If you specify a time range other than `All Time` for your search, the search results might not be precise. The metadata is stored as aggregate numbers for each bucket on the index. A bucket is either included or not included based on the time range you specify.

For example, you run the following search specifying a time range of `Last 7 days`. The time range corresponds to January 1st to January 7th.

```
| metadata type=sourcetypes index=ap
```

There is a bucket on the index that contains events from both December 31st and January 1st. The metadata from that bucket is included in the information returned from search.

Maximum results

By default, a maximum of 10,000 results are returned. This maximum is controlled by the `maxresultrows` setting in the `[metadata]` stanza in the `limits.conf` file.

Examples

1. Search multiple indexes

Return the metadata for indexes that represent different regions.

```
| metadata type=hosts index=cs* index=na* index=ap* index=eu*
```

2. Search for sourcetypes

Return the values of `sourcetypes` for events in the `_internal` index.

```
| metadata type=sourcetypes index=_internal
```

This returns the following report.

Events	Patterns	Statistics (11)	Visualization
20 Per Page Format Preview			
firstTime	lastTime	recentTime	sourcetype
1480952400	1483458059	1483543802	mongod
1480952431	1483543801	1483543809	scheduler
1481816674	1483112457	1483198203	splunk_migration
1481816665	1483112449	1483198203	splunk_version
1480952409	1483458062	1483543802	splunk_web_access
1480952405	1483543803	1483543808	splunk_web_service
1480952397	1483543808	1483543812	splunkd
1480952407	1483543801	1483543808	splunkd_access
1480952400	1483458056	1483543802	splunkd_conf
1480952399	1483458056	1483543802	splunkd_stderr
1480952409	1483543851	1483543809	splunkd_ui_access
totalCount	type		
3104	sourcetypes		
15178	sourcetypes		
4	sourcetypes		
4	sourcetypes		
30	sourcetypes		
5370	sourcetypes		
4208946	sourcetypes		
12641	sourcetypes		
30	sourcetypes		
56	sourcetypes		
4316	sourcetypes		

3. Format the results from the metadata command

You can also use the [fieldformat command](#) to format the results of the firstTime, lastTime, and recentTime columns to be more readable.

```
| metadata type=sourcetypes index=_internal | rename totalCount as Count firstTime as "First Event" lastTime as "Last Event" recentTime as "Last Update" | fieldformat Count=tostring(Count, "commas") | fieldformat "First Event"=strftime('First Event', "%c") | fieldformat "Last Event"=strftime('Last Event', "%c") | fieldformat "Last Update"=strftime('Last Update', "%c")
```

Click on the **Count** field label to sort the results and show the highest count first. Now, the results are more readable:

Events	Patterns	Statistics (11)	Visualization		
20 Per Page Format Preview					
sourcetype	type	First Event	Last Event	Last Update	Count
splunkd	sourcetypes	Mon Dec 5 07:39:57 2016	Wed Jan 4 07:30:08 2017	Wed Jan 4 07:30:12 2017	4,208,946
scheduler	sourcetypes	Mon Dec 5 07:40:31 2016	Wed Jan 4 07:30:01 2017	Wed Jan 4 07:30:09 2017	15,178
splunkd_access	sourcetypes	Mon Dec 5 07:40:07 2016	Wed Jan 4 07:30:01 2017	Wed Jan 4 07:30:08 2017	12,641
splunk_web_service	sourcetypes	Mon Dec 5 07:40:05 2016	Wed Jan 4 07:30:03 2017	Wed Jan 4 07:30:08 2017	5,370
splunkd_ui_access	sourcetypes	Mon Dec 5 07:40:09 2016	Wed Jan 4 07:22:31 2017	Wed Jan 4 07:30:09 2017	4,316
mongod	sourcetypes	Mon Dec 5 07:40:00 2016	Tue Jan 3 07:40:59 2017	Wed Jan 4 07:30:02 2017	3,104
splunkd_stderr	sourcetypes	Mon Dec 5 07:39:59 2016	Tue Jan 3 07:40:56 2017	Wed Jan 4 07:30:02 2017	56
splunk_web_access	sourcetypes	Mon Dec 5 07:40:09 2016	Tue Jan 3 07:41:02 2017	Wed Jan 4 07:30:02 2017	30
splunkd_conf	sourcetypes	Mon Dec 5 07:40:00 2016	Tue Jan 3 07:40:56 2017	Wed Jan 4 07:30:02 2017	30
splunk_migration	sourcetypes	Thu Dec 15 07:44:34 2016	Fri Dec 30 07:40:57 2016	Sat Dec 31 07:30:03 2016	4
splunk_version	sourcetypes	Thu Dec 15 07:44:25 2016	Fri Dec 30 07:40:49 2016	Sat Dec 31 07:30:03 2016	4

4. Return values of "sourcetype" for events in a specific index on a specific server

Return values of "sourcetype" for events in the `_audit` index on server foo.

```
| metadata type=sourcetypes index=_audit splunk_server=foo
```

See also

[dbinspect](#)
[tstats](#)

metasearch

Description

Retrieves event `metadata` from indexes based on terms in the <logical-expression>.

Syntax

`metasearch [<logical-expression>]`

Optional arguments

<logical-expression>

Syntax: <time-opts> | <search-modifier> | [NOT] <logical-expression> | <index-expression> | <comparison-expression> | <logical-expression> [OR <logical-expression>]

Description: Includes time and search modifiers, comparison and index expressions.

Logical expression

<comparison-expression>

Syntax: <field><cmp><value>

Description: Compare a field to a literal value or values of another field.

<index-expression>

Syntax: "<string>" | <term> | <search-modifier>

<time-opts>

Syntax: [<timeformat>] [<time-modifier>]...

Comparison expression

<cmp>

Syntax: = | != | < | <= | > | >=

Description: Comparison operators.

<field>

Syntax: <string>

Description: The name of one of the fields returned by the `metasearch` command. See [Usage](#).

<lit-value>

Syntax: <string> | <num>

Description: An exact, or literal, value of a field that is used in a comparison expression.

<value>

Syntax: <lit-value> | <field>

Description: In comparison-expressions, the literal value of a field or another field name. The <lit-value> must be a number or a string.

Index expression

<search-modifier>

Syntax: <field-specifier> | <savedsplunk-specifier> | <tag-specifier>

Time options

The search allows many flexible options for searching based on time. For a list of time modifiers, see the topic [Time modifiers for search](#) in the *Search Manual*.

<timeformat>

Syntax: timeformat=<string>

Description: Set the time format for starttime and endtime terms. By default, timestamp is formatted:

timeformat=%m/%d/%Y:%H:%M:%S .

<time-modifier>

Syntax: earliest=<time_modifier> | latest=<time_modifier>

Description: Specify start and end times using relative or absolute time. For more about the time modifier index, see [Specify time modifiers in your search](#) in the *Search Manual*.

Usage

The `metasearch` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

The `metasearch` command returns these fields:

Field	Description
host	A default field that contains the host name or IP address of the network device that generated an event.
index	The repository for data. When the Splunk platform indexes raw data, it transforms the data into searchable events.
source	A default field that identifies the source of an event, that is, where the event originated.
sourcetype	A default field that identifies the data structure of an event.
splunk_server	The name of the instance where Splunk Enterprise is installed.
_time	The _time field contains an event's timestamp expressed in UNIX time.

Examples

Example 1:

Return metadata on the default index for events with "404" and from host "webserver1".

```
| metasearch 404 host="webserver1"
```

See also

Commands

[metadata](#)
[search](#)

meventcollect

Description

Converts events generated by **streaming search commands** into metric data points and inserts the data into a metric index on the indexers.

You can use the `meventcollect` command only if your role has the `run_mcollect` capability. See Define roles on the Splunk platform with capabilities in *Securing Splunk Enterprise*.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

The required syntax is in **bold**.

```
| meventcollect index=<string>
[ file=<string> ]
[ split=<bool> ]
[ spool=<bool> ]
[ prefix_field=<string> ]
[ host=<string> ]
[ source=<string> ]
[ sourcetype=<string> ]
[ <field-list> ]
```

Required arguments

index

Syntax: `index=<string>`

Description: Name of the metric index where the collected metric data is added.

field-list

Syntax: `<field>, ...`

Description: A list of dimension fields. Required if `split=true`. Optional if `split=false`. If unspecified (which implies that `split=false`), `meventcollect` treats all fields as dimensions for the data point, except for the `metric_name`, `prefix_field`, and all internal fields.

Default: No default value

Optional arguments

file

Syntax: `file=<string>`

Description: The file name where you want the collected metric data to be written. Only applicable when `spool=false`. You can use a timestamp or a random number for the file name by specifying either `file=$timestamp$` or `file=$random$`.

Default: `$random$_metrics.csv`

split

Syntax: split=<bool>

Description: Determines how `meventcollect` identifies the measures in an event. See [How to use the split argument](#).

Default: false

spool

Syntax: spool=<bool>

Description: If set to true, `meventcollect` writes the metrics data file to the Splunk spool directory, `$SPLUNK_HOME/var/spool/splunk`, where the file is indexed automatically. If set to false, `meventcollect` writes the file to the `$SPLUNK_HOME/var/run/splunk` directory. The file remains in this directory unless further automation or administration is done.

Default: true

prefix_field

Syntax: prefix_field=<string>

Description: Only applicable when `split=true`. If specified, `meventcollect` ignores any data point with that field missing. Otherwise, `meventcollect` prefixes the field value to the metric name. See [Set a prefix field](#).

Default: No default value

host

Syntax: host=<string>

Description: The name of the host that you want to specify for the collected metrics data. Only applicable when `spool=true`.

Default: No default value

source

Syntax: source=<string>

Description: The name of the source that you want to specify for the collected metrics data.

Default: If the search is scheduled, the name of the search. If the search is ad-hoc, `meventcollect` writes the name of the file to the `var/spool/splunk` directory containing the search results.

sourcetype

Syntax: sourcetype=<string>

Description: The name of the source type that you want to specify for the collected metrics data.

Default: metrics_csv

Do not change this setting without assistance from Splunk Professional Services or Splunk Support. Changing the source type requires a change to the `props.conf` file.

Usage

You use the `meventcollect` command to convert streaming events into metric data to be stored in a metric index on the indexers. The metrics data uses a specific format for the metrics fields. See Metrics data format in *Metrics*.

Only **streaming commands** can precede the `meventcollect` command so that results can be ingested on the indexers. If you would like to run a search that uses **transforming commands** to generate metric data points, use `mcollect` instead of `meventcollect`.

The `meventcollect` command causes new data to be written to a metric index for every run of the search. In addition, if you run an `meventcollect` search over large amounts of data, it potentially can overwhelm indexers and indexer clusters

that do not have a significant amount of capacity.

All metrics search commands are case sensitive. This means, for example, that `meventcollect` treats as the following as three distinct values of `metric_name`: `cap.gear`, `CAP.GEAR`, and `Cap.Gear`.

The Splunk platform cannot index metric data points that contain `metric_name` fields which are empty or composed entirely of white spaces.

How to use the `split` argument

The `split` argument determines how `meventcollect` identifies the measurement fields in your search. It defaults to `false`.

When `split=false`, your search needs to explicitly identify its measurement fields. If necessary it can use `rename` or `eval` conversions to do this.

- If you have single-metric events, your `meventcollect` search must produce results with a `metric_name` field that provides the name of the measure, and a `_value` field that provides the measure's numeric value.
- If you have multiple-metric events, your `meventcollect` search must produce results that follow this syntax:
`metric_name:<metric_name>=<numeric_value>`. Each of these fields will be treated as a measurement.
`meventcollect`treats the remaining fields as dimensions.

When you set `split=true`, you use `field-list` to identify the dimensions in your search. `meventcollect` converts any field that is not in the `field-list` into a measurement. The only exceptions are internal fields beginning with an underscore and the `prefix_field`, if you have set one.

When you set `split=allnums`, `meventcollect` treats all numeric fields as metric measures and all non-numeric fields as dimensions. You can optionally use `field-list` to declare that `meventcollect` should treat certain numeric fields in the events as dimensions.

Set a prefix field

Use the `prefix_field` argument to apply a prefix to the metric fields in your event data.

For example, if you have the following data:

```
type=cpu usage=0.78 idle=0.22
```

You have two metric fields, `usage` and `idle`.

Say you include the following in an `mcatalog` search of that data:

```
...split=true prefix_field=type...
```

Because you have set `split = true` the Splunk software automatically converts those fields into measures, because they are not otherwise identified in a `<field-list>`. Then it applies the value of the specified `prefix_field` as a prefix to the metric field names. In this case, because you have specified the `type` field as the prefix field, its value, `cpu`, becomes the metric name prefix. The results look like this:

<code>metric_name:cpu.usage</code>	<code>metric_name:cpu.idle</code>
------------------------------------	-----------------------------------

0.78	0.22
------	------

Examples

1: Collect metrics.log data into a metrics index

The following example shows you how to collect metrics log data into a metric index called 'my_metric_index'.

```
index=_internal source=*/metrics.log | eval prefix = group + "." + name | meventcollect
index=my_metric_index split=true prefix_field=prefix name group
```

See also

Commands

[collect](#)
[mcollect](#)

mpreview

Description

Use `mpreview` to get an idea of the kinds of metric time series that are stored in your metrics indexes and to troubleshoot your metrics data.

`mpreview` returns a preview of the raw **metric data points** in a specified metric index that match a provided filter. By default, `mpreview` retrieves a target of five metric data points per **metric time series** from each metrics **time-series index file** (.tsidx file) associated with the search. You can change this target amount with the `target_per_timeseries` argument.

By design, `mpreview` returns metric data points in JSON format.

The `mpreview` command cannot search data that was indexed prior to your upgrade to the 8.0.x version of the Splunk platform.

You can use the `mpreview` command only if your role has the `run_msearch` capability. See [Define roles on the Splunk platform with capabilities](#) in *Securing Splunk Enterprise*.

Syntax

The required syntax is in **bold**.

```
| mpreview
[filter=<string>]
[<index-opt>]...
[splunk_server=<wc-string>]
[splunk_server_group=<wc-string>]...
[earliest=<time-specifier>]
[latest=<time-specifier>] :
[chunk_size=<unsigned-integer>]
```

[target_per_timeseries=<unsigned-integer>]

Required arguments

None. By default all types of terms are returned.

Optional arguments

chunk_size

Syntax: chunk_size=<unsigned-integer>

Description: Advanced option. This argument controls how many **metric time series** are retrieved at a time from a single **time-series index file** (.tsidx file) when the Splunk software processes searches. Lower this setting from its default only when you find a particular `mpreview` search is using too much memory, or when it infrequently returns events. This can happen when a search groups by excessively high-cardinality dimensions (dimensions with very large amounts of distinct values). In such situations, a lower `chunk_size` value can make `mpreview` searches more responsive, but potentially slower to complete. A higher `chunk_size`, on the other hand, can help long-running searches to complete faster, with the potential tradeoff of causing the search to be less responsive. For `mpreview`, `chunk_size` cannot be set lower than 10.

For more information about this setting, see [Use chunk_size to regulate mpreview performance](#).

Default: 1000

For Splunk Enterprise: The default value for the the `chunk_size` argument is set by the `chunk_size` setting for the [msearch] stanza in limits.conf.

earliest

Syntax: earliest=<time-specifier>

Description: Specify the earliest `_time` for the time range of your search. You can specify an exact time (`earliest="11/5/2016:20:00:00"`) or a relative time (`earliest=-h` or `earliest=@w0`).

For more information about setting exact times see [Date and time format variables](#). For more information about setting relative times, see [Time modifiers](#). Subsecond options are available only if you are searching over a metrics index with millisecond timestamp resolution.

filter

Syntax: filter= "<string>"

Description: An arbitrary boolean expression over the dimension or `metric_name`.

index-opt

Syntax: index=<index-name> (index=<index-name>)...

Description: Limits the search to results from one or more indexes. You can use wildcard characters (*). To match non-internal indexes, use `index=*`. To match internal indexes, use `index=_*`.

latest

Syntax: latest=<time-specifier>

Description: Specify the latest time for the `_time` range of your search. You can specify an exact time (`latest="11/12/2016:20:00:00"`) or a relative time (`latest=-30m` or `latest=@w6`).

For more information about setting exact times see [Date and time format variables](#). For more information about setting relative times, see [Time modifiers](#). Subsecond options are available only if you are searching over a metrics index with millisecond timestamp resolution.

splunk_server

Syntax: `splunk_server=<wc-string>`

Description: Specifies the distributed search peer from which to return results. If you are using Splunk Enterprise, you can specify only one `splunk_server` argument. However, you can use a wildcard when you specify the server name to indicate multiple servers. For example, you can specify `splunk_server=peer01` or `splunk_server=peer*`. Use `local` to refer to the search head.

splunk_server_group

Syntax: `splunk_server_group=<wc-string>`

Description: Limits the results to one or more server groups. If you are using Splunk Cloud Platform, omit this parameter. You can specify a wildcard character in the string to indicate multiple server groups.

target_per_timeseries

Syntax: `target_per_timeseries=<unsigned-integer>`

Description Determines the target number of metric data points to retrieve per metric time series from each metrics time-series index file (`.tsidx` file) associated with the `mpreview` search. If a time series has less than the `target_per_timeseries` of data points within a `.tsidx` file, the search head retrieves all of the data points for that time series within that particular `.tsidx` file.

If you set `target_per_timeseries` to 0 it returns all data points available within the given time range for each time series. This search will likely be very large in scale and therefore very slow to complete. If you must search on a large number of metric data points, use `mstats` instead.

For more information about this setting, see [How the target_per_timeseries argument works](#).

Default: 5

The default value for the `target_per_timeseries` argument is set by the `target_per_timeseries` setting for the `[msearch]` stanza in `limits.conf`

Usage

This search command generates a list of individual metric data points from a specified metric index that match a provided filter. The filter can be any arbitrary boolean expression over the dimensions or the `metric_name`. Specify `earliest` and `latest` to override the time range picker settings.

For more information about setting `earliest` and `latest`, see [Time modifiers](#).

The `mpreview` command is designed to display individual metric data points in JSON format. If you want to aggregate metric data points, use the `mstats` command.

All metrics search commands are case sensitive. This means, for example, that `mpreview` treats as the following as three distinct values of `metric_name`: `cap.gear`, `CAP.GEAR`, and `Cap.Gear`.

How the target_per_timeseries argument works

Unfiltered `mpreview` searches can cover extremely large numbers of raw metric data points. In some cases the sheer number of data points covered by the search can cause such searches to be slow or unresponsive.

The `target_per_timeseries` argument makes the `mpreview` command more responsive while giving you a relatively broad preview of your metric data. It limits the number of metric data points that `mpreview` can return from each metric time

series in each `.tsidx` file covered by the search.

For example, if you have 10 metrics `.tsidx` files that each contain 100 metric time series, and each time series has ≥ 5 data points. If you set `target_per_timeseries=5` in the search, you should expect a maximum of $10 \times 100 \times 5 = 5000$ metric data points to be returned by the search.

On the other hand, say you have 10 metrics `.tsidx` files that each contain 100 metric time series, but in this case, 50 of those time series have 3 data points and the other 50 of those time series have ≥ 5 data points. If you set `target_per_timeseries=5` in the search, you should expect to get $10 \times ((50 \times 3) + (50 \times 5)) = 4000$ data points.

The `target_per_timeseries` argument is especially useful when the number of metric data points covered by your `mpreview` search is significantly larger than the number of metric time series covered by the search. It's not particularly helpful if the number of data points in your search are slightly larger than or equal to the number of metric time series in the search.

You can run this search to determine the number of metric data points that could potentially be covered by an `mpreview` search:

```
| metadata index=<metric_index_name> type=hosts datatype=metric | fields totalCount
```

You can run this search to determine the number of metric time series that could potentially be covered by an `mpreview` search:

```
| mstats count(*) WHERE index=<metric_index_name> by _timeseries | stats count
```

Use `chunk_size` to regulate `mpreview` performance

If you find that `mpreview` is slow or unresponsive despite the `target_per_timeseries` argument you can also use `chunk_size` to regulate `mpreview` behavior. Reduce the `chunk_size` to make the search more responsive with the potential tradeoff of making the search slower to complete. Raise the `chunk_size` to help the `mpreview` search to complete faster, with the potential tradeoff of making it less responsive.

Examples

1. Return data points that match a specific filter

This search returns individual data points from the `_metrics` index that match a specific filter.

```
| mpreview index=_metrics filter="group=queue name=indexqueue metric_name=*.current_size"
```

Here is an example of a JSON-formatted result of the above search.

i	Time	Event
>	10/21/19 6:06:14,000 PM	<pre>{ [-] group: queue metric_name:spl.mlog.queue.current_size: 18 metric_name:spl.mlog.queue.current_size_kb: 5 metric_name:spl.mlog.queue.largest_size: 20 metric_name:spl.mlog.queue.max_size_kb: 500 metric_name:spl.mlog.queue.smallest_size: 0 name: indexqueue } Show as raw text</pre>

2. Return individual data points from the metrics index

```
| mpreview index=_metrics
```

3. Lower chunk_size to improve mpreview performance

The following search lowers `chunk_size` so that it returns 100 metric time series worth of metric data points in batches from `tsidx` files that belong to the `_metrics` index. Ordinarily it would return 1000 metric time series in batches.

```
| mpreview index=_metrics chunk_size=100
```

4. Speed up an mpreview search with target_per_timeseries

The following search uses `target_per_timeseries` to return a maximum of five metric data points per time series in each `tsidx` file searched in the `_metrics` index.

```
| mpreview index=_metrics target_per_timeseries=5
```

See also

Commands

[mcatalog](#)
[mcollect](#)
[mstats](#)

msearch

The `msearch` command is an alias for the `mpreview` command. See the `mpreview` command for the syntax and examples.

See also

Commands

[mcatalog](#)
[mstats](#)

mstats

Description

Use the `mstats` command to analyze metrics. This command performs statistics on the `measurement`, `metric_name`, and `dimension` fields in metric indexes. You can use `mstats` in **historical searches** and **real-time searches**. When you use `mstats` in a real-time search with a time window, a historical search runs first to backfill the data.

The `mstats` command provides the best search performance when you use it to search a single `metric_name` value or a small number of `metric_name` values.

Syntax

The required syntax is in **bold**.

```
| mstats  
[chart=<bool>]  
[<chart-options>]  
[prestats=<bool>]  
[append=<bool>]  
[backfill=<bool>]  
[update_period=<integer>]  
[fillnull_value=<string>]  
[chunk_size=<unsigned int>]  
<stats-metric-term>...  
WHERE [<logical-expression>]...  
[(BY|GROUPBY) <field-list> ]  
[<span-length>]
```

Required arguments

<stats-metric-term>

Syntax: <stats-func> | <stats-func-value>

Description: Provides two options for performing statistical calculations on metrics. Use <stats-func> to perform statistical calculations on one or more metrics that you name in the argument. Use <stats-func-value> for cases where a wildcard can be used to represent several metrics. You cannot blend the <stats-func> syntax and the <stats-func-value> syntax in a single `mstats` search.

Use the <stats-func> syntax for most cases. You only need to use the <stats-func-value> syntax in cases where a single metric may be represented by several different metric names, such as `cpu.util` and `cpu.utilization`. In these cases you can apply a wildcard to catch all of the permutations of the `metric_name`.

See [Stats metric term options](#) for details on the <stats-func> and <stats-func-value> syntax options.

Optional arguments

append

Syntax: append=<bool>

Description: Valid only when `prestats=true`. This argument runs the `mstats` command and adds the results to an existing set of results instead of generating new results.

Default: false

backfill

Syntax: backfill=<bool>

Description: Valid only with real-time searches that have a time window. When `backfill=true`, the `mstats` command runs a search on historical data to backfill events before searching the in-memory real-time data.

Default: true

chart

Syntax: chart=<bool>

Description: When set to `chart=t`, the `mstats` data output has a format suitable for charting. The `mstats` charting mode is valid only when `prestats=f`.

When a `span` is provided, the `mstats` chart mode format resembles that of the `timechart` command, and can support at most one group-by field, which is used as the series splitting field.

When no `span` is provided, the chart mode follows a format similar to that of the `chart` or `timechart` commands. Without a `span`, the `mstats` chart mode requires one or two grouping fields. The first grouping field represents the chart x-axis. The second grouping field represents the y-axis and is a series split field.

Default: `chart=f`

<chart-options>

Syntax: `chart.limit | chart.agg | chart.usenull | chart.useother | chart.nullstr | chart.otherstr`

Description: Options that you can specify to refine the result. See the Chart options section in this topic.

`chunk_size`

Syntax: `chunk_size=<unsigned_int>`

Description: Advanced option. This argument controls how many **metric time series** are retrieved at a time from a single **time-series index file** (`.tsidx` file) when the Splunk software processes searches. Lower this setting from its default only when you find a particular `mstats` search is using too much memory, or when it infrequently returns events. This can happen when a search groups by excessively high-cardinality dimensions (dimensions with very large amounts of distinct values). In such situations, a lower `chunk_size` value can make `mstats` searches more responsive, but potentially slower to complete. A higher `chunk_size`, on the other hand, can help long-running searches to complete faster, with the potential tradeoff of causing the search to be less responsive. For `mstats`, `chunk_size` cannot be set lower than `10000`.

Default: `10000000` (10 million)

`fillnull_value`

Description: This argument sets a user-specified value that the `mstats` command substitutes for null values for any field within its group-by field list. Null values include field values that are missing from a subset of the returned events as well as field values that are missing from all of the returned events. If you do not provide a `fillnull_value` argument, `mstats` omits rows for events with one or more null field values from its results.

Default: empty string

<field-list>

Syntax: `<field>, ...`

Description: Specifies one or more fields to group the results by. Required when using the BY clause.

<logical-expression>

Syntax: `<time-opts>|<search-modifier>|((NOT)?`

`<logical-expression>)|<search-modifier>|<comparison-expression>|(<logical-expression> (OR)?`

`<logical-expression>)`

Description: An expression describing the filters that are applied to your search. Includes time and search modifiers, and comparison expressions. See the following sections for descriptions of each of these logical expression components.

Cannot filter on `metric_name`. Does not support CASE or TERM directives. You also cannot use the WHERE clause to search for terms or phrases.

`prestats`

Syntax: `prestats=true | false`

Description: Specifies whether to use the prestats format. The prestats format is a Splunk internal format that is designed to be consumed by commands that generate aggregate calculations. When you use the prestats format, you can pipe the data into the `chart`, `stats`, or `timechart` commands, which are designed to accept the prestats format. When `prestats` is set to `true`, instructions with the `AS` clause are not relevant. The field names for the aggregates are determined by the command that consumes the prestats format and produces the aggregate

output.

Default: false

<span-length>

Syntax: span=<int><timescale> [every=<int><timescale>]

Description: The span of each time bin. If used with a <timescale>, the <span-length> is treated as a time range. If not, this is an absolute bucket length. If you do not specify a <span-length>, the default is auto, which means that the number of time buckets adjusts to produce a reasonable number of results. For example, if seconds are used initially for the <timescale> and too many results are returned, the <timescale> is changed to a longer value, such as minutes, to return fewer time buckets.

To improve the performance of `mstats` searches you can optionally use the `every` argument in conjunction with `span` to cause the search to reduce the amount of data it samples per span. In other words you could design a search where the search head samples a `span` of only ten minutes of data for `every` hour covered by the search. See [Span length options](#).

update_period

Syntax: update_period=<integer>

Description: Valid only with real-time searches. Specifies how frequently, in milliseconds, the real-time summary for the `mstats` command is updated. A larger number means less frequent updates to the summary and less impact on index processing.

Default: 1000 (1 second)

Stats metric term options

<stats-func>

Syntax: <stats-func> | <mstats-specific-func> "("<metric_name>"") [AS <string>]...

Description: Perform statistical calculations on one or more `metric_name` fields. You can rename the result of each function using the `AS` clause, unless `prestats` is set to `true`. The `metric_name` must be enclosed in parenthesis.

When you use the <stats-func> syntax, the `WHERE` clause cannot filter on `metric_name`.

<mstats-specific-func>

Syntax: rate_avg | rate_sum

Description: Two functions that are specific to `mstats`. `rate_avg` computes the per **metric time series rates** for an accumulating **counter metric** and then returns the average of those rates. `rate_sum` does the same thing as `rate_avg` except that it returns the sum of the rates. For more about counter metrics and these functions see Investigate counter metrics in *Metrics*.

<stats-func-value>

Syntax: count(_value) | <function>(_value) [AS <string>] WHERE metric_name=<metric_name>

Description: Specify a basic count of the `_value` field or a function on the `_value` field. The `_value` field uses a specific format to store the numeric value of the metric. You can specify one or more functions. You can rename the result of the function using AS unless `prestats=true`.

When you use the <stats-func-value> syntax, the `WHERE` clause must filter on the `metric_name`. Wildcards are okay.

The stats-func-value syntax does not support real-time searches. If you must run a real-time search, use the stats-func syntax instead.

The following table lists the supported functions for the `mstats` command by type of function. Use the links in the table to see descriptions and examples for each function.

Type of function	Supported functions and syntax			
Aggregate functions	<code>avg()</code> <code>count()</code> <code>max()</code> <code>median()</code> <code>min()</code>	<code>perc<num></code> <code>range()</code> <code>stdev()</code> <code>stdevp()</code>	<code>sum()</code> <code>sumsq()</code> <code>upperperc<num></code> <code>var()</code> <code>varp()</code>	
Time functions	<code>earliest()</code> <code>earliest_time()</code> <code>latest()</code>	<code>latest_time()</code> <code>rate()</code>	<code>rate_avg()</code> <code>rate_sum()</code>	

For an overview of using functions with commands, see [Statistical and charting functions](#).

Chart options

`chart.limit`

Syntax: `chart.limit=(top | bottom)<int>`

Description: Only valid when a column-split is specified. Use the `chart.limit` option to specify the number of results that should appear in the output. When you set `chart.limit=N` the top or bottom N values are retained, based on the sum of each series and the prefix you have selected. If `chart.limit=0`, all results are returned. If you opt not to provide a `top` or `bottom` prefix before the `chart.limit` value, the Splunk software provides the top N results. For example, if you set `chart.limit=10` the Splunk software defaults to providing the top 10 results.

This argument is identical to the `limit` argument of the `chart` and `timechart` commands.

Default: `top10`

`chart.agg`

Syntax: `chart.agg=(<stats-func> (<evalued-field> | <wc-field>) [AS <wc-field>])`

Description: A statistical aggregation function. See the table of supported functions in Stats metric term options. The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names. This argument is identical to the `agg` argument of the `chart` and `timechart` commands.

Default: `sum`

`chart.nullstr`

Syntax: `chart.nullstr=<string>`

Description: If `chart.usenull` is true, this series is labeled by the value of the `chart.nullstr` option, and defaults to NULL. This argument is identical to the `nullstr` argument of the `chart` and `timechart` commands.

`chart.otherstr`

Syntax: `chart.otherstr=<string>`

Description: If `chart.useother` is true, this series is labeled by the value of the `code.otherstr` option, and defaults to OTHER. This argument is identical to the `otherstr` argument of the `chart` and `timechart` commands.

chart.usenull

Syntax: chart.usenull=<bool>

Description: Determines whether a series is created for events that do not contain the split-by field. This argument is identical to the `usenull` argument of the `chart` and `timechart` commands.

chart.useother

Syntax: chart.useother=<bool>

Description: Specifies whether a series should be added for data series not included in the graph because they did not meet the criteria of the WHERE clause. This argument is identical to the `useother` argument of the `chart` and `timechart` commands.

Logical expression options

<comparison-expression>

Syntax: <field><comparison-operator><value> | <field> IN (<value-list>)

Description: Compares a field to a literal value or provides a list of values that can appear in the field.

<search-modifier>

Syntax: <sourcetype-specifier> | <host-specifier> | <source-specifier> | <splunk_server-specifier>

Description: Search for events from specified fields. For example, search for one or a combination of hosts, sources, and source types. See searching with default fields in the *Knowledge Manager manual*.

<time-opt>

Syntax: [<timeformat>] (<time-modifier>)*

Description: Describes the format of the `<starttime>` and `<endtime>` terms of the search.

Comparison expression options

<comparison-operator>

Syntax: = | != | < | <= | > | >=

Description: Use comparison expressions when searching field-value pairs. Comparison expressions with the equal (=) or not equal (!=) operator compare string values. For example, "1" does not match "1.0".

Comparison expressions with greater than or less than operators < > <= >= numerically compare two numbers and lexicographically compare other values. See [Usage](#).

<field>

Syntax: <string>

Description: The name of a field.

<value>

Syntax: <literal-value>

Description: In comparison expressions, this is the literal number or string value of a field.

<value-list>

Syntax: (<literal-value>, <literal-value>, ...)

Description: Used with the IN operator to specify two or more values. For example use `error IN (400, 402, 404, 406)` instead of `error=400 OR error=402 OR error=404 OR error=406`.

Search modifier options

<sourcetype-specifier>

Syntax: sourcetype=<string>

Description: Search for events from the specified sourcetype field.

<host-specifier>

Syntax: host=<string>

Description: Search for events from the specified host field.

<source-specifier>

Syntax: source=<string>

Description: Search for events from the specified source field.

<splunk_server-specifier>

Syntax: splunk_server=<string>

Description: Search for events from a specific server. Use "local" to refer to the search head.

Span length options

every

Syntax: every=<int><timescale>

Description: Use in conjunction with `span` to search data in discrete time intervals over the full timespan of a search. The `every` argument is valid only when `span` is set to a valid value other than `auto`. Set the `every` timespan to a value that is greater than the `span` timespan.

This method of "downsampling" the search data improves search performance at the expense of data granularity. For example, this search returns an average of the `active_logins` measurement for the first ten seconds of every twenty seconds covered by the time range of the search: `| mstats avg(active_logins) span=10s every=20s`

Month intervals for `every` are exactly 30 days long. Year intervals for `every` are exactly 365 days long.

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month> | <subseconds>

Description: Time scale units.

Default: sec

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

`mstats` only supports subsecond timescales such as `ms` when it is searching metric indexes that are configured for millisecond timestamp resolution.

For more information about enabling metrics indexes to index metric data points with millisecond timestamp precision, see:

- Manage Splunk Cloud Platform indexes in the *Splunk Cloud Platform Admin Manual* if you use Splunk Cloud Platform.
- Create custom indexes in *Managing indexers and clusters of indexers* if you use Splunk Enterprise.

Time options

<timeformat>

Syntax: timeformat=<string>

Description: Set the time format for `starttime` and `endtime` terms.

Default: timeformat=%m/%d/%Y:%H:%M:%S.

For more about setting exact times with the available `timeformat` options, see [Date and time format variables](#). Subsecond options are only available if you are searching over a metrics index with millisecond timestamp resolution.

<time-modifier>

Syntax: starttime=<string> | endtime=<string> | earliest=<time_modifier> | latest=<time_modifier>

Description: Specify start and end times using relative or absolute time.

You can also use the `earliest` and `latest` arguments to specify absolute and relative time ranges for your search.

For more about the relative `<time_modifier>` syntax, see [Time modifiers](#).

For more information about setting absolute time ranges see [Date and time format variables](#). Subsecond options are only available if you are searching over a metrics index with millisecond timestamp resolution.

starttime

Syntax: starttime=<string>

Description: Events must be later or equal to this time. The `starttime` must match the `timeformat`.

endtime

Syntax: endtime=<string>

Description: All events must be earlier or equal to this time.

Usage

The `mstats` command is a **report-generating command**, except when `append=true`. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search, except when `append=true` is specified with the command.

Use the `mstats` command to search metrics data. The metrics data uses a specific format for the metrics fields. See Metrics data format in *Metrics*.

All metrics search commands are case sensitive. This means, for example, that `mstats` treats as the following as three distinct values of `metric_name`: `cap.gear`, `CAP.GEAR`, and `Cap.Gear`.

`mstats` searches cannot return results for metric data points with `metric_name` fields that are empty or which contain blank spaces.

Append `mstats` searches together

The `mstats` command does not support subsearches. You can use the `append` argument to add the results of an `mstats` search to the results of a preceding `mstats` search. See the topic on the `tstats` command for an `append` usage example.

Aggregations

If you are using the `<stats-func>` syntax, numeric aggregations are only allowed on specific values of the `metric_name` field. The metric name must be enclosed in parenthesis. If there is no data for the specified `metric_name` in parenthesis, the search is still valid.

If you are using the `<stats-func-value>` syntax, numeric aggregations are only allowed on the `_value` field.

Aggregations are not allowed for values of any other field, including the `_time` field.

When `prestats = true` and you run an `mstats` search that uses the `c` and `count` aggregation functions without an aggregation field, the Splunk software processes them as if they are actually `count(_value)`. In addition, any statistical functions that follow in the search string must reference the `_value` field. For example: `| mstats count | timechart count(_value)`

Wildcard characters

The `mstats` command supports wildcard characters in any search filter, with the following exceptions:

- You cannot use wildcard characters in the GROUP BY clause.
- If you are using the `<stats_func_value>` syntax, you cannot use wildcard characters in the `_value` field.
- If you are using wildcard characters in your aggregations and you are renaming them, your rename must have matching wildcards.

For example, this search is invalid:

```
| mstats sum(*.free) as FreeSum
```

This search is valid:

```
| mstats sum(*.free) as *FreeSum
```

- Real-time `mstats` searches cannot utilize wildcarded metric aggregations when you use the `<stats-func>` syntax.

For example, this search is invalid, when you set it up as a real-time search:

```
| mstats avg(cpu.*) max(cpu.*) where index=sysmetrics
```

This real-time search is valid:

```
| mstats avg(cpu.sys) max(cpu.usr) where index=sysmetrics
```

WHERE clause

Use the WHERE clause to filter by any of the supported dimensions.

If you are using the <stats-func> syntax, the WHERE clause cannot filter by metric_name. Filtering by metric_name is performed based on the metric_name fields specified with the <stats-func> argument.

If you are using the <stats-func-value> syntax, the WHERE clause must filter by metric_name.

If you do not specify an index name in the WHERE clause, the mstats command returns results from the default metrics indexes associated with your role. If you do not specify an index name and you have no default metrics indexes associated with your role, mstats returns no results. To search against all metrics indexes use WHERE index=*.

The WHERE clause must come before the BY or GROUPBY clause, if they are both used in conjunction with mstats.

For more information about defining default metrics indexes for a role, see Add and edit roles with Splunk Web in *Securing Splunk Enterprise*.

Group results by metric name and dimension

You can group results by the metric_name and dimension fields.

You can also group by time. You must specify a timespan using the <span-length> argument to group by time buckets. For example, span=1hr or span=auto. The <span-length> argument is separate from the BY clause and can be placed at any point in the search between clauses.

Grouping by the _value or _time fields is not allowed.

Group by metric time series

You can group results by **metric time series**. A metric time series is a set of **metric data points** that share the same metrics and the same dimension field-value pairs. Grouping by metric time series ensures that you are not mixing up data points from different metric data sources when you perform statistical calculations on them.

Use BY _timeseries to group by metric time series. The _timeseries field is internal and won't display in your results. If you want to display the _timeseries values in your search, add | rename _timeseries AS timeseries to the search.

For a detailed overview of the _timeseries field with examples, see Perform statistical calculations on metric time series in *Metrics*.

Time dimensions

The mstats command does not recognize the following time-related dimensions.

Unsupported dimensions		
date_hour	date_wday	timeendpos
date_mday	date_year	timestamp
date_minute	date_zone	timestartpos
date_month	metric_timestamp	
date_second	time	

Unsupported dimensions

Subsecond bin time spans

You can only use subsecond `span` timescales—time spans that are made up of deciseconds (ds), centiseconds (cs), milliseconds (ms), or microseconds (us)—for `mstats` searches over metrics indexes that have been configured to have millisecond timestamp resolution.

Subsecond `span` timescales should be numbers that divide evenly into a second. For example, `1s = 1000ms`. This means that valid millisecond `span` values are 1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, or 500ms. In addition, `span = 1000ms` is not allowed. Use `span = 1s` instead.

For more information about giving indexes millisecond timestamp resolution:

- For Splunk Cloud Platform: See Manage Splunk Cloud Platform indexes in the *Splunk Cloud Platform Admin Manual*.
- For Splunk Enterprise: See Create custom indexes in *Managing indexes and clusters of indexes*.

Search over a set of indexes with varying levels of timestamp resolution

If you run an `mstats` search over multiple metrics indexes with varying levels of timestamp resolution, the results of the search may contain results with timestamps of different resolutions.

For example, say you have two metrics indexes. Your "metrics-second" metrics index has a second timestamp resolution. Your "metrics-ms" metrics index has a millisecond timestamp resolution. You run the following search over both indexes:

```
| mstats count(*) WHERE index=metric* span=100ms.
```

The search produces the following results:

_time	count(cpu.nice)
1549496110	48
1549496110.100	2

The `11549496110` row counts results from both indexes. The count from "metric-ms" includes only metric data points with timestamps from `1549496110.000` to `1549496110.099`. The "metric-ms" metric data points with timestamps from `1549496110.100` to `1549496110.199` appear in the `1549496110.100` row.

Meanwhile, the metric data points in the "metric-second" index do not have millisecond timestamp precision. The `1549496110` row only counts those "metric-second" metric data points with the `11549496110` timestamp, and no metric data points from "metric-second" are counted in the `1549496110.100` row.

Time bin limits for mstats search jobs

Splunk software regulates `mstats` search jobs that use `span` or a similar method to group results by time. When Splunk software processes these jobs, it limits the number of "time bins" that can be allocated within a single `.tsidx` file.

For metrics indexes with second timestamp resolution, this only affects searches with large time ranges and very small time spans, such as a search over a year with `span = 1s`. If you are searching on a metrics index with millisecond timestamp resolution, you might encounter this limit over shorter ranges, such as a search over an hour with `span = 1ms`.

This limit is set by `time_bin_limit` in `limits.conf`, which is set to 1 million bins by default. If you need to run these kinds of `mstats` search jobs, lower this value if they are using too much memory per search. Raise this value if these kinds of search jobs are returning errors.

The Splunk platform estimates the number of time bins that a search requires by dividing the search time range by its group-by span. If this produces a number that is larger than the `time_bin_limit`, the Splunk platform returns an error.

The search time range is determined by the `earliest` and `latest` values of the search. Some kinds of searches—such as all-time searches—do not have `earliest` and `latest`. In such cases the Splunk platform checks within each single TSIDX file to derive a time range for the search.

Metrics indexes have second timestamp resolution by default. You can give a metrics index a millisecond timestamp resolution when you create it, or you can edit an existing metrics index to switch it to millisecond timestamp resolution.

If you use Splunk Cloud, see Manage Splunk Cloud Platform indexes in the *Splunk Cloud Platform Admin Manual*. If you use Splunk Enterprise, see Create custom indexes in *Managing indexes and clusters of indexes*.

Memory and mstats search performance

A pair of `limits.conf` settings strike a balance between the performance of `mstats` searches and the amount of memory they use during the search process, in RAM and on disk. If your `mstats` searches are consistently slow to complete you can adjust these settings to improve their performance, but at the cost of increased search-time memory usage, which can lead to search failures.

If you use Splunk Cloud Platform, you will need to file a Support ticket to change these settings.

For more information, see Memory and stats search performance in the *Search Manual*.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

You can specify a custom sort order that overrides the lexicographical order. See the blog Order Up! Custom Sort Orders.

Examples

1. Calculate a single metric grouped by time

Return the average value of the `aws.ec2.CPUUtilization` metric in the `mymetricdata` metric index. Bucket the results into 30 second time spans.

```
| mstats avg(aws.ec2.CPUUtilization) WHERE index=mymetricdata span=30s
```

2. Combine metrics with different metric names

Return the average value of both the `aws.ec2.CPUUtilization` metric and the `os.cpu.utilization` metric. Group the results by host and bucket the results into 1 minute time spans. Both metrics are combined and considered a single metric series.

```
| mstats avg(aws.ec2.CPUUtilization) avg(os.cpu.utilization) WHERE index=mymetricdata BY host span=1m
```

3. Use chart=t mode to chart metric event counts by the top ten hosts

Return a chart of the number of `aws.ec2.CPUUtilization` metric data points for each day, split by the top ten hosts.

```
| mstats chart=t count(aws.ec2.CPUUtilization) WHERE index=mymetricdata by host span=1d chart.limit=top10
```

4. Filter the results on a dimension value and split by the values of another dimension

Return the average value of the `aws.ec2.CPUUtilization` metric for all measurements with `host=www2` and split the results by the values of the `app` dimension.

```
| mstats avg(aws.ec2.CPUUtilization) WHERE host=www2 BY app
```

5. Specify multiple aggregations of multiple metrics

Return the average and maximum of the resident set size and virtual memory size. Group the results by `metric_name` and bucket them into 1 minute spans

```
| mstats avg(os.mem.rss) AS "AverageRSS" max(os.mem.rss) AS "MaxRSS" avg(os.mem.vsz) AS "AverageVMS" max(os.mem.vsz) AS "MaxVMS" WHERE index=mymetricdata BY metric_name span=1m
```

6. Aggregate a metric across all of your default metrics indexes, using downsampling to speed up the search

Find the median of the `aws.ec2.CPUUtilization` metric. Do not include an index filter to search for measurements in all of the default metrics indexes associated with your role. Speed up the search by using `every` to compute the median for one minute of every five minutes covered by the search.

```
| mstats median(aws.ec2.CPUUtilization) span=1m every=5m
```

7. Get the rate of an accumulating counter metric and group the results by time series

See Perform statistical calculations on metric time series in *Metrics* for more information.

```
| mstats rate(spl.intr.resource_usage.PerProcess.data.elapsed) as data.elapsed WHERE index=_metrics BY _timeseries | rename _timeseries AS timeseries
```

8. Stats-func-value example

Use the `<stats-func-value>` syntax to get a count of all of the measurements for the `aws.ec2.CPUUtilization` metric in the `mymetricdata` index.

```
| mstats count(_value) WHERE metric_name=aws.ec2.CPUUtilization AND index=mymetricdata
```

See also

Related information

Overview of metrics in Metrics

multikv

Description

Extracts field-values from table-formatted search results, such as the results of the `top`, `tstat`, and so on. The `multikv` command creates a new event for each table row and assigns field names from the title row of the table.

An example of the type of data the `multikv` command is designed to handle:

Name	Age	Occupation
Josh	42	SoftwareEngineer
Francine	35	CEO
Samantha	22	ProjectManager

The key properties here are:

- Each line of text represents a conceptual record.
- The columns are aligned.
- The first line of text provides the names for the data in the columns.

The `multikv` command can transform this table from one event into three events with the relevant fields. It works more easily with the fixed-alignment though can sometimes handle merely ordered fields.

The general strategy is to identify a header, offsets, and field counts, and then determine which components of subsequent lines should be included into those field names. Multiple tables in a single event can be handled (if `multitable=true`), but might require ensuring that the secondary tables have capitalized or ALLCAPS names in a header row.

Auto-detection of header rows favors rows that are text, and are ALLCAPS or Capitalized.

For Splunk Cloud Platform, you must create a private app to extract field-value pairs from table-formatted search results. If you are a Splunk Cloud administrator with experience creating private apps, see [Manage private apps](#) in your Splunk Cloud deployment in the Splunk Cloud Admin Manual. If you have not created private apps, contact your Splunk account representative for help with this customization.

Syntax

`multikv [conf=<stanza_name>] [<multikv-option>...]`

Optional arguments

`conf`

Syntax: `conf=<stanza_name>`

Description: If you have a field extraction defined in `multikv.conf`, use this argument to reference the stanza in your search. For more information, refer to the configuration file reference for `multikv.conf` in the *Admin Manual*.

<multikv-option>

Syntax: `copyattrs=<bool> | fields <field-list> | filter <term-list> | forceheader=<int> | multitable=<bool> | noheader=<bool> | rmorig=<bool>`

Description: Options for extracting fields from tabular events.

Descriptions for multikv options

copyattrs

Syntax: `copyattrs=<bool>`

Description: When true, `multikv` copies all fields from the original event to the events generated from that event. When false, no fields are copied from the original event. This means that the events will have no `_time` field and the UI will not know how to display them.

Default: true

fields

Syntax: `fields <field-list>`

Description: Limit the fields set by the `multikv` extraction to this list. Ignores any fields in the table which are not on this list.

filter

Syntax: `filter <term-list>`

Description: If specified, `multikv` skips over table rows that do not contain at least one of the strings in the filter list. Quoted expressions are permitted, such as "multiple words" or "trailing_space".

forceheader

Syntax: `forceheader=<int>`

Description: Forces the use of the given line number (1 based) as the table's header. Does not include empty lines in the count.

Default: The `multikv` command attempts to determine the header line automatically.

multitable

Syntax: `multitable=<bool>`

Description: Controls whether or not there can be multiple tables in a single `_raw` in the original events.

Default: true

noheader

Syntax: `noheader=<bool>`

Description: Handle a table without header row identification. The size of the table will be inferred from the first row, and fields will be named `Column_1`, `Column_2`, ... `noheader=true` implies `multitable=false`.

Default: false

rmorig

Syntax: `rmorig=<bool>`

Description: When true, the original events will not be included in the output results. When false, the original events are retained in the output results, with each original emitted after the batch of generated results from that original.

Default: true

Usage

The `multikv` command is a distributable streaming command. See [Command types](#).

Examples

Example 1: Extract the "COMMAND" field when it occurs in rows that contain "splunkd".

```
... | multikv fields COMMAND filter splunkd
```

Example 2: Extract the "pid" and "command" fields.

```
... | multikv fields pid command
```

See also

[extract](#), [kvform](#), [rex](#), [spath](#), [xmlkv](#),

multisearch

Description

The `multisearch` command is a generating command that runs multiple *streaming* searches at the same time. This command requires at least two subsearches and allows only streaming operations in each subsearch. Examples of streaming searches include searches with the following commands: `search`, `eval`, `where`, `fields`, and `rex`. For more information, see [Types of commands](#) in the *Search Manual*.

Syntax

```
| multisearch <subsearch1> <subsearch2> <subsearch3> ...
```

Required arguments

<subsearch>

Syntax: `"[>search <logical-expression>]"`

Description: At least two streaming searches must be specified. See the [search](#) command for detailed information about the valid arguments for `<logical-expression>`.

To learn more, see [About subsearches](#) in the *Search Manual*.

Usage

The `multisearch` command is an event-generating command. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

The multisearch command doesn't support peer selection

You can't exclude **search peers** from `multisearch` searches because the `multisearch` command connects to all peers by default. For example, the following `multisearch` search connects to the indexer called `myServer` even though it is excluded using `NOT`:

```
| multisearch [ search index=_audit NOT splunk_server=myServer]
```

Instead of using the `multisearch` command to exclude search peers from your search, you can use other commands such as `append` with search optimization turned off. If you don't turn off search optimization, Splunk software might internally convert the `append` command to the `multisearch` command in order to optimize the search and might not exclude the search peers.

You can turn off search optimization for a specific search by including the following command at the end of your search:

```
| noop search_optimization=false
```

For example, the following workaround uses the `append` command to exclude `myServer`:

```
index=_internal splunk_server=myServer | append[| search index=_audit] | noop search_optimization=false
```

See Optimization settings in the *Search Manual*.

Subsearch processing and limitations

With the `multisearch` command, the events from each subsearch are interleaved. Therefore the `multisearch` command is not restricted by the subsearch limitations.

Unlike the `append` command, the `multisearch` command does not run the subsearch to completion first. The following subsearch example with the `append` command is not the same as using the `multisearch` command.

```
index=a | eval type = "foo" | append [search index=b | eval mytype = "bar"]
```

Examples

Example 1:

Search for events from both index a and b. Use the `eval` command to add different fields to each set of results.

```
| multisearch [search index=a | eval type = "foo"] [search index=b | eval mytype = "bar"]
```

See also

[append](#), [join](#)

mvcombine

Description

Takes a group of events that are identical except for the specified field, which contains a single value, and combines

those events into a single event. The specified field becomes a multivalue field that contains all of the single values from the combined events.

The mvcombine command does not apply to internal fields.

See Use default fields in the *Knowledge Manager Manual*.

Syntax

`mvcombine [delim=<string>] <field>`

Required arguments

field

Syntax: `<field>`

Description: The name of a field to merge on, generating a multivalue field.

Optional arguments

delim

Syntax: `delim=<string>`

Description: Defines the string to use as the delimiter for the values that get combined into the multivalue field. For example, if the values of your field are "1", "2", and "3", and `delim` is a semi-colon (;), then the combined multivalue field is "1";"2";"3".

Default: a single space, (" ")

To see the output of the `delim` argument, you must use the `nomv` command immediately after the `mvcombine` command. See Usage

Usage

The `mvcombine` command is a **transforming command**. See [Command types](#).

You can use [evaluation functions](#) and [statistical functions](#) on multivalue fields or to return multivalue fields.

The `mvcombine` command accepts a set of input results and finds groups of results where all field values are identical, except the specified field. All of these results are merged into a single result, where the specified field is now a multivalue field.

Because raw events have many fields that vary, this command is most useful after you reduce the set of available fields by using the `fields` command. The command is also useful for manipulating the results of certain transforming commands, like `stats` or `timechart`.

Specifying delimiters

The `mvcombine` command creates a multivalue version of the field you specify, as well as a single value version of the field. The multivalue version is displayed by default.

The single value version of the field is a flat string that is separated by a space or by the delimiter that you specify with the `delim` argument.

By default the multivalue version of the field is displayed in the results. To display the single value version with the delimiters, add the `| nomv` command to the end of your search. For example `... | mvcombine delim= "," host | nomv host.`

Some modes of search result investigation prefer this single value representation, such as exporting to CSV in the UI, or running a command line search with `splunk search "..." -output csv`. Some commands that are not `multivalue` aware might use this single value as well.

Most ways of accessing the search results prefer the multivalue representation, such as viewing the results in the UI, or exporting to JSON, requesting JSON from the command line search with `splunk search "..." -output json` or requesting JSON or XML from the REST API. For these forms of, the selected `delim` has no effect.

Other ways of turning multivalue fields into single-value fields

If your primary goal is to convert a multivalue field into a single-value field, `mvcombine` is probably not your best option. `mvcombine` is mainly meant for the creation of new multivalue fields. Instead, try either the `nomv` command or the `mvjoin eval` function.

Conversion option	Description	For more information
<code>nomv</code> command	Use for simple multivalue field to single-value field conversions. Provide the name of a multivalue field in your search results and <code>nomv</code> will convert each instance of the field into a single-value field.	nomv
<code>mvjoin eval</code> function	Use when you want to perform multivalue field to single-value field conversion where the former multivalues are separated by a delimiter that you supply. For example, you start with a multivalue field that contains the values 1, 2, 3,4, 5. You can use <code>mvjoin</code> to transform your multivalue field into a single-valued field with <code>OR</code> as the delimiter. The new single value of the field is 1 OR 2 OR 3 OR 4 OR 5.	Multivalue eval functions

Examples

1. Creating a multivalue field

This example uses the sample dataset from the Search Tutorial. To try this example yourself, download the data set from Get the tutorial data into Splunk and follow the instructions in the Search Tutorial to upload the data.

To understand how `mvcombine` works, let's explore the data.

1. Set the time range to **All time**.
2. Run the following search.

```
index=* | stats max(bytes) AS max, min(bytes) AS min BY host
```

The results show that the **max** and **min** fields have duplicate entries for the hosts that start with `www`. The other hosts show no results for the **max** and **min** fields.

The screenshot shows the Splunk search interface with the following details:

- Search Bar:** index=* | stats max(bytes) as max, min(bytes) as min by host
- Time Range:** All time
- Panel Buttons:** Save As, Close, Job, Smart Mode
- Event Types:** Events, Patterns, Statistics (5), Visualization
- Table Headers:** host, max, min
- Data Rows:**

host	max	min
mailsv		
vendor_sales		
www1	4000	200
www2	4000	200
www3	4000	200

3. To remove the other hosts from your results, modify the search to add `host=www*` to the search criteria.

```
index=* host=www* | stats max(bytes) AS max, min(bytes) AS min BY host
```

Because the values in the `max` and `min` columns contain the exact same values, you can use the `mvcombine` to combine the host values into a multivalue result.

4. Add `| mvcombine host` to your search and run the search again.

```
index=* host=www* | stats max(bytes) AS max, min(bytes) AS min BY host | mvcombine host
```

Instead of three rows, one row is returned. The host field is now a multivalue field.

The screenshot shows the Splunk search interface with the following details:

- Search Bar:** index=* host=www* | stats max(bytes) as max, min(bytes) as min by host | mvcombine host
- Time Range:** All time
- Panel Buttons:** Save As, Close, Job, Smart Mode
- Event Types:** Events, Patterns, **Statistics (1)**, Visualization
- Table Headers:** host, max, min
- Data Rows:**

host	max	min
www1 www2 www3	4000	200

2. Returning the delimited values

As mentioned in the [Usage](#) section, by default the delimited version of the results are not returned in the output. To return the results with the delimiters, you must return the single value string version of the field.

Add the `nomv` command to your search. For example:

```
index=* host=www* | stats max(bytes) AS max, min(bytes) AS min BY host | mvcombine delim="," host | nomv host
```

The search results that are returned are shown in the following table.

host	max	min
www1, www2, www3	4000	200

To return the results with a space after each comma, specify `delim=" ", "`.

Example 3:

In multivalue events:

```
sourcetype="WMI:WinEventLog:Security" | fields EventCode, Category,RecordNumber | mvcombine delim="," RecordNumber | nomv RecordNumber
```

Example 4:

Combine the values of "foo" with a colon delimiter.

```
... | mvcombine delim=":" foo
```

See also

Commands:

[makemv](#)
[mvexpand](#)
[nomv](#)

Functions:

[Multivalue eval functions](#)
[Multivalue stats and chart functions](#)
[split](#)

mvexpand

Description

Expands the values of a multivalue field into separate events, one event for each value in the multivalue field. For each result, the `mvexpand` command creates a new result for every multivalue field.

The `mvexpand` command can't be applied to internal fields.

See Use default fields in the *Knowledge Manager Manual*.

Syntax

```
mvexpand <field> [limit=<int>]
```

Required arguments

field

Syntax: <field>

Description: The name of a multivalue field.

Optional arguments

limit

Syntax: limit=<int>

Description: Specify the number of values of <field> to use for each input event.

Default: 0, or no limit

Usage

The `mvexpand` command is a distributable streaming command. See [Command types](#).

You can use [evaluation functions](#) and [statistical functions](#) on multivalue fields or to return multivalue fields.

Limits

A limit exists on the amount of RAM that the `mvexpand` command is permitted to use while expanding a batch of results. By default the limit is 500MB. The input chunk of results is typically `maxresultrows` or smaller in size, and the expansion of all these results resides in memory at one time. The total necessary memory is the average result size multiplied by the number of results in the chunk multiplied by the average size of the multivalue field being expanded.

If this attempt exceeds the configured maximum on any chunk, the chunk is truncated and a warning message is emitted. If you have Splunk Enterprise, you can adjust the limit by editing the `max_mem_usage_mb` setting in the `limits.conf` file.

Prerequisites

- Have the permissions to increase the `maxresultrows` and `max_mem_usage_mb` settings. Only users with file system access, such as system administrators, can increase the `maxresultrows` and `max_mem_usage_mb` settings using configuration files.
- Know how to edit configuration files. Review the steps in [How to edit a configuration file](#) in the [Splunk Enterprise Admin Manual](#).
- Decide which directory to store configuration file changes in. There can be configuration files with the same name in your default, local, and app directories. See [Where you can place \(or find\) your modified configuration files](#) in the [Splunk Enterprise Admin Manual](#).

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

If you use Splunk Cloud Platform and encounter problems because of this limit, file a Support ticket.

Examples

Example 1:

Create new events for each value of multivalue field, "foo".

```
... | mvexpand foo
```

Example 2:

Create new events for the first 100 values of multivalue field, "foo".

```
... | mvexpand foo limit=100
```

Example 3:

The `mvexpand` command only works on one multivalue field. This example walks through how to expand an event with more than one multivalue field into individual events for each field value. For example, given these events, with `sourcetype=data`:

```
2018-04-01 00:11:23 a=22 b=21 a=23 b=32 a=51 b=24  
2018-04-01 00:11:22 a=1 b=2 a=2 b=3 a=5 b=2
```

First, use the [rex command](#) to extract the field values for a and b. Then use the [eval command](#) and [mvzip function](#) to create a new field from the values of a and b.

```
source="mvexpandData.csv" | rex field=_raw "a=(?<a>\d+)" max_match=5 | rex field=_raw "b=(?<b>\d+)" max_match=5 | eval fields = mvzip(a,b) | table _time fields
```

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

_time	fields
2018-04-01 00:11:23	22,21 23,32 51,24
2018-04-01 00:11:22	1,2 2,3 5,2

Use the `mvexpand` command and the [rex command](#) on the new field, **fields**, to create new events and extract the alpha and beta values:

```
source="mvexpandData.csv" | rex field=_raw "a=(?<a>\d+)" max_match=5 | rex field=_raw "b=(?<b>\d+)" max_match=5 | eval fields = mvzip(a,b) | mvexpand fields | rex field=fields "(?<alpha>\d+), (?<beta>\d+)" | table _time alpha beta
```

Use the [table command](#) to display only the `_time`, `alpha`, and `beta` fields in a results table.

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

_time	alpha	beta
2018-04-01 00:11:23	23	32
2018-04-01 00:11:23	51	24
2018-04-01 00:11:22	1	2
2018-04-01 00:11:22	2	3

_time	alpha	beta
2018-04-01 00:11:22	5	2

(Thanks to Splunk user Duncan for this example.)

See also

Commands:

[makemv](#)
[mvcombine](#)
[nomv](#)

Functions:

[Multivalue eval functions](#)
[Multivalue stats and chart functions](#)
[split](#)

nomv

Description

Converts values of the specified multivalue field into one single value. Separates the values using a new line "\n" delimiter.

Overrides the configurations for the multivalue field that are set in the `fields.conf` file.

Syntax

`nomv <field>`

Required arguments

`field`

Syntax: `<field>`

Description: The name of a multivalue field.

Usage

The `nomv` command is a distributable streaming command. See [Command types](#).

You can use [evaluation functions](#) and [statistical functions](#) on multivalue fields or to return multivalue fields.

Examples

Example 1:

For sendmail events, combine the values of the senders field into a single value. Display the top 10 values.

```
eventtype="sendmail" | nomv senders | top senders
```

See also

Commands:

[makemv](#)
[mvcombine](#)
[mvexpand](#)
[convert](#)

Functions:

[Multivalue eval functions](#)
[Multivalue stats and chart functions](#)
[split](#)

outlier

Description

This command is used to remove outliers, not detect them. It removes or truncates outlying numeric values in selected fields. If no fields are specified, then the `outlier` command attempts to process all fields.

To identify outliers and create alerts for outliers, see finding and removing outliers in the *Search Manual*.

Syntax

`outlier <outlier-options>... [<field-list>]`

Optional arguments

<outlier-options>

Syntax: <action> | <mark> | <param> | <uselower>
Description: Outlier options.

<field-list>

Syntax: <field> ...
Description: A space-delimited list of field names.

Outlier options

<action>

Syntax: action=remove | transform
Description: Specifies what to do with the outliers. The `remove` option removes events that contain the outlying numerical values. The `transform` option truncates the outlying values to the threshold for outliers. If `action=transform` and `mark=true`, prefixes the values with "000".
Abbreviations: The `remove` action can be shorted to `rm`. The `transform` action can be shorted to `tf`.
Default: transform

<mark>

Syntax: mark=<bool>
Description: If `action=transform` and `mark=true`, prefixes the outlying values with "000". If `action=remove`, the `mark` argument has no effect.

Default: false

<param>

Syntax: param=<num>

Description: Parameter controlling the threshold of outlier detection. An outlier is defined as a numerical value that is outside of `param` multiplied by the inter-quartile range (IQR).

Default: 2.5

<uselower>

Syntax: uselower=<bool>

Description: Controls whether to look for outliers for values below the median in addition to above.

Default: false

Usage

The `outlier` command is a dataset processing command. See [Command types](#).

Filtering is based on the inter-quartile range (IQR), which is computed from the difference between the 25th percentile and 75th percentile values of the numeric fields. If the value of a field in an event is less than `(25th percentile) - param*IQR` or greater than `(75th percentile) + param*IQR`, that field is transformed or that event is removed based on the `action` parameter.

Examples

Example 1: For a timechart of webserver events, transform the outlying average CPU values.

```
404 host="webserver" | timechart avg(cpu_seconds) by host | outlier action=tf
```

Example 2: Remove all outlying numerical values.

```
... | outlier
```

See also

[anomalies](#), [anomalousvalue](#), [cluster](#), [kmeans](#)

Finding and removing outliers

outputcsv

Description

If you have Splunk Enterprise, this command saves search results to the specified CSV file on the local search head in the `$SPLUNK_HOME/var/run/splunk/csv` directory. Updates to `$SPLUNK_HOME/var/run/*.csv` using the `outputcsv` command are not replicated across the cluster.

If you have Splunk Cloud Platform, you cannot use this command. Instead, you have these options:

- Export search results using Splunk Web. See Export data using Splunk Web in the [Search Manual](#).
- Export search results using REST API. See Export data using the REST APIs in the [Search Manual](#).

- Create an alert action that includes a CSV file as an email attachment. See Email notification action in the *Alerting Manual*.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

```
outputcsv [append=<bool>] [create_empty=<bool>] [override_if_empty=<bool>] [dispatch=<bool>] [usexml=<bool>]
[singlefile=<bool>] [<filename>]
```

Optional arguments

append

Syntax: append=<bool>

Description: If `append` is true, the command attempts to append to an existing CSV file, if the file exists. If the CSV file does not exist, a file is created. If there is an existing file that has a CSV header already, the command only emits the fields that are referenced by that header. The command cannot append to .gz files.

Default: false

create_empty

Syntax: create_empty=<bool>

Description: If set to `true` and there are no results, a zero-length file is created. When set to `false` and there are no results, no file is created. If the file previously existed, the file is deleted.

Default: false

dispatch

Syntax: dispatch=<bool>

Description: If set to `true`, refers to a file in the job directory in `$SPLUNK_HOME/var/run/splunk/dispatch/<job id>/`.

filename

Syntax: <filename>

Description: Specify the name of a CSV file to write the search results to. This file should be located in `$SPLUNK_HOME/var/run/splunk/csv`. Directory separators are not permitted in the filename. If no filename is specified, the command rewrites the contents of each result as a CSV row into the `_xml` field. Otherwise the command writes into a file. The `.csv` file extension is appended to the filename if the filename has no file extension.

override_if_empty

Syntax: override_if_empty=<bool>

Description: If `override_if_empty=true` and no results are passed to the output file, the existing output file is deleted. If `override_if_empty=false` and no results are passed to the output file, the command does not delete the existing output file.

Default: true

singlefile

Syntax: singlefile=<bool>

Description: If `singlefile` is set to `true` and the output spans multiple files, collapses it into a single file.

Default: true

usexml

Syntax: usexml=<bool>

Description: If there is no filename, specifies whether or not to encode the CSV output into XML. This option should not be used when invoking the `outputcsv` from the UI.

Usage

There is no limit to the number of results that can be saved to the CSV file.

Internal fields and the `outputcsv` command

When the `outputcsv` command is used there are internal fields that are automatically added to the CSV file. The internal fields that are added to the output in the CSV file are:

- `_raw`
- `_time`
- `_indextime`
- `_serial`
- `_sourcetype`
- `_subsecond`

To exclude internal fields from the output, use the `fields` command and specify the fields that you want to exclude. For example:

```
... | fields - _indextime _sourcetype _subsecond _serial | outputcsv MyTestCsvFile
```

Multivalued fields

The `outputcsv` command merges values in a multivalued field into single space-delimited value.

Distributed deployments

The `outputcsv` command is not compatible with **search head pooling** and **search head clustering**.

The command saves the `*.csv` file on the local search head in the `$SPLUNK_HOME/var/run/splunk/` directory. The `*.csv` files are not replicated on the other search heads.

Examples

1. Output search results to a CSV file

Output the search results to the `mysearch.csv` file. The CSV file extension is automatically added to the file name if you don't specify the extension in the search.

```
... | outputcsv mysearch
```

2. Add a dynamic timestamp to the file name

You can add a timestamp to the file name by using a subsearch.

```
... | outputcsv [stats count | eval search=strftime(now(), "mysearch-%y%m%d-%H%M%S.csv") ]
```

3. Exclude internal fields from the output CSV file

You can exclude unwanted internal fields from the output CSV file. In this example, the fields to exclude are `_indexetime`, `_sourcetype`, `_subsecond`, and `_serial`.

```
index=_internal sourcetype="splunkd" | head 5 | fields _raw _time | fields - _indexetime _sourcetype  
_subsecond _serial | outputcsv MyTestCsvfile
```

4. Do not delete the CSV file if no search results are returned

Output the search results to the `mysearch.csv` file if results are returned from the search. Do not delete the `mysearch.csv` file if no results are returned.

```
... | outputcsv mysearch.csv override_if_empty=false
```

See also

[inputcsv](#)

outputlookup

Description

Writes search results to a static lookup table, or KV store collection, that you specify.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

The required syntax is in **bold**.

```
| outputlookup  
[append=<bool>]  
[create_empty=<bool>]  
[override_if_empty=<bool>]  
[max=<int>]  
[key_field=<field>]  
[createinapp=<bool>]  
[create_context=<string>]  
[output_format=<string>]  
<filename> | <tablename>
```

Required arguments

You must specify one of the following required arguments, either `filename` or `tablename`.

`filename`

Syntax: <string>

Description: The name of the lookup file. The file must end with `.csv` or `.csv.gz`.

`tablename`

Syntax: <string>

Description: The name of the lookup table as specified by a stanza name in `transforms.conf`, which corresponds to the lookup definition. The lookup table can be configured for any lookup type (CSV, external, or KV store).

If your lookup file and the lookup definition that it is associated with have the same name, you can provide a `tablename` that is the same value as the corresponding `filename` without the `.csv` extension. For example, say you have a lookup file named `staff.csv`. If you associate that file with a lookup called `staff`, you can use either `staff.csv` or `staff` as the `tablename` with the `outputlookup` command. See Create a CSV lookup definition in the Splunk Enterprise *Knowledge Manager Manual*.

Optional arguments

`append`

Syntax: `append=<bool>`

Description: The default setting, `append=false`, writes the search results to the `.csv` file or KV store collection. Fields that are not in the current search results are removed from the file. If `append=true`, the `outputlookup` command attempts to append search results to an existing `.csv` file or KV store collection. Otherwise, it creates a file. If there is an existing `.csv` file, the `outputlookup` command writes only the fields that are present in the previously existing `.csv` file. An `outputlookup` search that is run with `append=true` might result in a situation where the lookup table or collection is only partially updated. This means that a subsequent `lookup` or `inputlookup` search on that lookup table or collection might return stale data along with new data. The `outputlookup` command cannot append to `.gz` files.

Default: `false`

`create_context`

Syntax: `create_context= app | user | system`

Description: Specifies where the lookup table file is created. Ignored in favor of the `createinapp` argument if both arguments are used in the search. See [Usage](#) for details.

Default: `app`

`create_empty`

Syntax: `create_empty=<bool>`

Description: If set to `true` and there are no results, a zero-length file is created. When set to `false` and there are no results, no file is created. If the file previously existed, the file is deleted.

For example, suppose there is a system-level lookup called "test" with the lookup defined in "test.csv". There is also an app-level lookup with the same name. If an app overrides that "test.csv" in its own app directory with an empty file `create_empty=true`, the app-level lookup behaves as if the lookup is empty. However, if there's no file at all `create_empty=false` at the app level, then the lookup file in the system-level is used.

Default: `false`

`createinapp`

Syntax: `createinapp=<bool>`

Description: Specifies whether the lookup table file is created in the system directory or the lookups directory for the current app context. Overrides the `create_context` argument if both arguments are used in the search. See [Usage](#) for details.

Default: true

`key_field`

Syntax: `key_field=<field>`

Description: For KV store-based lookups, uses the specified field name as the key to a value and replaces that value. An `outputlookup` search using the `key_field` argument might result in a situation where the lookup table or collection is only partially updated. A subsequent `lookup` or `inputlookup` search on that collection might return stale data along with new data. A partial update only occurs with concurrent searches, one with the `outputlookup` command and a search with the `inputlookup` command. It is possible that the `inputlookup` occurs when the `outputlookup` is still updating some of the records.

When `key_field` is used in an `outputlookup` search, by default, `append` is set to `true`, which appends search results to an existing KV store collection. You can override this default behavior by directly setting `key_field` with `append` set to `false`.

`max`

Syntax: `max=<int>`

Description: The number of rows to output. Include the `max` argument in the `outputlookup` command to set a limit for a CSV file or change the limit for a KV store collection. The default number of rows to output to a KV store collection is 50000 and is controlled by the `max_rows_per_query` setting in the `limits.conf` file.

Default: no limit for a CSV file, 50000 for a KV store.

`output_format`

Syntax: `output_format=splunk_sv_csv | splunk_mv_csv`

Description: Controls the output data format of the lookup. Use `output_format=splunk_mv_csv` when you want to output multivalued fields to a lookup table file, and then read the fields back into Splunk using the `inputlookup` command. The default, `splunk_sv_csv` outputs a CSV file which excludes the `_mv_<fieldname>` fields.

Default: `splunk_sv_csv`

`override_if_empty`

Syntax: `override_if_empty=<bool>`

Description: If `override_if_empty=true` and no results are passed to the output file, the existing output file is deleted. If `override_if_empty=false` and no results are passed to the output file, the command does not delete the existing output file.

Default: true

Usage

The lookup table must be a CSV or GZ file, or a table name specified with a lookup table configuration in `transforms.conf`. The lookup table can refer to a KV store collection or a CSV lookup. The `outputlookup` command cannot be used with external lookups.

Determine where the lookup table file is created

For CSV lookups, `outputlookup` creates a lookup table file for the results of the search. There are three locations where `outputlookup` can put the file it creates:

- The system lookups directory: `$SPLUNK_HOME/etc/system/local/lookups`
- The lookups directory for the current app context: `$SPLUNK_HOME/etc/apps/<app>/lookups`
- The app-based lookups directory for the user running the search: `etc/users/<user>/<app>/lookups`

You can use the `createinapp` or `create_context` arguments to determine where `outputlookup` creates the lookup table for a given search. If you try to use both of these arguments in the same search, `createinapp` argument overrides the `create_context` argument.

If you do not use either argument in your search, the `create_context` setting in `limits.conf` determines where `outputlookup` creates the lookup table file. This setting defaults to `app` if there is an app context when you run the search, or to `system`, if there is not an app context when you run the search.

To have `outputlookup` create the lookup table file in the system lookups directory, set `createinapp=false` or set `create_context=system`. Alternatively, if you do not have an app context when you run the search, leave both arguments out of the search and rely on the `limits.conf` version of `create_context` to put the lookup table file in the system directory. This last approach only works if the `create_context` setting in `limits.conf` has not been set to `user`.

To have `outputlookup` create the lookup table file in the lookups directory for the current app context, set `createinapp=true` or set `create_context=app`. Alternatively, if you do have an app context when you run the search, leave both arguments out of the search and rely on the `limits.conf` version of `create_context` to put the lookup table file in the app directory. This last approach only works if the `create_context` setting in `limits.conf` has not been set to `user`.

To have `outputlookup` create the lookup table file in the lookups directory for the user running the search, set `create_context=user`. Alternatively, if you want all `outputlookup` searches to create lookup table files in user lookup directories by default, you can set `create_context=user` in `limits.conf`. The `createinapp` and `create_context` arguments can override this setting if they are used in the search.

If the lookup table file already exists in the location to which it is written, the existing version of the file is overwritten with the results of the `outputlookup` search.

Restrict write access to lookup table files with check_permission

For permissions in CSV lookups, use the `check_permission` field in `transforms.conf` and `outputlookup_check_permission` in `limits.conf` to restrict write access to users with the appropriate permissions when using the `outputlookup` command. Both `check_permission` and `outputlookup_check_permission` default to false. Set to true for Splunk software to verify permission settings for lookups for users. You can change lookup table file permissions in the `.meta` file for each lookup file, or **Settings > Lookups > Lookup table files**. By default, only users who have the admin or power role can write to a shared CSV lookup file.

For more information about creating lookups, see About lookups in the *Knowledge Manager Manual*.

For more information about App Key Value Store collections, see About KV store in the *Admin Manual*.

Append results

Suppose you have an existing CSV file that contains fields A, D, and J. The results of your search are fields A, C, and J. If you run a search with `outputlookup append=false`, then fields A, C, and J are written to the CSV file. Field D is not retained.

If you run a search with `outputlookup append=true`, then only the fields that are currently in the file are preserved. In this example, fields A and J are written to the CSV file. Field C is lost because it does not already exist in the CSV file. Field D

is retained.

You can work around this issue by using the `eval` command to add a field to your CSV file **before** you run the search. For example, if your CSV file is named **users**, you would do something like this:

```
| inputlookup users | eval c=null | outputlookup users append=false ....
```

Then run your search and pipe the results to the `fields` command for the fields in the file that you want to preserve.

```
... | fields A C J | outputlookup append=true users
```

Multivalued fields

When you output to a static lookup table, the `outputlookup` command merges values in a multivalued field into single space-delimited value. This does not apply to a KV store collection.

Examples

1. Write to a lookup table using settings in the transforms.conf file

Write to `usertogroup` lookup table as specified in the `transforms.conf` file.

```
| outputlookup usertogroup
```

2. Write to a lookup file in a specific system or app directory

Write to `users.csv` lookup file under `$SPLUNK_HOME/etc/system/lookups` or `$SPLUNK_HOME/etc/apps/*/lookups`.

```
| outputlookup users.csv
```

3. Specify not to override the lookup file if no results are returned

Write to `users.csv` lookup file, if results are returned, under `$SPLUNK_HOME/etc/system/lookups` or `$SPLUNK_HOME/etc/apps/*/lookups`. Do not delete the `users.csv` file if no results are returned.

```
| outputlookup users.csv override_if_empty=false
```

4. Write to a KV store collection

Write food inspection events for Shalimar Restaurant to a KV store collection called `kvstorecoll`. This collection is referenced in a lookup table called `kvstorecoll_lookup`.

```
index=sf_food_health sourcetype=sf_food_inspections name="SHALIMAR RESTAURANT" | outputlookup  
kvstorecoll_lookup
```

5. Overwrite KV store collections

By default, `append` is set to `true` when the `key_field` is used with the `outputlookup` command. If you don't want to append search results to an existing KV store collection, you can override the default behavior by directly setting `key_field` with `append=false`.

For example, in the following `outputlookup` search, the KV store called `accounts` is appended. This is because `key_field` sets `append=true` by default.

```
| makeresults | eval key=1 | outputlookup key_field=key accounts
```

However, in the following `outputlookup` search, the KV store called `accounts` is overwritten because `append=false`. In this case, the `append` subsearch runs before the main search, which empties the entire KV store before the fields are written to `accounts`.

```
| makeresults | eval key=1 | outputlookup append=false key_field=key accounts
```

Alternatively, if you want your entire lookup to reflect your search results and you don't mind using the default system-generated keys, eliminate `key_field=key` from your `outputlookup` search, like this.

```
| makeresults | eval key=1 | outputlookup accounts
```

6. Write from a CSV file to a KV store collection

Write the contents of a CSV file to the KV store collection `kvstorecoll` using the lookup table `kvstorecoll_lookup`. This requires usage of both `inputlookup` and `outputlookup` commands.

```
| inputlookup customers.csv | outputlookup kvstorecoll_lookup
```

7. Update field values for a single KV store collection record

Update field values for a single KV store collection record. This requires you to use the `inputlookup`, `outputlookup`, and `eval` commands. The record is indicated by the value of its internal key ID (the `_key` field) and is updated with a new customer name and customer city. The record belongs to the KV store collection `kvstorecoll`, which is accessed through the lookup table `kvstorecoll_lookup`.

```
| inputlookup kvstorecoll_lookup | search _key=544948df3ec32d7a4c1d9755 | eval CustName="Vanya Patel" | eval CustCity="Springfield" | outputlookup kvstorecoll_lookup append=True key_field=_key
```

To learn how to obtain the internal key ID values of the records in a KV store collection, see Example 5 for the `inputlookup` command.

See also

Commands

[collect](#)
[inputlookup](#)
[lookup](#)
[inputcsv](#)
[mcollect](#)
[meventcollect](#)
[outputcsv](#)
[outputtext](#)

outputtext

Description

Outputs the contents of the `_raw` field to the `_xml` field.

The `outputtext` command was created as an internal mechanism to render event texts for output.

Syntax

```
outputtext [usexml=<bool>]
```

Optional arguments

usexml

Syntax: `usexml=<bool>`

Description: If set to true, the copy of the `_raw` field in the `_xml` is escaped XML. If `usexml` is set to false, the `_xml` field is an exact copy of `_raw`.

Default: true

Usage

The `outputtext` command is a reporting command.

The `outputtext` command writes all search results to the search head. In Splunk Web, the results appear in the Statistics tab.

Examples

1. Output the `_raw` field into escaped XML

Output the `"_raw"` field of your current search into `"_xml"`.

```
... | outputtext
```

See also

[outputcsv](#)

overlap

Note: We do not recommend using the `overlap` command to fill or backfill summary indexes. Splunk Enterprise provides a script called `fill_summary_index.py` that backfills your indexes or fill summary index gaps. If you have Splunk Cloud Platform and need to backfill, open a Support ticket and specify the time range, app, search name, user and any other details required to enable Splunk Support to backfill the required data. For more information, see "Manage summary index gaps" in the *Knowledge Manager Manual*.

Description

Find events in a summary index that overlap in time, or find gaps in time during which a scheduled saved search might have missed events.

- **If you find a gap**, run the search over the period of the gap and summary index the results using "`| collect`".
- **If you find overlapping events**, manually delete the overlaps from the summary index by using the search language.

The `overlap` command invokes an external python script `$SPLUNK_HOME/etc/apps/search/bin/sumindexoverlap.py`. The script expects input events from the summary index and finds any time overlaps and gaps between events with the same 'info_search_name' but different 'info_search_id'.

Important: Input events are expected to have the following fields: 'info_min_time', 'info_max_time' (inclusive and exclusive, respectively) , 'info_search_id' and 'info_search_name' fields. If the index contains raw events (`_raw`), the `overlap` command does not work. Instead, the index should contain events such as `chart`, `stats`, and `timechart` results.

Syntax

`overlap`

Examples

Example 1:

Find overlapping events in the "summary" index.

```
index=summary | overlap
```

See also

[collect](#), [sistats](#), [sitop](#), [sirare](#), [sichart](#), [sitimechart](#)

pivot

Description

The `pivot` command makes simple pivot operations fairly straightforward, but can be pretty complex for more sophisticated pivot operations. Fundamentally this command is a wrapper around the `stats` and `xseries` commands.

The `pivot` command does not add new behavior, but it might be easier to use if you are already familiar with how Pivot works. See the Pivot Manual. Also, read how to open non-transforming searches in Pivot.

Run pivot searches against a particular **data model** object. This requires a large number of inputs: the data model, the data model object, and pivot elements.

Syntax

```
| pivot <datamodel-name> <object-name> <pivot-element>
```

Required arguments

`datamodel-name`

Syntax: `<string>`

Description: The name of the data model to search.

`objectname`

Syntax: `<string>`

Description: The name of a data model object to search.

pivot element

Syntax: (<cellvalue>)* (SPLITROW <rowvalue>)* (SPLITCOL colvalue [options])* (FILTER <filter expression>)* (LIMIT <limit expression>)* (ROWSUMMARY <true | false>)* (COLSUMMARY <true | false>)* (SHOWOTHER <true | false>)* (NUMCOLUMNS <num>)* (rowsort [options])*

Description: Use pivot elements to define your pivot table or chart. Pivot elements include cell values, split rows, split columns, filters, limits, row and column formatting, and row sort options. Cell values always come first. They are followed by split rows and split columns, which can be interleaved, for example: avg(val), SPLITCOL foo, SPLITROW bar, SPLITCOL baz.

Cell value

<cellvalue>

Syntax: <function>(fieldname) [AS <label>]

Description: Define the values of a cell and optionally rename it. Here, `label` is the name of the cell in the report.

The set of allowed functions depend on the data type of the `fieldname`:

- **Strings:** list, values, first, last, count, and distinct_count (dc)
- **Numbers:** sum, count, avg, max, min, stdev, list, and values
- **Timestamps:** duration, earliest, latest, list, and values
- **Object or child counts:** count

Descriptions for row split-by elements

SPLITROW <rowvalue>

Syntax: SPLITROW <field> [AS <label>] [RANGE start=<value> end=<value> max=<value> size=<value>] [PERIOD (auto | year | month | day | hour | minute | second)] [TRUELABEL <label>] [FALSELABEL <label>]

Description: You can specify one or more of these options on each SPLITROW. The options can appear in any order. You can rename the <field> using "AS <label>", where "label" is the name of the row in the report.

Other options depend on the data type of the <field> specified:

- RANGE applies only for numbers. You do not need to specify all of the options (start, end, max, and size).
- PERIOD applies only for timestamps. Use it to specify the period to bucket by.
- TRUELABEL applies only for booleans. Use it to specify the label for true values.
- FALSELABEL applies only for booleans. Use it to specify the label for false values.

Descriptions for column split-by elements

SPLITCOL colvalue <options>

Syntax: fieldname [RANGE start=<value> end=<value> max=<value> size=<value>] [PERIOD (auto | year | month| day | hour | minute | second)] [TRUELABEL <label>] [FALSELABEL <label>]

Description: You can have none, some, or all of these options on each SPLITCOL. They may appear in any order.

Other options depend on the data type of the field specified (fieldname):

- RANGE applies only for numbers. The options (start, end, max, and size) do not all have to be specified.
- PERIOD applies only for timestamps. Use it to specify the period to bucket by.
- TRUELABEL applies only for booleans. Use it to specify the label for true values.

- FALSELABEL applies only for booleans. Use it to specify the label for false values.

Descriptions for filter elements

Filter <filter expression>

Syntax: <fieldname> <comparison-operator> <value>

Description: The expression used to identify values in a field. The comparison operator that you use depends on the type of field value.

- **Strings:** is, contains, in, isNot, doesNotContain, startsWith, endsWith, isNull, isNotNull

For example: ... filter *fieldname* in (*value1*, *value2*, ...)

- **ipv4:** is, contains, isNot, doesNotContain, startsWith, isNull, isNotNull

- **Numbers:** =, !=, <, <=, >, >=, isNull, isNotNull

- **Booleans:** is, isNull, isNotNull

Descriptions for limit elements

Limit <limit expression>

Syntax: LIMIT <fieldname> BY <limittype> <number> <stats-function>(<fieldname>)

Description: Use to limit the number of elements in the pivot. The `limittype` argument specifies where to place the limit. The valid values are `top` or `bottom`. The `number` argument must be a positive integer. You can use any stats function, such as `min`, `max`, `avg`, and `sum`.

Example: LIMIT foo BY TOP 10 avg(bar)

Usage

The `pivot` command is a report-generating command. See [Command types](#).

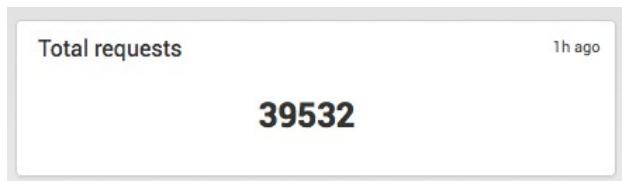
Generating commands use a leading pipe character and should be the first command in a search.

Examples

Example 1: This command counts the number of events in the "HTTP Requests" object in the "Tutorial" data model.

```
| pivot Tutorial HTTP_requests count(HTTP_requests) AS "Count of HTTP requests"
```

This can be formatted as a single value report in the dashboard panel:



Example 2: Using the Tutorial data model, create a pivot table for the count of "HTTP Requests" per host.

```
| pivot Tutorial HTTP_requests count(HTTP_requests) AS "Count" SPLITROW host AS "Server" SORT 100 host
```

Total requests by server		1h ago
Server	Count	
www1	13628	
www2	12912	
www3	12992	

See also

[datamodel](#), [stats](#), [xseries](#)

predict

Description

The `predict` command forecasts values for one or more sets of time-series data. The command can also fill in missing data in a time-series and provide predictions for the next several time steps.

The `predict` command provides confidence intervals for all of its estimates. The command adds a predicted value and an upper and lower 95th percentile range to each event in the time-series. See the **Usage** section in this topic.

Syntax

`predict <field-list> [AS <newfield>] [<predict_options>]`

Required arguments

`<field-list>`

Syntax: `<field>...`

Description: The names of the fields for the variable that you want to predict. You can specify one or more fields.

Optional arguments

`<newfield>`

Syntax: `<string>`

Description: Renames the fields that are specified in the `<field-list>`. You do not need to rename every field that you specify in the `<field-list>`. However, for each field that you want to rename, you must specify a separate `AS <newfield>` clause.

`<predict_options>`

Syntax: `algorithm=<algorithm_name> | correlate_field=<field> | future_timespan=<number> | holdback=<number> | period=<number> | suppress=<bool> | lowerXX=<field> | upperYY=<field>`

Description: Options you can specify to control the predictions. You can specify one or more options, in any order. Each of these options is described in the **Predict options** section.

Predict options

algorithm

Syntax: algorithm= LL | LLT | LLP | LLP5 | LLB | BiLL

Description: Specify the name of the forecasting algorithm to apply. LL, LLT, LLP, and LLP5 are univariate algorithms. LLB and BiLL are bivariate algorithms. All the algorithms are variations based on the Kalman filter. Each algorithm expects a minimum number of data points. If not enough effective data points are supplied, an error message is displayed. For instance, the field itself might have more than enough data points, but the number of effective data points might be small if the `holdback` value that you specify is large.

Default: LLP5

Algorithm option	Algorithm type	Description
LL	Local level	A univariate model with no trends and no seasonality. Requires a minimum of 2 data points. The LL algorithm is the simplest algorithm and computes the levels of the time series. For example, each new state equals the previous state, plus the Gaussian noise.
LLT	Local level trend	A univariate model with trend, but no seasonality. Requires a minimum of 3 data points.
LLP	Seasonal local level	A univariate model with seasonality. The number of data points must be at least twice the number of periods, using the <code>period</code> attribute. The LLP algorithm takes into account the cyclical regularity of the data, if it exists. If you know the number of periods, specify the <code>period</code> argument. If you do not set the <code>period</code> , this algorithm tries to calculate it. LLP returns an error message if the data is not periodic.
LLP5	Combines LLT and LLP models for its prediction.	If the time series is periodic, LLP5 computes two predictions, one using LLT and the other using LLP. The algorithm then takes a weighted average of the two values and outputs that as the prediction. The confidence interval is also based on a weighted average of the variances of LLT and LLP.
LLB	Bivariate local level	A bivariate model with no trends and no seasonality. Requires a minimum of 2 data points. LLB uses one set of data to make predictions for another. For example, assume it uses dataset Y to make predictions for dataset X. If <code>holdback=10</code> , LLB takes the last 10 data points of Y to make predictions for the last 10 data points of X.
BiLL	Bivariate local level	A bivariate model that predicts both time series simultaneously. The covariance of the two series is taken into account.

correlate

Syntax: correlate=<field>

Description: Specifies the time series that the LLB algorithm uses to predict the other time series. Required when you specify the LLB algorithm. Not used for any other algorithm.

Default: None

future_timespan

Syntax: future_timespan=<num>

Description: Specifies how many future predictions the `predict` command will compute. This number must be a non-negative number. You would not use the `future_timespan` option if `algorithm=LLB`.

Default: 5

holdback

Syntax: holdback=<num>

Description: Specifies the number of data points from the end that are not to be used by the `predict` command. Use in conjunction with the `future_timespan` argument. For example, 'holdback=10 future_timespan=10'

computes the predicted values for the last 10 values in the data set. You can then judge how accurate the predictions are by checking whether the actual data point values fall into the predicted confidence intervals.

Default: 0

lowerXX

Syntax: lower<int>=<field>

Description: Specifies a percentage for the confidence interval and a field name to use for the lower confidence interval curve. The <int> value is a percentage that specifies the confidence level. The integer must be a number between 0 and 100. The <field> value is the field name.

Default: The default confidence interval is 95%. The default field name is 'lower95(prediction(X))' where X is the name of the field to be predicted.

period

Syntax: period=<num>

Description: Specifies the length of the time period, or recurring cycle, in the time series data. The number must be at least 2. The LLP and LLP5 algorithms attempt to compute the length of time period if no value is specified. If you specify the `span` argument with the `timechart` command, the unit that you specify for `span` is the unit used for `period`. For example, if your search is `...|timechart span=1d foo2| predict foo2 period=3`. The spans are 1 day and the period for the predict is 3 days. Otherwise, the unit for the time period is a data point. For example, if there are a thousand events, then each event is a unit. If you specify `period=7`, that means the data recycles after every 7 data points, or events.

Default: None

suppress

Syntax: suppress=<field>

Description: Used with the multivariate algorithms. Specifies one of the predicted fields to hide from the output.

Use `suppress` when it is difficult to see all of the predicted visualizations at the same time.

Default: None

upperYY

Syntax: upper<int>=<field>

Description: Specifies a percentage for the confidence interval and a field name to use for the upper confidence interval curve. The <int> value is a percentage that specifies the confidence level. This must be a number between 0 and 100. The <field> value is the field name.

Default: The default confidence interval is 95%. The default field name is 'upper95(prediction(X))' where X is the name of the field to be predicted.

Confidence intervals

The lower and upper confidence interval parameters default to `lower95` and `upper95`. These values specify a confidence interval where 95% of the predictions are expected fall.

It is typical for some of the predictions to fall outside the confidence interval.

- The confidence interval does not cover 100% of the predictions.
- The confidence interval is about a probabilistic expectation and results do not match the expectation exactly.

Usage

Command sequence requirement

The `predict` command must be preceded by the `timechart` command. The `predict` command requires time series data. See the **Examples** section for more details.

How it works

The `predict` command models the data by stipulating that there is an unobserved entity which progresses through time in different states.

To predict a value, the command calculates the best estimate of the state by considering all of the data in the past. To compute estimates of the states, the command hypothesizes that the states follow specific linear equations with Gaussian noise components.

Under this hypothesis, the least-squares estimate of the states are calculated efficiently. This calculation is called the Kalman filter, or Kalman-Bucy filter. For each state estimate, a confidence interval is obtained. The estimate is not a point estimate. The estimate is a range of values that contain the observed, or predicted, values.

The measurements might capture only some aspect of the state, but not necessarily the whole state.

Missing values

The `predict` command can work with data that has missing values. The command calculates the best estimates of the missing values.

Do not remove events with missing values. Removing the events might distort the periodicity of the data. Do not specify `cont=false` with the `timechart` command. Specifying `cont=false` removes events with missing values.

Specifying span

The unit for the `span` specified with the `timechart` command must be seconds or higher. The `predict` command cannot accept subseconds as an input when it calculates the `period`.

Examples

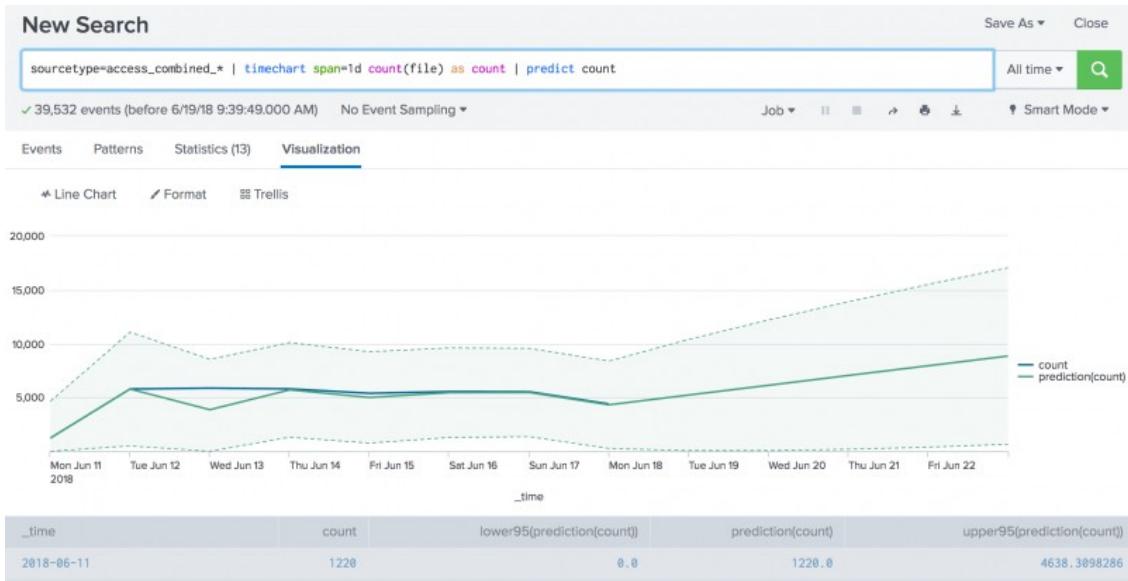
1. Predict future access

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 **All time** when you run the search.

Predict future access based on the previous access numbers that are stored in Apache web access log files. Count the number of access attempts using a span of 1 day.

```
sourcetype=access_combined_* | timechart span=1d count(file) as count | predict count
```

The results appear on the Statistics tab. Click the Visualization tab. If necessary change the chart type to a Line Chart.



2. Predict future purchases for a product

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Chart the number of purchases made daily for a specific product.

```
sourcetype=access_* action=purchase arcade | timechart span=1d count
```

- This example searches for all purchases events, defined by the `action=purchase` for the arc, and pipes those results into the `timechart` command.
- The `span=1day` argument buckets the count of purchases into daily chunks.

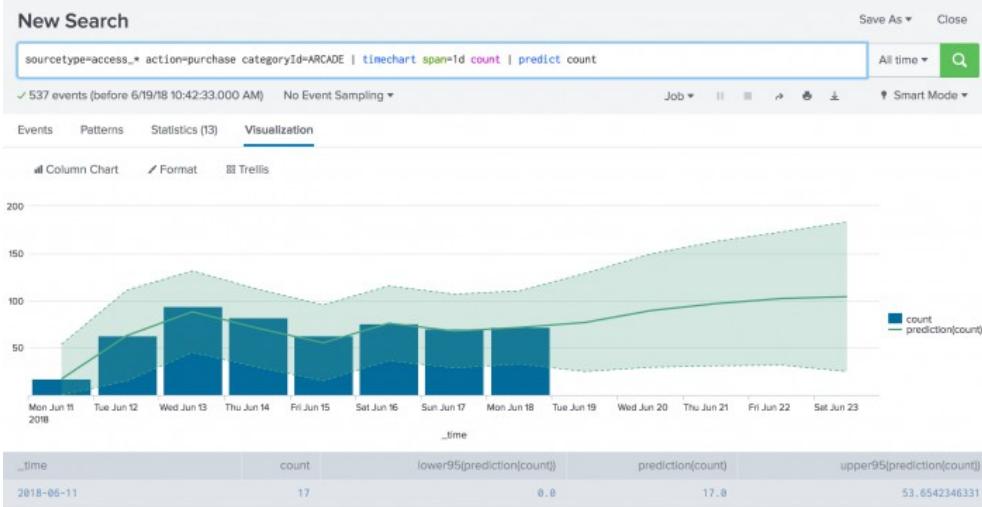
The results appear on the Statistics tab and look something like this:

_time	count
2018-06-11	17
2018-06-12	63
2018-06-13	94
2018-06-14	82
2018-06-15	63
2018-06-16	76
2018-06-17	70
2018-06-18	72

Add the `predict` command to the search to calculate the prediction for the number of purchases of the Arcade games that might be sold in the near future.

```
sourcetype=access_* action=purchase arcade | timechart span=1d count | predict count
```

The results appear on the Statistics tab. Click the Visualization tab. If necessary change the chart type to a Bar Chart.



3. Predict the values using the default algorithm

Predict the values of foo using the default LLP5 algorithm, an algorithm that combines the LLP and LLT algorithms.

```
... | timechart span="1m" count AS foo | predict foo
```

4. Predict multiple fields using the same algorithm

Predict multiple fields using the same algorithm. The default algorithm in this example.

```
... | timechart ... | predict foo1 foo2 foo3
```

5. Specifying different upper and lower confidence intervals

When specifying confidence intervals, the upper and lower confidence interval values do not need to match. This example predicts 10 values for a field using the LL algorithm, holding back the last 20 values in the data set.

```
... | timechart span="1m" count AS foo | predict foo AS foobar algorithm=LL upper90=high lower97=low  
future_timespan=10 holdback=20
```

6. Predict the values using the LLB algorithm

This example illustrates the LLB algorithm. The foo3 field is predicted by correlating it with the foo2 field.

```
... | timechart span="1m" count(x) AS foo2 count(y) AS foo3 | predict foo3 AS foobar algorithm=LLB  
correlate=foo2 holdback=100
```

7. Omit the last 5 data points and predict 5 future values

In this example, the search abstains from using the last 5 data points and makes 5 future predictions. The predictions correspond to the last 5 values in the data. You can judge how accurate the predictions are by checking whether the observed values fall into the predicted confidence intervals.

```
... | timechart ... | predict foo holdback=5 future_timespan=5
```

8. Predict multiple fields using the same algorithm and the same future_timespan and holdback

Predict multiple fields using the same algorithm and same future_timespan and holdback.

```
... | timechart ... | predict foo1 foo2 foo3 algorithm=LLT future_timespan=15 holdback=5
```

9. Specify aliases for fields

Use aliases for the fields by specifying the AS keyword for each field.

```
... | timechart ... | predict foo1 AS foobar1 foo2 AS foobar2 foo3 AS foobar3 algorithm=LLT future_timespan=15 holdback=5
```

10. Predict multiple fields using different algorithms and options

Predict multiple fields using different algorithms and different options for each field.

```
... | timechart ... | predict foo1 algorithm=LL future_timespan=15 foo2 algorithm=LLP period=7 future_timespan=7
```

11. Predict multiple fields using the BiLL algorithm

Predict values for foo1 and foo2 together using the bivariate algorithm BiLL.

```
... | timechart ... | predict foo1 foo2 algorithm=BiLL future_timespan=10
```

See also

[trendline](#), [x11](#)

rangemap

Description

Use the `rangemap` command to categorize the values in a numeric field. The command adds in a new field called `range` to each event and displays the category in the `range` field. The values in the `range` field are based on the numeric ranges that you specify.

Set the `range` field to the names of any `attribute_name` that the value of the input `field` is within. If no range is matched, the `range` value is set to the `default` value.

The ranges that you set can overlap. If you have overlapping values, the `range` field is created as a multivalue field containing all the values that apply. For example, if `low=1-10`, `elevated=5-15`, and the input field value is 10, `range=low` and `code=elevated`.

Syntax

The required syntax is in **bold**.

```
rangemap  
field=<string>  
[<attribute_name>=<numeric_range>]...  
[default=<string>]
```

Required arguments

field

Syntax: field=<string>

Description: The name of the input field. This field must contain numeric values.

Optional arguments

attribute_name=numeric_range

Syntax: <string>=<num>-<num>

Description: The <attribute_name> is a string value that is output when the <numeric_range> matches the value in the <field>. The <attribute_name> is a output to the range field. The <numeric_range> is the starting and ending values for the range. The values can be integers or floating point numbers. The first value must be lower than the second. The <numeric_range> can include negative values.

Example: Dislike=-5--1 DontCare=0-0 Like=1-5

default

Syntax: default=<string>

Description: If the input field does not match a range, use this to define a default value.

Default: "None"

Usage

The rangemap command is a distributable streaming command. See [Command types](#).

Basic examples

Example 1:

Set range to "green" if the date_second is between 1-30; "blue", if between 31-39; "red", if between 40-59; and "gray", if no range matches (for example, if date_second=0).

```
... | rangemap field=date_second green=1-30 blue=31-39 red=40-59 default=gray
```

Example 2:

Sets the value of each event's range field to "low" if its count field is 0 (zero); "elevated", if between 1-100; "severe", otherwise.

```
... | rangemap field=count low=0-0 elevated=1-100 default=severe
```

Extended example

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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and add it as an input. The following examples uses the **All Earthquakes** under the **Past 30 days** list.

This search counts the number and magnitude of each earthquake that occurred in and around Alaska. Then a color is assigned to each magnitude using the `rangemap` command.

```
source=all_month.csv place=*alaska* mag>=3.5 | stats count BY mag | rename mag AS magnitude | rangemap field=magnitude light=3.9-4.3 strong=4.4-4.9 severe=5.0-9.0 default=weak
```

The results look something like this:

magnitude	count	range
3.7	15	weak
3.8	31	weak
3.9	29	light
4	22	light
4.1	30	light
4.2	15	light
4.3	10	light
4.4	22	strong
4.5	3	strong
4.6	8	strong
4.7	9	strong
4.8	6	strong
4.9	6	strong
5	2	severe
5.1	2	severe
5.2	5	severe

Summarize the results by range value

```
source=all_month.csv place=*alaska* mag>=3.5 | stats count BY mag | rename mag AS magnitude | rangemap field=magnitude green=3.9-4.2 yellow=4.3-4.6 red=4.7-5.0 default=gray | stats sum(count) by range
```

The results look something like this:

range	sum(count)
gray	127
green	96

range	sum(count)
red	23
yellow	43

Arrange the results in a custom sort order

By default the values in the search results are in descending order by the `sum(count)` field. You can apply a custom sort order to the results using the `eval` command with the `case` function.

```
source=all_month.csv place=*alaska* mag>=3.5 | stats count BY mag | rename mag AS magnitude | rangemap
field=magnitude green=3.9-4.2 yellow=4.3-4.6 red=4.7-5.0 default=gray | stats sum(count) by range | eval
sort_field=case(range="red",1, range="yellow",2, range="green",3, range="gray",4) | sort sort_field
```

The results look something like this:

range	sum(count)	sort_field
red	23	1
yellow	43	2
green	96	3
gray	127	4

See also

Commands

[eval](#)

Blogs

[Order Up! Custom Sort Orders](#)

rare

Description

Displays the least common values in a field.

Finds the least frequent tuple of values of all fields in the field list. If the `<by-clause>` is specified, this command returns rare tuples of values for each distinct tuple of values of the group-by fields.

This command operates identically to the `top` command, except that the `rare` command finds the least frequent values instead of the most frequent values.

Syntax

`rare [<rare-options>...] <field-list> [<by-clause>]`

Required arguments

<field-list>

Syntax: <string>...

Description: Comma-delimited list of field names.

Optional arguments

<rare-options>

Syntax: countfield=<string> | limit=<int> | percentfield=<string> | showcount=<bool> | showperc=<bool>

Description: Options that specify the type and number of values to display. These are the same as the <top-options> used by the `top` command.

<by-clause>

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

Rare options

countfield

Syntax: countfield=<string>

Description: The name of a new field to write the value of count into.

Default: "count"

limit

Syntax: limit=<int>

Description: Specifies how many tuples to return. If you specify `limit=0`, all values up to the `maxresultrows` are returned. Specifying a value larger than the `maxresultrows` produces an error. See [Usage](#).

Default: 10

percentfield

Syntax: percentfield=<string>

Description: Name of a new field to write the value of percentage.

Default: "percent"

showcount

Syntax: showcount=<bool>

Description: Specifies whether to add a field to your results with the count of the tuple. The name of the field is controlled by the `countfield` argument.

Default: true

showperc

Syntax: showperc=<bool>

Description: Specifies whether to add a field to your results with the relative prevalence of that tuple. The name of the field is controlled by the `percentfield` argument.

Default: true

Usage

The `rare` command is a **transforming command**. See [Command types](#).

Limit maximum

The number of results returned by the `rare` command is controlled by the `limit` argument. The default value for the `limit` argument is 10. The default maximum is 50,000, which effectively keeps a ceiling on the memory that the `rare` command uses.

You can change this limit up to the maximum value specified in the `maxresultrows` setting in the `[rare]` stanza in the `limits.conf` file.

Splunk Cloud Platform

To change the `maxresultrows` setting, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the the `maxresultrows` setting in the `limits.conf` file, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in How to edit a configuration file in the *Splunk Enterprise Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file in the desired path. For example, use the `$SPLUNK_HOME/etc/apps/search/local` path to apply this change only to the Search app.
2. Under the `[rare]` stanza, change the value for the `maxresultrows` setting.

Examples

1. Return the least common values in a field

Return the least common values in the `url` field. Limits the number of values returned to 5.

```
... | rare url limit=5
```

2. Return the least common values organized by host

Find the least common values in the `user` field for each `host` value. By default, a maximum of 10 results are returned.

```
... | rare user by host
```

See also

Related commands

[top](#)
[stats](#)
[sirare](#)

regex

Description

Removes results that match or do not match the specified regular expression.

Syntax

The required syntax is in **bold**.

```
regex  
(<field>=<regex-expression> | <field>!=<regex-expression> | <regex-expression>)
```

Required arguments

<regex-expression>

Syntax: "<string>"

Description: An unanchored regular expression. The regular expression must be a Perl Compatible Regular Expression supported by the PCRE library. Quotation marks are required.

Optional arguments

<field>

Syntax: <field>

Description: Specify the field name from which to match the values against the regular expression.

You can specify that the `regex` command keeps results that match the expression by using

<field>=<regex-expression>. To keep results that do not match, specify <field>!=<regex-expression>.

Default: `_raw`

Usage

The `regex` command is a distributable streaming command. See [Command types](#).

When you use regular expressions in searches, you need to be aware of how characters such as pipe (|) and backslash (\) are handled. See SPL and regular expressions in the *Search Manual*.

Although != is valid within a `regex` command, NOT is not valid.

For general information about regular expressions, see About Splunk regular expressions in the *Knowledge Manager Manual*.

The difference between the `regex` and `rex` commands

Use the `regex` command to remove results that match or do not match the specified regular expression.

Use the `rex` command to either extract fields using regular expression named groups, or replace or substitute characters in a field using sed expressions.

Using the regex command with !=

If you use regular expressions in conjunction with the `regex` command, note that `!=` behaves differently for the `regex` command than for the `search` command.

You can use a `regex` command with `!=` to filter for events that don't have a field value matching the regular expression, or for which the field is null. For example, this search will include events that do not define the field `Location`.

```
... | regex Location!="Calaveras Farms"
```

The `search` command behaves the opposite way. You can use a `search` command with `!=` to filter for events that don't contain a field matching the search string, and for which the field is defined. For example, this search will not include events that do not define the field `Location`.

```
... | search Location!="Calaveras Farms"
```

If you use `!=` in the context of the `regex` command, keep this behavior in mind and make sure you want to include null fields in your results.

Examples

Example 1: Keep only search results whose `_raw` field contains IP addresses in the non-routable class A (10.0.0.0/8). This example uses a negative lookbehind assertion at the beginning of the expression.

```
... | regex _raw="(?<!\d)10\.\d{1,3}\.\d{1,3}\.\d{1,3}(?!d)"
```

Example 2: Keep only the results that match a valid email address. For example, `buttercup@example.com`.

```
... | regex email="^([a-z0-9_\.]+@[\\da-z\.\-]+\.\([a-z\.\.]{2,6}\)$"
```

This regular expression is for example purposes only and isn't a fully RFC-compliant email address validator.

The following table explains each part of the expression.

Part of the expression	Description
<code>^</code>	Specifies the beginning of the string.
<code>([a-z0-9_\.]+)</code>	This is the first group in the expression. Specifies to match one or more lowercase letters, numbers, underscores, dots, or hyphens. The backslash (<code>\</code>) character is used to escape the dot (<code>.</code>) character. The dot character is escaped, because a non-escaped dot matches any character. The plus (+) sign specifies to match from 1 to unlimited characters in this group. In this example this part of the expression matches buttercup in the email address <code>buttercup@example.com</code> .
<code>@</code>	Matches the at symbol.
<code>([\da-z\.\-]+)</code>	This is the second group in the expression. Specifies to match the domain name, which can be one or more lowercase letters, numbers, underscores, dots, or hyphens. This is followed by another escaped dot character. The plus (+) sign specifies to match from 1 to unlimited characters in this group. In this example this part of the expression matches example in the email address <code>buttercup@example.com</code> .
<code>([a-z\.\.]{2,6})</code>	This is the third group. Specifies to match the top-level domain (TLD), which can be 2 to 6 letters or dots. This group matches all types of TLDs, such as <code>.co.uk</code> , <code>.edu</code> , or <code>.asia</code> . In this example it matches .com in the email address <code>buttercup@example.com</code> .

Part of the expression	Description
	buttercup@example.com.
\$	Specifies the end of the string.

Example 3: Filter out zip codes that are formatted like a United States zip code or zip+4 code. For example, this search would return a Canadian zip code.

```
... | regex not_usa_zip!="[0-9]{5}(-[0-9]{4})?"
```

Example 4: The search with `regex` and `!=` in the following example creates 5 events with `Country="Canada"` and 5 events with `City="Ontario"`, and filters on events where `Country` does not equal "Canada".

```
| makeresults count=5 | eval Country="Canada" | append [ | makeresults count=5 | eval city="Ontario" ] | regex country!="Canada"
```

This search returns the union of two groups of events: events where the field `Country` is defined and has a value not equal to "Canada"; and events where the field `Country` is not defined. As a result, 5 events are displayed for the `City` field, even though a `Country` field was not defined for those events. Also, the `Country` field is displayed, but the values are null. The results look something like this.

_time	city	country
2020-11-02 15:48:47	Ontario	

In contrast, the search with `search` and `!=` in the following example doesn't return any events because all of the events with field `City` where the field `Country` is null are excluded.

```
| makeresults count=5 | eval country="Canada" | append [ | makeresults count=5 | eval city="Ontario" ] | search country!="Canada"
```

See also

Commands

[rex](#)
[search](#)

reltime

Description

Creates one or more relative time fields and adds the field or fields to returned events. Each added relative time field provides a human-readable value of the difference between "now" (the start time of the search) and the `timestamp` value

of a corresponding field in the returned event. Human-readable values look like 5 days ago, 1 minute ago, 2 years ago, and so on.

Syntax

The required syntax is in **bold**.

```
| reltime
[timefield=<field-list>]
[prefix=<string>]
```

Optional arguments

timefield

Syntax: timefield=<field-list>

Description: Specifies one or more time fields in the events returned by the search. The `reltime` command uses these fields as the basis for the relative time field that it adds to the events. `timefield` can specify only fields with values that are valid timestamps. `timefield` can specify multiple time fields as a comma-separated list bounded by double quotation marks.

Default: `_time`

prefix

Syntax: prefix=<string>

Description: Sets a prefix string for relative time field names. Use it to help others identify fields added by `reltime` or to provide unique field names when you identify multiple `timefield` values. If you specify multiple values for `timefield` but do not specify a `prefix`, the `reltime` command prefixes the relative time fields that it adds with `reltime_`.

Usage

The `reltime` command adds one or more relative time fields to your events. Each field added provides a human-readable value that represents the difference between `now` (the start time of the search) and the timestamp value of a field in the event.

For example, say you tie `reltime` to the `_time` fields in your events. If you run a search at 6 a.m., and the search returns an event with a `_time` value that translates to 5 a.m., `reltime` adds a field to that event named `reltime` with the value 1 hour ago.

If you use `reltime` without arguments, the command adds a relative time field to your events named `reltime`. This new field will be based on the `_time` field in each of your events.

The following table explains how `reltime` defines and names the fields that it adds.

Custom <code>timefield</code> specified?	Custom <code>prefix</code> specified?	Basis for field(s) added by <code>reltime</code>	Name(s) of field(s) added by <code>reltime</code>
None	No	<code>_time</code>	<code>reltime</code>
One <code>timefield</code> specified	No	The time field you specified for <code>timefield</code>	<code>reltime</code>
One <code>timefield</code> specified	Yes	The time field you specified for <code>timefield</code>	<code>reltime</code> , prefixed by your custom <code>prefix</code> string

Custom timefield specified?	Custom prefix specified?	Basis for field(s) added by reltime	Name(s) of field(s) added by reltime
Multiple time fields specified	No	The list of time fields you specified for timefield	The names of the fields you specified for timefield, prefixed by reltime_
Multiple time fields specified	Yes	The list of time fields you specified for timefield	The names of the fields you specified for timefield, prefixed by your custom prefix string

The `reltime` command is a distributable streaming command. See [Command types](#).

Examples

Example 1:

Adds a field called `reltime` to the events returned by the search, based on the `_time` field in those events.

```
... | reltime
```

Example 2:

Adds a field called `reltime` to events returned by the search, based on the `earliest_time` field in those events.

```
... | reltime timefield=earliest_time
```

Example 3:

Adds a field called `reltime_now_current_time` to events, based on the `current_time` field in those events.

```
... | reltime timefield=current_time prefix=reltime_now_
```

Example 4:

Adds three new relative time fields called `reltime_max_time`, `reltime_min_time`, and `reltime_current_time` to returned events with `max_time`, `min_time`, and `current_time` fields.

```
... | reltime timefield="max_time,min_time,current_time"
```

Example 5:

Adds two new relative time fields called `usr_prefix_max_time` and `usr_prefix_min_time` to returned events with `max_time` and `min_time` fields.

```
... | reltime timefield="max_time,min_time" prefix=usr_prefix_
```

See also

[convert](#)

rename

Description

Use the `rename` command to rename one or more fields. This command is useful for giving fields more meaningful names, such as "Product ID" instead of "pid". If you want to rename fields with similar names, you can use a wildcard character. See the [Usage](#) section.

Syntax

```
rename <wc-field> AS <wc-field>...
```

Required arguments

wc-field

Syntax: <string>

Description: The name of a field and the name to replace it. Field names with spaces must be enclosed in quotation marks. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Usage

The `rename` command is a distributable streaming command. See [Command types](#).

Rename with a phrase

Use quotation marks when you rename a field with a phrase.

```
... | rename SESSIONID AS "The session ID"
```

Rename multiple, similarly named fields

Use wildcards to rename multiple fields with similar names. For example, suppose you have the following field names:

- EU_UK
- EU_DE
- EU_PL

You can rename the fields to replace EU with EMEA:

```
... | rename EU* AS EMEA*
```

The results show these field names:

- EMEA_UK
- EMEA_DE
- EMEA_PL

Both the original and renamed fields must include the same number of wildcards, otherwise a wildcard mismatch error is returned. See [Examples](#).

You can't rename one field with multiple names

You can't rename one field with multiple names. For example if you have field A, you can't specify `| rename A as B, A as C`. This rule also applies to other commands where you can rename fields, such as the `stats` command.

The following example is not valid:

```
... | stats first(host) AS site, first(host) AS report
```

You can't merge multiple fields into one field

You can't use the `rename` command to merge multiple fields into one field because null, or non-present, fields are brought along with the values.

For example, if you have events with either `product_id` or `pid` fields, `... | rename pid AS product_id` would not merge the `pid` values into the `product_id` field. It overwrites `product_id` with Null values where `pid` does not exist for the event. See [the eval command](#) and [coalesce\(\) function](#).

You can't match wildcard characters while renaming fields

You can use the asterisk (`*`) in your searches as a wildcard character, but you can't use a backslash (`\`) to escape an asterisk in search strings. A backslash `\` and an asterisk `*` match the characters `*` in searches, not an escaped wildcard character. Because the Splunk platform doesn't support escaping wildcards, asterisk (`*`) characters in field names in `rename` searches can't be matched and replaced.

Renaming a field that does not exist

Renaming a field can cause loss of data.

Suppose you rename `fieldA` to `fieldB`, but `fieldA` does not exist.

- If `fieldB` does not exist, nothing happens.
- If `fieldB` does exist, the result of the `rename` is that the data in `fieldB` is removed. The data in `fieldB` will contain null values.

The original and new field names must have the same number of wildcards

The number of asterisks (`*`) in the original name must match the number of asterisks in the new name. For example, the following search fails because there is one wildcard character in the original name, but none in the name that replaces it:

```
... | rename price-a*price-b AS price-a\price-b
```

The following search completes successfully because the number of wildcard characters in both names is the same.

```
... | rename price-a*price-b AS price-a*Newprice-b
```

Support for backslash characters (`\`) in the `rename` command

To match a backslash character (`\`) in a field name when using the `rename` command, use 2 backslashes for each backslash in the original field name. For example, to rename the field name `http\\:8000` to `localhost:8000`, use the following command in your search:

```
... | rename http\\\\\\:* AS localhost:*
```

See Backslashes in the *Search Manual*.

Examples

1. Rename a single field

Rename the "_ip" field to "IPAddress".

```
... | rename _ip AS IPAddress
```

2. Rename fields with similar names using a wildcard

Rename fields that begin with "usr" to begin with "user".

```
... | rename usr* AS user*
```

3. Specifying a field name that contains spaces

Rename the "count" field. Names with spaces must be enclosed in quotation marks.

```
... | rename count AS "Count of Events"
```

See also

Commands

[fields](#)
[replace](#)
[table](#)

replace

Description

Replaces field values in your search results with the values that you specify. Does not replace values in fields generated by `stats` or `eval` functions. If you do not specify a field, the value is replaced in all non-generated fields.

Syntax

```
replace (<wc-string> WITH <wc-string>)... [IN <field-list>]
```

Required arguments

wc-string

Syntax: <string>

Description: Specify one or more field values and their replacements. You can use wildcard characters to match one or multiple terms.

Optional arguments

field-list

Syntax: <string> ...

Description: Specify a comma or space delimited list of one or more field names for the field value replacements. To replace values on _internal fields, you must specify the field name with the IN <fieldname> clause.

Usage

The `replace` command is a distributable streaming command. See [Command types](#).

Non-wildcard replacement values specified later take precedence over those replacements specified earlier. For a wildcard replacement, fuller matches take precedence over lesser matches. To assure precedence relationships, you are advised to split the replace into two separate invocations. When using wildcard replacements, the result must have the same number of wildcards, or none at all. Wildcards (*) can be used to specify many values to replace, or replace values with.

Examples

1. Replace a value in all fields

Change any host value that ends with "localhost" to simply "localhost" in all fields.

```
... | replace *localhost WITH localhost
```

2. Replace a value in a specific field

Replace an IP address with a more descriptive name in the `host` field.

```
... | replace 127.0.0.1 WITH localhost IN host
```

3. Change the value of two fields

Replaces the values in the `start_month` and `end_month` fields. You can separate the names in the field list with spaces or commas.

```
... | replace aug WITH August IN start_month end_month
```

4. Change the order of values in a field

In the host field, change the order of string values that contain the word `localhost` so that the string "localhost" precedes the other strings.

```
... | replace "* localhost" WITH "localhost *" IN host
```

5. Replace multiple values in a field

Replace the values in a field with more descriptive names. Separate the value replacements with comma.

```
... | replace 0 WITH Critical, 1 WITH Error IN msg_level
```

6. Replace empty strings

Search for an error message and replace empty strings with a whitespace.

This example will not work unless you have values that are actually the empty string, which is not the same as not having a value.

```
"Error exporting to XYZ :" | rex "Error exporting to XYZ:(?.*)" | replace "" WITH " " IN errmsg
```

7: Replace values in an internal field

Replace values of the internal field `_time`.

```
sourcetype=* | head 5 | eval _time="XYZ" | stats count BY _time | replace *XYZ* WITH *ALL* IN _time
```

See also

Commands

[rename](#)

require

Description

Causes a search to fail if the queries and commands that precede it in the search string return zero events or results.

Syntax

The required syntax is in **bold**.

| **require**

Usage

When `require` is used in a search string, it causes the search to fail if the queries and commands that precede it in the search string return zero events or results. When you use it in a subsearch, it causes the parent search to fail when the subsearch fails to return results.

Use this command to prevent the Splunk platform from running zero-result searches when this might have certain negative side effects, such as generating false positives, running custom search commands that make costly API calls, or creating empty search filters via a `subsearch`.

The `require` command cannot be used in **real-time searches**.

Require and subsequent commands

Do not expect the `require` command to mitigate all possible negative consequences of a search. When the `require` command causes a search to fail, it prevents subsequent commands in the search from receiving the results, but it does not prevent the Splunk software from invoking those commands before the search is finalized. This means that those

subsequent search command processors may receive empty "chunks" before the search is finalized.

If you are implementing a custom search command, make sure it interoperates well with the `require` command. Ensure that it avoids exhibiting side effects in response to partial input.

See Create custom search commands for apps in Splunk Cloud Platform or Splunk Enterprise in the *Developer Guide* on the Developer Portal.

Examples

1. Stop running a search if it returns zero results or events

```
... | require
```

2. Raise an exception if the subsearch returns zero events or results, and stop the parent search.

```
... [ search index=other_index NOSUCHVALUE | require ]
```

rest

Description

The `rest` command reads a Splunk REST API endpoint and returns the resource data as a search result.

Splunk Cloud Platform

For information about Splunk REST API endpoints, see the *REST API Reference Manual*.

Splunk Enterprise

For information about the REST API, see the *REST API User Manual*. For information about Splunk REST API endpoints, see the *REST API Reference Manual*.

Syntax

The required syntax is in **bold**.

```
| rest <rest-uri>
[count=<int>]
[strict=<bool>]
[splunk_server=<wc-string>]
[splunk_server_group=<wc-string>]...
[timeout=<int>]
[<get-arg-name>=<get-arg-value>]...
```

Required arguments

rest-uri

Syntax: <uri>

Description: URI path to the Splunk REST API endpoint.

Optional arguments

count

Syntax: count=<int>

Description: Limits the number of results returned from each REST call. For example, you have four indexers and one search head. You set the limit to `count=25000`. This results in a total limit of 125000, which is 25000 x 5. When `count=0`, there is no limit.

Default: 0

get-arg-name

Syntax: <string>

Description: REST argument name for the REST endpoint. See the Splunk Cloud Platform *REST API Reference Manual*.

get-arg-value

Syntax: <string>

Description: REST argument value for the REST endpoint. See the Splunk Cloud Platform *REST API Reference Manual*.

splunk_server

Syntax: splunk_server=<wc-string>

Description: Specifies the distributed search peer from which to return results. You can specify only one `splunk_server` argument. However, you can use a wildcard character when you specify the server name to indicate multiple servers. For example, you can specify `splunk_server=peer01` or `splunk_server=peer*`. Use `local` to refer to the search head.

Default: All configured search peers return information

splunk_server_group

Syntax: splunk_server_group=<wc-string>...

Description: Limits the results to one or more server groups. You can specify a wildcard character in the string to indicate multiple server groups.

strict

Syntax: strict=<bool>

Description: When set to `true` this argument forces the search to fail completely if `rest` raises an error. This happens even when the errors apply to a subsearch. When set to `false`, many `rest` error conditions return warning messages but do not otherwise cause the search to fail. Certain error conditions cause the search to fail even when `strict=false`.

Default: false

timeout

Syntax: timeout=<int>

Description: Specify the timeout, in seconds, to wait for the REST endpoint to respond. Specify `timeout=0` to indicate no limit on the time to wait for the REST endpoint to respond.

Default: 60

Usage

The `rest` command authenticates using the ID of the person that runs the command.

Strict error handling

Use the `strict` argument to make `rest` searches fail whenever they encounter an error condition. You can set this at the system level for all `rest` searches by changing `restprocessor_errors_fatal` in `limits.conf`.

If you use Splunk Cloud Platform, file a Support ticket to change the `restprocessor_errors_fatal` setting.

Use the `strict` argument to override the `restprocessor_errors_fatal` setting for a `rest` search.

Examples

1. Access saved search jobs

```
| rest /services/search/jobs count=0 splunk_server=local | search isSaved=1
```

2. Find all saved searches with searches that include a specific sourcetype

Find all saved searches with search strings that include the `speccsv` sourcetype.

```
| rest /services/saved/searches splunk_server=local | rename search AS saved_search | fields author, title, saved_search | search saved_search=*speccsv*
```

3. Showing events only associated with the current user

To create reports that only show events associated with the logged in user, you can add the current search user to all events.

```
* | head 10 | join [ | rest splunk_server=local /services/authentication/current-context | rename username as auth_user_id | fields auth_user_id ]
```

4. Use the GET method pagination and filtering parameters

Most GET methods support a set of pagination and filtering parameters.

To determine if an endpoint supports these parameters, find the endpoint in the Splunk Cloud Platform *REST API Reference Manual*. Click **Expand** on the GET method and look for a link to the Pagination and filtering parameters topic. For more information about the Pagination and filtering parameters, see the Request and response details in the Splunk Cloud Platform *REST API Reference manual*.

The following example uses the `search` parameter for the `saved/searches` endpoint to identify if a search is scheduled and disabled. The search looks for scheduled searches on Splunk servers that match the Monitoring Console role of "search heads".

```
| rest /servicesNS/-/-/saved/searches splunk_server_group=dmc_group_search_head timeout=0  
search="is_scheduled=1" search="disabled=0"
```

Here is an explanation for each part of this search:

Description	Part of the search
The name of the REST call.	<code> rest /servicesNS/-/-/saved/searches</code>

Description	Part of the search
Look only at Splunk servers that match the Monitoring Console role of "search heads".	splunk_server_group=dmc_group_search_head
Don't time out waiting for the REST call to finish.	timeout=0
Look only for scheduled searches.	search="is_scheduled=1"
Look only for active searches (not disabled).	search="disabled=0"

5. Return a table of results with custom endpoints

When you create a custom endpoint, you can format the response to return a table of results. The following example shows a custom endpoint:

```
| rest /servicesNS/-/myapp/myapp/endpoint
```

Here's an example of the response you can use to return a table of results:

```
{
    "links": {},
    "entry": [
        {"content": {"name": "world", "fish": "salmon"}},
        {"content": {"name": "muu", "fish": "whale"}}
    ]
}
return
```

Description

Returns values from a subsearch.

The `return` command is used to pass values up from a subsearch. The command replaces the incoming events with one event, with one attribute: "search". To improve performance, the `return` command automatically limits the number of incoming results with the `head` command and the resulting fields with the `fields` command.

By default, the `return` command uses only the first row of results. Use the `count` argument to specify the number of results to use.

Syntax

```
return [<count>] [<alias>=<field>...] [<field>...] [$<field>...]
```

Required arguments

None.

Optional arguments

<count>

Syntax: <int>

Description: Specify the number of rows.

Default: 1, which is the first row of results passed into the command.

<alias>

Syntax: <alias>=<field>...

Description: Specify the field alias and value to return. You can specify multiple pairs of aliases and values, separated by spaces.

<field>

Syntax: <field>...

Description: Specify one or more fields to return, separated by spaces.

<\$field>

Syntax: <\$field>

Description: Specify one or more field values to return, separated by spaces.

Usage

The command is convenient for outputting a field name, a alias-value pair, or just a field value.

Output	Example
Field name	return source
Alias=value	return ip=srcip
Value	return \$srcip

By default, the `return` command uses only the first row of results. You can specify multiple rows, for example '`return 2 ip`'. Each row is viewed as an OR clause, that is, output might be '(`ip=10.1.11.2`) OR (`ip=10.2.12.3`)'. Multiple values can be specified and are placed within OR clauses. So, '`return 2 user ip`' might output '(`user=bob ip=10.1.11.2`) OR (`user=fred ip=10.2.12.3`)'.

In most cases, using the `return` command at the end of a subsearch removes the need for `head`, `fields`, `rename`, `format`, and `dedup`.

Duplicate values

Suppose you have the following search:

```
sourcetype=WinEventLog:Security | return 2 user
```

You might logically expect the command to return the first two distinct users. Instead the command looks at the first two events, based on the ordering from the implied `head` command. The `return` command returns the users within those two events. The command does not determine if the user value is unique. If the same user is listed in these events, the command returns only the one user.

To return unique values, you need to include the `dedup` command in your search. For example:

```
sourcetype=WinEventLog:Security | dedup user | return 2 user
```

Quotations in returned fields

The `return` command does not escape quotation marks that are in the fields that are returned. You must use an `eval` command to escape the quotation marks before you use the `return` command. For example:

```
...[search eval field2=replace(field1,"\"", "\\\"") | return field2]
```

Examples

Example 1:

Search for 'error ip=<someip>', where <someip> is the most recent ip used by user 'boss'.

```
error [ search user=boss | return ip ]
```

Example 2:

Search for 'error (user=user1 ip=ip1) OR (user=user2 ip=ip2)', where the users and IPs come from the two most-recent logins.

```
error [ search login | return 2 user ip ]
```

Example 3:

Return to eval the userid of the last user, and increment it by 1.

```
... | eval nextid = 1 + [ search user=* | return $id ] | ...
```

See also

[format](#), [search](#)

reverse

Description

Reverses the order of the results.

The `reverse` command does not affect which results are returned by the search, only the order in which the results are displayed. For the CLI, this includes any default or explicit `maxout` setting.

On very large result sets, which means sets with millions of results or more, `reverse` command requires large amounts of temporary storage, I/O, and time.

Syntax

```
reverse
```

Usage

The `reverse` command is a dataset processing command. See [Command types](#).

Examples

Example 1:

Reverse the order of a result set.

```
... | reverse
```

See also

Commands

```
head  
sort  
tail
```

rex

Description

Use this command to either extract fields using regular expression named groups, or replace or substitute characters in a field using sed expressions.

The `rex` command matches the value of the specified field against the unanchored regular expression and extracts the named groups into fields of the corresponding names.

When `mode=sed`, the given sed expression used to replace or substitute characters is applied to the value of the chosen field. This sed-syntax is also used to mask, or anonymize, sensitive data at **index-time**. Read about using sed to anonymize data in the *Getting Data In Manual*.

If a field is not specified, the regular expression or sed expression is applied to the `_raw` field. Running the `rex` command against the `_raw` field might have a performance impact.

Use the `rex` command for **search-time field extraction** or string replacement and character substitution.

Syntax

The required syntax is in **bold**.

```
rex [field=<field>]  
( <regex-expression> [max_match=<int>] [offset_field=<string>] ) | (mode=sed <sed-expression>)
```

Required arguments

You must specify either <regex-expression> or mode=sed <sed-expression>.

regex-expression

Syntax: "<string>"

Description: The PCRE regular expression that defines the information to match and extract from the specified field.

mode

Syntax: mode=sed

Description: Specify to indicate that you are using a sed (UNIX stream editor) expression.

sed-expression

Syntax: "<string>"

Description: When mode=sed, specify whether to replace strings (s) or substitute characters (y) in the matching regular expression. No other sed commands are implemented. Sed mode supports the following flags: global (g) and Nth occurrence (N), where N is a number that is the character location in the string.

Optional arguments

field

Syntax: field=<field>

Description: The field that you want to extract information from.

Default: _raw

max_match

Syntax: max_match=<int>

Description: Controls the number of times the regex is matched. If greater than 1, the resulting fields are multivalued fields. Use 0 to specify unlimited matches. Multiple matches apply to the repeated application of the whole pattern. If your regex contains a capture group that can match multiple times **within** your pattern, only the last capture group is used for multiple matches.

Default: 1

offset_field

Syntax: offset_field=<string>

Description: Creates a field that lists the position of certain values in the `field` argument, based on the regular expression specified in `regex-expression`. For example, if the `rex` expression is "`(?<tenchars>.){10}`" the first ten characters of the `field` argument are matched. The `offset_field` shows `tenchars=0-9`. The offset calculation always uses zero (0) for the first position. For another example, see [Examples](#).

Default: No default

Usage

The `rex` command is a distributable streaming command. See [Command types](#).

`rex command` or `regex command`?

Use the `rex` command to either extract fields using regular expression named groups, or replace or substitute characters in a field using sed expressions.

Use the `regex` command to remove results that do not match the specified regular expression.

Regular expressions

Splunk SPL supports perl-compatible regular expressions (PCRE).

When you use regular expressions in searches, you need to be aware of how characters such as pipe (|) and backslash (\) are handled. See SPL and regular expressions in the *Search Manual*.

For general information about regular expressions, see About Splunk regular expressions in the *Knowledge Manager Manual*.

Sed expressions

When using the `rex` command in sed mode, you have two options: replace (s) or character substitution (y).

The syntax for using sed to replace (s) text in your data is: "s/<regex>/<replacement>/<flags>"

- <regex> is a PCRE regular expression, which can include capturing groups.
- <replacement> is a string to replace the regex match. Use \n for back references, where "n" is a single digit.
- <flags> can be either g to replace all matches, or a number to replace a specified match.

The syntax for using sed to substitute characters is: "y/<string1>/<string2>/"

- This substitutes the characters that match <string1> with the characters in <string2>.

Examples

1. Extract email values using regular expressions

Extract email values from events to create `from` and `to` fields in your events. For example, you have events such as:

```
Mon Mar 19 20:16:27 2018 Info: Bounced: DCID 8413617 MID 19338947 From: <MariaDubois@example.com> To: <zecora@buttercupgames.com> RID 0 - 5.4.7 - Delivery expired (message too old) ('000', ['timeout'])
```

```
Mon Mar 19 20:16:03 2018 Info: Delayed: DCID 8414309 MID 19410908 From: <WeiZhang@example.com> To: <mcintosh@buttercupgames.com> RID 0 - 4.3.2 - Not accepting messages at this time ('421', ['4.3.2 try again later'])
```

```
Mon Mar 19 20:16:02 2018 Info: Bounced: DCID 0 MID 19408690 From: <Exit_Desk@sample.net> To: <lyra@buttercupgames.com> RID 0 - 5.1.2 - Bad destination host ('000', ['DNS Hard Error looking up mahidnrasatyambsg.com (MX): NXDomain'])
```

```
Mon Mar 19 20:15:53 2018 Info: Delayed: DCID 8414166 MID 19410657 From: <Manish_Das@example.com> To: <dash@buttercupgames.com> RID 0 - 4.3.2 - Not accepting messages at this time ('421', ['4.3.2 try again later'])
```

When the events were indexed, the From and To values were not identified as fields. You can use the `rex` command to extract the field values and create `from` and `to` fields in your search results.

The from and to lines in the _raw events follow an identical pattern. Each from line is **From:** and each to line is **To:**. The email addresses are enclosed in angle brackets. You can use this pattern to create a regular expression to extract the values and create the fields.

```
source="cisco_esa.txt" | rex field=_raw "From: <(?<from>.*)> To: <(?<to>.*)>"
```

You can remove duplicate values and return only the list of address by adding the `dedup` and `table` commands to the search.

```
source="cisco_esa.txt" | rex field=_raw "From: <(?<from>.*)> To: <(?<to>.*)>" | dedup from to | table from to
```

from	to
eduardo.rodriguez@example.net	pinkie@buttercupgames.com
na.lui@example.net	dash@buttercupgames.com
Vanya_Patel@example.com	rutherford@buttercupgames.com
MariaDubois@example.com	zecora@buttercupgames.com
na.lui@example.net	rarity@buttercupgames.com
WeiZhang@example.com	mcmintosh@buttercupgames.com
Exit_Desk@example.net	lyra@buttercupgames.com

The results look something like this:

2. Extract from multi-valued fields using max_match

You can use the `max_match` argument to specify that the regular expression runs multiple times to extract multiple values from a field.

For example, use the `makeresults` command to create a field with multiple values:

```
| makeresults | eval test="a$1,b$2"
```

The results look something like this:

_time	test
2019-12-05 11:15:28	a\$1,b\$2

To extract each of the values in the `test` field separately, you use the `max_match` argument with the `rex` command. For example:

```
... | rex field=test max_match=0 "((?<field>[^$]*)\$(?<value>[^,]*),?)"
```

The results look something like this:

_time	field	test	value
2019-12-05 11:36:57	a	a\$1,b\$2	1
	b		2

3. Extract values from a field in scheduler.log events

Extract "user", "app" and "SavedSearchName" from a field called "savedsearch_id" in scheduler.log events. If savedsearch_id=bob;search;my_saved_search then user=bob , app=search and SavedSearchName=my_saved_search

```
... | rex field=savedsearch_id "(?<user>\w+);(?<app>\w+);(?<SavedSearchName>\w+)"
```

4. Use a sed expression

Use `sed` syntax to match the regex to a series of numbers and replace them with an anonymized string.

```
... | rex field=ccnumber mode=sed "s/(\d{4}-){3}/XXXX-XXXX-XXXX-/g"
```

5. Use a sed expression with capture replace for strings

This example shows how to use the `rex` command `sed` expression with capture replace using `\1`, `\2` to reuse captured pieces of a string.

This search creates an event with three fields, `_time`, `search`, and `orig_search`. The regular expression removes the quotation marks and any leading or trailing spaces around the quotation marks.

```
|makeresults |eval orig_search="src_ip=TERM( \"10.8.2.33\" ) OR src_ip=TERM( \"172.17.154.197\" )", search=orig_search |rex mode=sed field=search "s/\s\"(\d+\.\d+\.\d+\.\d+)\\"/\s/\1/g"
```

The results look like this:

<code>_time</code>	<code>orig_search</code>	<code>search</code>
2021-05-31 23:36:29	src_ip=TERM("10.8.2.33") OR src_ip=TERM("172.17.154.197")	src_ip=TERM(10.8.2.33) OR src_ip=TERM(172.17.154.197)

6. Use an offset_field

To identify the position of certain values in a field, use the `rex` command with the `offset_field` argument and a regular expression.

The following example starts with the `makeresults` command to create a field with a value:

```
| makeresults | eval list="abcdefghijklmnopqrstuvwxyz"
```

The results look something like this:

<code>_time</code>	<code>list</code>
2022-05-21 11:36:57	abcdefghijklmnopqrstuvwxyz

Add the `rex` command with the `offset_field` argument to the search to create a field called `off`. You can identify the position of the first five values in the field `list` using the regular expression `"(?<firstfive>abcde)"`. For example:

```
| makeresults | eval list="abcdefghijklmnopqrstuvwxyz" | rex offset_field=off field=list "(?<firstfive>abcde)"
```

The results look something like this:

_time	firstfive	list	off
2022-05-21 11:36:57	abcde	abcdefghijklmnopqrstuvwxyz	firstfive=0-4

You can identify the position of several of the middle values in the field `list` using the regular expression `"(?<middle>fgh)".` For example:

```
| makeresults | eval list="abcdefghijklmnopqrstuvwxyz" | rex offset_field=off field=list "(?<middle>fgh)"
```

The results look something like this:

_time	list	middle	off
2022-05-21 11:36:57	abcdefghijklmnopqrstuvwxyz	fgh	middle=5-7

7. Display IP address and ports of potential attackers

Display IP address and ports of potential attackers.

```
sourcetype=linux_secure port "failed password" | rex "\s+(?<ports>port \d+)" | top src_ip ports showperc=0
```

This search uses the `rex` command to extract the port field and values. The search returns a table that lists the top source IP addresses (`src_ip`) and ports of the potential attackers.

See also

Commands

- [extract](#)
- [kvform](#)
- [multikv](#)
- [regex](#)
- [spath](#)
- [xmlkv](#)

rtorder

Description

Buffers events from real-time search to emit them in ascending time order when possible.

The `rtorder` command creates a streaming event buffer that takes input events, stores them in the buffer in ascending time order, and emits them in that order from the buffer. This is only done after the current time reaches at least the span of time given by `buffer_span`, after the timestamp of the event.

Events are also emitted from the buffer if the maximum size of the buffer is exceeded.

If an event is received as input that is earlier than an event that has already been emitted previously, the out of order event is emitted immediately unless the `discard` option is set to true. When `discard` is set to true, out of order events are always discarded to assure that the output is strictly in time ascending order.

Syntax

`rtorder [discard=<bool>] [buffer_span=<span-length>] [max_buffer_size=<int>]`

Optional arguments

`buffer_span`

Syntax: `buffer_span=<span-length>`

Description: Specify the length of the buffer.

Default: 10 seconds

`discard`

Syntax: `discard=<bool>`

Description: Specifies whether or not to always discard out-of-order events.

Default: false

`max_buffer_size`

Syntax: `max_buffer_size=<int>`

Description: Specifies the maximum size of the buffer.

Default: 50000, or the `max_result_rows` setting of the [search] stanza in limits.conf.

Examples

Example 1:

Keep a buffer of the last 5 minutes of events, emitting events in ascending time order once they are more than 5 minutes old. Newly received events that are older than 5 minutes are discarded if an event after that time has already been emitted.

```
... | rtorder discard=t buffer_span=5m
```

See also

[sort](#)

run

The `run` command is an alias for the `script` command. See the [script](#) command for the syntax and examples.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

savedsearch

Description

Runs a saved search, or report, and returns the search results of a saved search. If the search contains replacement placeholder terms, such as \$replace_me\$, the search processor replaces the placeholders with the strings you specify. For example:

```
| savedsearch mysearch replace_me="value"
```

Syntax

```
| savedsearch <savedsearch_name> [<savedsearch-options>...]
```

Required arguments

savedsearch_name

Syntax: <string>

Description: Name of the saved search to run.

Optional arguments

savedsearch-options

Syntax: <substitution-control> | <replacement>

Description: Specify whether substitutions are allowed. If allowed, specify the key-value pair to use in the string substitution replacement.

substitution-control

Syntax: nosubstitution=<bool>

Description: If true, no string substitution replacements are made.

Default: false

replacement

Syntax: <field>=<string>

Description: A key-value pair to use in string substitution replacement.

Usage

The `savedsearch` command is a generating command and must start with a leading pipe character.

The `savedsearch` command always runs a new search. To reanimate the results of a previously run search, use the `loadjob` command.

When the `savedsearch` command runs a saved search, the command always applies the permissions associated with the role of the person running the `savedsearch` command to the search. The `savedsearch` command never applies the permissions associated with the role of the person who created and owns the search to the search. This happens even when a saved search has been set up to run as the report owner.

See Determine whether to run reports as the report owner or user in the *Reporting Manual*.

Time ranges

- If you specify **All Time** in the time range picker, the `savedsearch` command uses the time range that was saved with the saved search.
- If you specify any other time in the time range picker, the time range that you specify overrides the time range that was saved with the saved search.

Examples

Example 1

Run the saved search "mysecurityquery".

```
| savedsearch mysecurityquery
```

Example2

Run the saved search "mysearch". Where the replacement placeholder term `$replace_me$` appears in the saved search, use "value" instead.

```
| savedsearch mysearch replace_me="value"...
```

See also

[search](#), [loadjob](#)

script

Description

Calls an external python program that can modify or generate search results.

Splunk Cloud Platform

You must create a private app that contains your custom script. If you are a Splunk Cloud administrator with experience creating private apps, see [Manage private apps](#) in your Splunk Cloud Platform deployment in the *Splunk Cloud Admin Manual*. If you have not created private apps, contact your Splunk account representative for help with this customization.

Splunk Enterprise

Scripts must be declared in the `commands.conf` file and be located in the `$SPLUNK_HOME/etc/apps/<app_name>/bin/` directory. The script is executed using `$SPLUNK_HOME/bin/python`.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See [SPL safeguards for risky commands](#) in [Securing the Splunk Platform](#).

Syntax

```
script <script-name> [<script-arg>...] [maxinputs=<int>]
```

Required arguments

script-name

Syntax: <string>

Description: The name of the scripted search command to run, as defined in the commands.conf file.

Optional arguments

maxinputs

Syntax: maxinputs=<int>

Description: Specifies how many of the input results are passed to the script per invocation of the command. The `script` command is invoked repeatedly in increments according to the `maxinputs` argument until the search is complete and all of the results have been displayed. Do not change the value of `maxinputs` unless you know what you are doing.

Default: 50000

script-arg

Syntax: <string> ...

Description: One or more arguments to pass to the script. If you are passing multiple arguments, delimit each argument with a space.

Usage

The `script` command is effectively an alternative way to invoke custom search commands. See Create custom search commands for apps in Splunk Cloud Platform or Splunk Enterprise in the *Developer Guide* on the Developer Portal.

The following search:

```
| script commandname
```

is the same as this search:

```
| commandname
```

Some functions of the `script` command have been removed over time. The explicit choice of Perl or Python as an argument is no longer functional and such an argument is ignored. If you need to write Perl search commands, you must declare them as Perl in the `commands.conf` file. This is not recommended, as you need to determine a number of underdocumented things about the input and output formats. Additionally, support for the `etc/searchscripts` directory has been removed. Search commands must be located in the `bin` directory of an app in your Splunk deployment. For more information about creating custom search commands for apps in Splunk Cloud Platform or Splunk Enterprise, see the *Developer Guide* for Splunk Cloud Platform and Splunk Enterprise.

Examples

Example 1:

Run the Python script "myscript" with arguments, myarg1 and myarg2; then, email the results.

```
... | script myscript myarg1 myarg2 | sendemail to=david@splunk.com
```

scrub

Description

Anonymizes the search results by replacing identifying data - usernames, ip addresses, domain names, and so forth - with fictional values that maintain the same word length. For example, it might turn the string `user=carol@adalberto.com` into `user=aname@mycompany.com`. This lets Splunk users share log data without revealing confidential or personal information.

See the **Usage** section for more information.

Syntax

```
scrub [public-terms=<filename>] [private-terms=<filename>] [name-terms=<filename>] [dictionary=<filename>]  
[timeconfig=<filename>] [namespace=<string>]
```

Required arguments

None

Optional arguments

public-terms

Syntax: public-terms=<filename>

Description: Specify a filename that includes the public terms NOT to anonymize.

private-terms

Syntax: private-terms=<filename>

Description: Specify a filename that includes the private terms to anonymize.

name-terms

Syntax: name-terms=<filename>

Description: Specify a filename that includes the names to anonymize.

dictionary

Syntax: dictionary=<filename>

Description: Specify a filename that includes a dictionary of terms NOT to anonymize, unless those terms are in the `private-terms` file.

timeconfig

Syntax: timeconfig=<filename>

Description: Specify a filename that includes the time configurations to anonymize.

namespace

Syntax: namespace=<string>

Description: Specify an application that contains the alternative files to use for anonymizing, instead of using the built-in anonymizing files.

Usage

By default, the `scrub` command uses the dictionary and configuration files that are located in the `$SPLUNK_HOME/etc/anonymizer` directory. These default files can be overridden by specifying arguments to the `scrub` command. The arguments exactly correspond to the settings in the `splunk anonymize` CLI command. For details, issue the `splunk help anonymize` command.

You can add your own versions of the configuration files to the default location.

Alternatively, you can specify an application where you maintain your own copy of the dictionary and configuration files. To specify the application, use the `namespace=<string>` argument, where `<string>` is the name of the application that corresponds to the name that appears in the path `$SPLUNK_HOME/etc/apps/<app>/anonymizer`.

If the `$SPLUNK_HOME/etc/apps/<app>/anonymizer` directory does not exist, the Splunk software looks for the files in the `$SPLUNK_HOME/etc/slave-apps/<app>/anonymizer` directory.

The `scrub` command anonymizes all attributes, except those that start with underscore (`_`) except `_raw`) or start with `date_`. Additionally, the following attributes are not anonymized: `eventtype`, `linecount`, `punct`, `sourcetype`, `timeendpos`, `timestartpos`.

The `scrub` command adheres to the default `maxresultrows` limit of 50000 results. This setting is documented in the `limits.conf` file in the `[searchresults]` stanza. See `limits.conf` in the *Admin Manual*.

Examples

1. Anonymize the current search results using the default files.

```
... | scrub
```

2. Anonymize the current search results using the specified private-terms file.

This search uses the `abc_private-terms` file that is located in the `$SPLUNK_HOME/etc/anonymizer` directory.

```
... | scrub private-file=abc_private-terms
```

search

Description

Use the `search` command to retrieve events from indexes or filter the results of a previous search command in the pipeline. You can retrieve events from your indexes, using keywords, quoted phrases, wildcards, and field-value expressions. The `search` command is implied at the beginning of any search. You do not need to specify the `search` command at the beginning of your search criteria.

You can also use the `search` command later in the search pipeline to filter the results from the previous command in the pipeline.

The search command can also be used in a subsearch. See about subsearches in the *Search Manual*.

After you retrieve events, you can apply commands to transform, filter, and report on the events. Use the vertical bar (|) , or pipe character, to apply a command to the retrieved events.

The `search` command supports IPv4 and IPv6 addresses and subnets that use CIDR notation.

Syntax

`search <logical-expression>`

Required arguments

`<expression>`

Syntax: `<logical-expression> | <time-opts> | <search-modifier> | NOT <logical-expression> | <index-expression> | <comparison-expression> | <logical-expression> [OR] <logical-expression>`

Description: Includes all keywords or field-value pairs used to describe the events to retrieve from the index.

Include parenthesis as necessary. Use Boolean expressions, comparison operators, time modifiers, search modifiers, or combinations of expressions for this argument.

The AND operator is always implied between terms and expressions. For example, `web error` is the same as `web AND error`. Specifying `clientip=192.0.2.255 earliest=-1h@h` is the same as `clientip=192.0.2.255 AND earliest=-1h@h`. So unless you want to include it for clarity reasons, you do not need to specify the AND operator.

Logical expression options

`<comparison-expression>`

Syntax: `<field><comparison-operator><value> | <field> IN (<value-list>)`

Description: Compare a field to a literal value or provide a list of values that can appear in the field.

`<index-expression>`

Syntax: `"<string>" | <term> | <search-modifier>`

Description: Describe the events you want to retrieve from the index using literal strings and search modifiers.

`<time-opts>`

Syntax: `[<timeformat>] (<time-modifier>...)`

Description: Describe the format of the starttime and endtime terms of the search. See [Time options](#).

Comparison expression options

`<comparison-operator>`

Syntax: `= | != | < | <= | > | >=`

Description: You can use comparison operators when searching field/value pairs. Comparison expressions with the equal (=) or not equal (!=) operator compare string values. For example, "1" does not match "1.0". Comparison expressions with greater than or less than operators < > <= >= numerically compare two numbers and lexicographically compare other values. See [Usage](#).

`<field>`

Syntax: `<string>`

Description: The name of a field.

<value>

Syntax: <literal-value>

Description: In comparison-expressions, the literal number or string value of a field.

<value-list>

Syntax: (<literal-value>, <literal-value>, ...)

Description: Used with the IN operator to specify two or more values. For example use `error IN (400, 402, 404, 406)` instead of `error=400 OR error=402 OR error=404 OR error=406`

Index expression options

<string>

Syntax: "<string>"

Description: Specify keywords or quoted phrases to match. When searching for strings and quoted strings (anything that's not a search modifier), Splunk software searches the `_raw` field for the matching events or results.

<search-modifier>

Syntax: <sourcetype-specifier> | <host-specifier> | <hosttag-specifier> | <source-specifier> | <savedsplunk-specifier> | <eventtype-specifier> | <eventtypetag-specifier> | <splunk_server-specifier>

Description: Search for events from specified fields or field tags. For example, search for one or a combination of hosts, sources, source types, saved searches, and event types. Also, search for the field tag, with the format: `tag::<field>=<string>`.

- ◊ Read more about searching with default fields in the *Knowledge Manager manual*.
- ◊ Read more about using tags and field aliases in the *Knowledge Manager manual*.

<sourcetype-specifier>

Syntax: sourcetype=<string>

Description: Search for events from the specified sourcetype field.

<host-specifier>

Syntax: host=<string>

Description: Search for events from the specified host field.

<hosttag-specifier>

Syntax: hosttag=<string>

Description: Search for events that have hosts that are tagged by the string.

<eventtype-specifier>

Syntax: eventtype=<string>

Description: Search for events that match the specified event type.

<eventtypetag-specifier>

Syntax: eventtypetag=<string>

Description: Search for events that would match all eventtypes tagged by the string.

<savedsplunk-specifier>

Syntax: savedsearch=<string> | savedsplunk=<string>

Description: Search for events that would be found by the specified saved search.

<source-specifier>

Syntax: source=<string>

Description: Search for events from the specified source field.

<splunk_server-specifier>

Syntax: splunk_server=<string>

Description: Search for events from a specific server. Use "local" to refer to the search head.

Time options

For a list of time modifiers, see [Time modifiers for search](#).

<timeformat>

Syntax: timeformat=<string>

Description: Set the time format for starttime and endtime terms.

Default: timeformat=%m/%d/%Y:%H:%M:%S.

<time-modifier>

Syntax: starttime=<string> | endtime=<string> | earliest=<time_modifier> | latest=<time_modifier>

Description: Specify start and end times using relative or absolute time.

You can also use the earliest and latest attributes to specify absolute and relative time ranges for your search. For more about this time modifier syntax, see [Specify time modifiers in your search](#) in the [Search Manual](#).

starttime

Syntax: starttime=<string>

Description: Events must be later or equal to this time. Must match `timeformat`.

endtime

Syntax: endtime=<string>

Description: All events must be earlier or equal to this time.

Usage

The `search` command is an **event-generating command** when it is the first command in the search, before the first pipe. When the `search` command is used further down the pipeline, it is a **distributable streaming command**. See [Command types](#).

A subsearch can be initiated through a search command such as the `search` command. See [Initiating subsearches with search commands](#) in the Splunk Cloud Platform *Search Manual*.

The implied search command

The `search` command is implied at the beginning of every search.

When `search` is the first command in the search, you can use terms such as keywords, phrases, fields, boolean expressions, and comparison expressions to specify exactly which events you want to retrieve from Splunk indexes. If you don't specify a field, the search looks for the terms in the `_raw` field.

Some examples of search terms are:

- **keywords:** `error login`, which is the same as specifying `for error AND login`
- **quoted phrases:** `"database error"`
- **boolean operators:** `login NOT (error OR fail)`

- wildcards: `fail*`
- field-value pairs: `status=404`, `status!=404`, or `status>200`

To search field values that are SPL operators or keywords, such as `country=IN`, `country=AS`, `iso=AND`, or `state=OR`, you must enclose the operator or keyword in quotation marks. For example: `country="IN"`.

See Use the `search` command in the *Search Manual*.

Using the search command later in the search pipeline

In addition to the implied `search` command at the beginning of all searches, you can use the `search` command later in the search pipeline. The search terms that you can use depend on which fields are passed into the `search` command.

If the `_raw` field is passed into the `search` command, you can use the same types of search terms as you can when the `search` command is the first command in a search.

However, if the `_raw` field is not passed into the `search` command, you must specify field-values pairs that match the fields passed into the `search` command. Transforming commands, such as `stats` and `chart`, do not pass the `_raw` field to the next command in the pipeline.

Boolean expressions

The order in which Boolean expressions are evaluated with the `search` is:

1. Expressions within parentheses
2. NOT clauses
3. OR clauses
4. AND clauses

This evaluation order is different than the order used with the `where` command. The `where` command evaluates AND clauses before OR clauses.

Comparing two fields

To compare two fields, **do not** specify `index=myindex fieldA=fieldB` or `index=myindex fieldA!=fieldB` with the `search` command. When specifying a `comparison_expression`, the `search` command expects a `<field>` compared with a `<value>`. The `search` command interprets `fieldB` as the value, and not as the name of a field.

Use the `where` command to compare two fields.

```
index=myindex | where fieldA=fieldB
```

For not equal comparisons, you can specify the criteria in several ways.

```
index=myindex | where fieldA!=fieldB
```

or

```
index=myindex | where NOT fieldA=fieldB
```

See Difference between NOT and != in the *Search Manual*.

Multiple field-value comparisons with the IN operator

Use the IN operator when you want to determine if a field contains one of several values.

For example, use this syntax:

```
... error_code IN (400, 402, 404, 406) | ...
```

Instead of this syntax:

```
... error_code=400 OR error_code=402 OR error_code=404 OR error_code=406 | ...
```

When used with the `search` command, you can use a wildcard character in the list of values for the IN operator. For example:

```
... error_code IN (40*) | ...
```

You can use the NOT operator with the IN operator. For example:

```
... NOT clientip IN (211.166.11.101, 182.236.164.11, 128.241.220.82) | ...
```

There is also an IN function that you can use with the `eval` and `where` commands. Wild card characters are not allowed in the values list when the IN function is used with the `eval` and `where` commands. See [Comparison and Conditional functions](#).

CIDR matching

The `search` command can perform a CIDR match on a field that contains IPv4 and IPv6 addresses.

Suppose the `ip` field contains these values:

```
10.10.10.12  
50.10.10.17  
10.10.10.23
```

If you specify `ip="10.10.10.0/24"`, the search returns the events with the first and last values: 10.10.10.12 and 10.10.10.23.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

You can specify a custom sort order that overrides the lexicographical order. See the blog Order Up! Custom Sort Orders.

Quotes and escaping characters

In general, you need quotation marks around phrases and field values that include white spaces, commas, pipes, quotations, and brackets. Quotation marks must be balanced. An opening quotation must be followed by an unescaped closing quotation. For example:

- A search such as `error | stats count` will find the number of events containing the string error.
- A search such as `... | search "error | stats count"` would return the raw events containing error, a pipe, stats, and count, in that order.

Additionally, use quotation marks around keywords and phrases if you don't want to search for their default meaning, such as Boolean operators and field/value pairs. For example:

- A search for the keyword AND without meaning the Boolean operator: `error "AND"`
- A search for this field/value phrase: `error "startswith=foo"`

The backslash character () is used to escape quotes, pipes, and the backslash character itself. Backslash escape sequences are expanded inside quotation marks. For example:

- The sequence `\|` as part of a search sends a pipe character to the command, instead of using the pipe as a split between commands.
- The sequence `\"` sends a literal quotation mark to the command. For example, this is useful if you want to search for a literal quotation mark or insert a literal quotation mark into a field using regular expressions.
- The `\\"` sequence sends a literal backslash to the command.

Unrecognized backslash sequences are not altered:

- For example, `\s` in a search string is available as `\s` to the command, because `\s` is not a known escape sequence.
- However, the search string `\\\s` is available as `\s` to the command, because `\\\` is a known escape sequence that is converted to `\`.

See Backslashes in the *Search Manual*.

Search with TERM()

You can use the TERM() directive to force Splunk software to match whatever is inside the parentheses as a single term in the index. TERM is more useful when the term contains minor segmenters, such as periods, and is bounded by major segmenters, such as spaces or commas. In fact, TERM does not work for terms that are not bounded by major breakers.

See Use CASE and TERM to match phrases in the *Search Manual*.

Search with CASE()

You can use the CASE() directive to search for terms and field values that are case-sensitive.

See Use CASE and TERM to match phrases in the *Search Manual*.

Examples

These examples demonstrate how to use the `search` command. You can find more examples in the Start Searching topic of the Search Tutorial.

1. Field-value pair matching

This example demonstrates field-value pair matching for specific values of source IP (src) and destination IP (dst).

```
src="10.9.165.*" OR dst="10.9.165.8"
```

2. Using boolean and comparison operators

This example demonstrates field-value pair matching with boolean and comparison operators. Search for events with code values of either 10 or 29, and any host that isn't "localhost", and an `xqp` value that is greater than 5.

```
(code=10 OR code=29) host!="localhost" xqp>5
```

In this example you could also use the IN operator since you are specifying two field-value pairs on the same field. The revised search is:

```
code IN(10, 29) host!="localhost" xqp>5
```

3. Using wildcards

This example demonstrates field-value pair matching with wildcards. Search for events from all the web servers that have an HTTP client or server error status.

```
host=webserver* (status=4* OR status=5*)
```

In this example you could also use the IN operator since you are specifying two field-value pairs on the same field. The revised search is:

```
host=webserver* status IN(4*, 5*)
```

4. Using the IN operator

This example shows how to use the IN operator to specify a list of field-value pair matchings. In the events from an `access.log` file, search the `action` field for the values `addtocart` or `purchase`.

```
sourcetype=access_combined_wcookie action IN (addtocart, purchase)
```

5. Specifying a secondary search

This example uses the `search` command twice. The `search` command is implied at the beginning of every search with the criteria `eventtype=web-traffic`. The `search` command is used again later in the search pipeline to filter out the results. This search defines a web session using the `transaction` command and searches for the user sessions that contain more than three events.

```
eventtype=web-traffic | transaction clientip startswith="login" endswith="logout" | search eventcount>3
```

6. Using the NOT or != comparisons

Searching with the boolean "NOT" comparison operator is not the same as using the "!=" comparison.

The following search returns everything except fieldA="value2", including all other fields.

```
NOT fieldA="value2"
```

The following search returns events where `fieldA` exists and does not have the value "value2".

```
fieldA!="value2"
```

If you use a wildcard for the value, `NOT fieldA=*` returns events where fieldA is null or undefined, and `fieldA!=*` never returns any events.

See Difference between NOT and != in the *Search Manual*.

7. Using search to perform CIDR matching

You can use the `search` command to match IPv4 and IPv6 addresses and subnets that use CIDR notation. For example, this search identifies whether the specified IPv4 address is located in the subnet.

```
| makeresults | eval ip="192.0.2.56" | search ip="192.0.2.0/24"
```

The IP address is located in the subnet, so search displays it in the search results, which look like this.

time	ip
2020-11-19 16:43:31	192.0.2.56

Note that you can get identical results using the `eval` command with the `cidrmatch("X",Y)` function, as shown in this example.

```
| makeresults | eval ip="192.0.2.56" | where cidrmatch("192.0.2.0/24", ip)
```

Alternatively, if you're using IPv6 addresses, you can use the `search` command to identify whether the specified IPv6 address is located in the subnet.

```
| makeresults | eval ip="2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99" | search ip="2001:0db8:ffff:ffff:ffff:ffff:ff00/120"
```

The IP address is in the subnet, so the search results look like this.

time	ip
2020-11-19 16:43:31	2001:0db8:ffff:ffff:ffff:ffff:ffff:ff99

See also

Commands

[iplocation](#)
[lookup](#)

Functions

cidrmatch

searchtxn

Description

Efficiently returns transaction events that match a transaction type and contain specific text.

For Splunk Cloud Platform, you must create a private app that contains your transaction type definitions. If you are a Splunk Cloud administrator with experience creating private apps, see [Manage private apps in your Splunk Cloud Platform deployment](#) in the Splunk Cloud Admin Manual. If you have not created private apps, contact your Splunk account representative for help with this customization.

Syntax

```
| searchtxn <transaction-name> [max_terms=<int>] [use_disjunct=<bool>] [eventsonly=<bool>] <search-string>
```

Required arguments

<transaction-name>

Syntax: <transactiontype>

Description: The name of the transaction type stanza that is defined in `transactiontypes.conf`.

<search-string>

Syntax: <string>

Description: Terms to search for within the transaction events.

Optional arguments

eventsonly

Syntax: eventsonly=<bool>

Description: If true, retrieves only the relevant events but does not run "| transaction" command.

Default: false

max_terms

Syntax: maxterms=<int>

Description: Integer between 1-1000 which determines how many unique field values all fields can use. Using smaller values speeds up search, favoring more recent values.

Default: 1000

use_disjunct

Syntax: use_disjunct=<bool>

Description: Specifies if each term in <search-string> should be processed as if separated by an OR operator on the initial search.

Default: true

Usage

The `searchtxn` command is an **event-generating command**. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Transactions

The command works only for transactions bound together by particular field values, not by ordering or time constraints.

Suppose you have a `<transactiontype>` stanza in the `transactiontypes.conf.in` file called "email". The stanza contains the following settings.

- `fields=qid, pid`
- `search=sourcetype=sendmail_syslog to=root`

The `searchtxn` command finds all of the events that match `sourcetype="sendmail_syslog" to=root`.

From those results, all fields that contain a qid or pid located are used to further search for relevant transaction events. When no additional qid or pid values are found, the resulting search is run:

```
sourcetype="sendmail_syslog" ((qid=val1 pid=val1) OR (qid=valn pid=valm)) | transaction name=email | search to=root
```

Examples

Example 1:

Find all email transactions to root from David Smith.

```
| searchtxn email to=root from="David Smith"
```

See also

[transaction](#)

selfjoin

Description

Join search result rows with other search result rows in the same result set, based on one or more fields that you specify.

Syntax

```
selfjoin [<selfjoin-options>...] <field-list>
```

Required arguments

`<field-list>`

Syntax: `<field>...`

Description: The field or list of fields to join on.

Optional arguments

<selfjoin-options>

Syntax: overwrite=<bool> | max=<int> | keepsingle=<bool>

Description: Options that control the search result set that is returned. You can specify one or more of these options.

Selfjoin options

keepsingle

Syntax: keepsingle=<bool>

Description: Controls whether or not to retain results that have a unique value in the join fields and no matching values to join with. When `keepsingle=true`, search results that have no other results to join with are kept in the output. For example, if you're joining results matching employees to their managers, and one of the employees is the CEO who doesn't have a manager, the field for that employee is included in the results when `keepsingle=true`.

Default: false

max

Syntax: max=<int>

Description: Indicates the maximum number of 'other' results to join with each main result. If `max=0`, there is no limit. This argument sets the maximum for the 'other' results. The maximum number of main results is 100,000.

Default: 1

overwrite

Syntax: overwrite=<bool>

Description: When `overwrite=true`, causes fields from the 'other' results to overwrite fields of the main results. The main results are used as the basis for the join.

Default: true

Usage

Self joins are more commonly used with relational database tables. They are used less commonly with event data.

An example of an events usecase is with events that contain information about processes, where each process has a parent process ID. You can use the `selfjoin` command to correlate information about a process with information about the parent process.

See the [Extended example](#).

Basic example

1: Use a single field to join results

Join the results with itself on the 'id' field.

```
... | selfjoin id
```

Extended example

The following example shows how the `selfjoin` command works against a simple set of results. You can follow along with this example on your own Splunk instance.

This example builds a search incrementally. With each addition to the search, the search is rerun and the impact of the additions are shown in a results table. The values in the `_time` field change each time you rerun the search. However, in this example the values in the results table are not changed so that we can focus on how the changes to the search impact the results.

1. Start by creating a simple set of 5 results by using the `makeresults` command.

```
| makeresults count=5
```

There are 5 results created, each with the same timestamp.

<code>_time</code>
2018-01-18 14:38:59
2018-01-18 14:38:59
2018-01-18 14:38:59
2018-01-18 14:38:59
2018-01-18 14:38:59

2. To keep better track of each result use the `streamstats` command to add a field that numbers each result.

```
| makeresults count=5 | streamstats count as a
```

The `a` field is added to the results.

<code>_time</code>	<code>a</code>
2018-01-18 14:38:59	1
2018-01-18 14:38:59	2
2018-01-18 14:38:59	3
2018-01-18 14:38:59	4
2018-01-18 14:38:59	5

3. Additionally, use the `eval` command to change the timestamps to be 60 seconds apart. Different timestamps make this example more realistic.

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a)
```

The minute portion of the timestamp is updated.

<code>_time</code>	<code>a</code>
2018-01-18 14:38:59	1
2018-01-18 14:39:59	2

<u>time</u>	a
2018-01-18 14:40:59	3
2018-01-18 14:41:59	4
2018-01-18 14:42:59	5

4. Next use the `eval` command to create a field to use as the field to join the results on.

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a) | eval joiner="x"
```

The new field is added.

<u>time</u>	a	joiner
2018-01-18 14:38:59	1	x
2018-01-18 14:39:59	2	x
2018-01-18 14:40:59	3	x
2018-01-18 14:41:59	4	x
2018-01-18 14:42:59	5	x

5. Use the `eval` command to create some fields with data.

An `if` function is used with a modulo (modulus) operation to add different data to each of the new fields. A modulo operation finds the remainder after the division of one number by another number:

- The `eval b` command processes each result and performs a modulo operation. If the remainder of $a/2$ is 0, put "something" into the field "b", otherwise put "nada" into field "b".
- The `eval c` command processes each result and performs a modulo operation. If the remainder $a/2$ is 1, put "something else" into the field "c", otherwise put nothing (NULL) into field "c".

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a) | eval joiner="x" | eval b = if(a%2==0,"something","nada"), c = if(a%2==1,"somethingelse",null())
```

The new fields are added and the fields are arranged in alphabetical order by field name, except for the `_time` field.

<u>time</u>	a	b	c	joiner
2018-01-18 14:38:59	1	nada	somethingelse	x
2018-01-18 14:39:59	2	something		x
2018-01-18 14:40:59	3	nada	somethingelse	x
2018-01-18 14:41:59	4	something		x
2018-01-18 14:42:59	5	nada	somethingelse	x

6. Use the `selfjoin` command to join the results on the `joiner` field.

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a) | eval joiner="x" | eval b = if(a%2==0,"something","nada"), c = if(a%2==1,"somethingelse",null()) | selfjoin joiner
```

The results are joined.

_time	a	b	c	joiner
2018-01-18 14:39:59	2	something	somethingelse	x
2018-01-18 14:40:59	3	nada	somethingelse	x
2018-01-18 14:41:59	4	something	somethingelse	x
2018-01-18 14:42:59	5	nada	somethingelse	x

7. To understand how the `selfjoin` command joins the results together, remove the `| selfjoin joiner` portion of the search. Then modify the search to append the values from the `a` field to the values in the `b` and `c` fields.

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a) | eval joiner="x" | eval b = if(a%2==0,"something"+a,"nada"+a), c = if(a%2==1,"somethingelse"+a,null())
```

The results now have the row number appended to the values in the `b` and `c` fields.

_time	a	b	c	joiner
2018-01-18 14:38:59	1	nada1	somethingelse1	x
2018-01-18 14:39:59	2	something2		x
2018-01-18 14:40:59	3	nada3	somethingelse3	x
2018-01-18 14:41:59	4	something4		x
2018-01-18 14:42:59	5	nada5	somethingelse5	x

8. Now add the `selfjoin` command back into the search.

```
| makeresults count=5 | streamstats count as a | eval _time = _time + (60*a) | eval joiner="x" | eval b = if(a%2==0,"something"+a,"nada"+a), c = if(a%2==1,"somethingelse"+a,null()) | selfjoin joiner
```

The results of the self join.

_time	a	b	c	joiner
2018-01-18 14:39:59	2	something2	somethingelse1	x
2018-01-18 14:40:59	3	nada3	somethingelse3	x
2018-01-18 14:41:59	4	something4	somethingelse3	x
2018-01-18 14:42:59	5	nada5	somethingelse5	x

If there are values for a field in both rows, the last result row, based on the `_time` value, takes precedence. The joins performed are shown in the following table.

Result row	Output	Description
1	Row 1 is joined with row 2 and returned as row 2.	In field <code>b</code> , the value <code>nada1</code> is discarded because the value <code>something2</code> in row 2 takes precedence. In field <code>c</code> , there is no value in row 2. The value <code>somethingelse1</code> from row 1 is returned.
2	Row 2 is joined with row 3 and returned as row 3.	Since row 3 contains values for both field <code>b</code> and field <code>c</code> , the values in row 3 take precedence and the values in row 2 are discarded.
3	Row 3 is joined with row 4 and returned as row 4.	In field <code>b</code> , the value <code>nada3</code> is discarded because the value <code>something4</code> in row 4 takes precedence. In field <code>c</code> , there is no value in row 4. The value <code>somethingelse3</code> from row 3 is returned.
4		

Result row	Output	Description
	Row 4 is joined with row 5 and returned as row 5.	Since row 5 contains values for both field <code>b</code> and field <code>c</code> , the values in row 5 take precedence and the values in row 4 are discarded.
5	Row 5 has no other row to join with.	No additional results are returned.

(Thanks to Splunk user Alacercoigitatus for helping with this example.)

See also

[join](#)

sendalert

Description

Use the `sendalert` command to invoke a custom alert action. The command gathers the configuration for the alert action from the `alert_actions.conf` file and the saved search and custom parameters passed using the command arguments. Then the command performs token replacement. The command determines the alert action script and arguments to run, creates the alert action payload and executes the script, handing over the payload by using STDIN to the script process.

When running the custom script, the `sendalert` command honors the `maxtime` setting from the `alert_actions.conf` file and terminates the process if the process runs longer than the configured threshold. By default the threshold is set to 5 minutes.

See Write the script for a custom alert action for Splunk Cloud Platform or Splunk Enterprise in the *Developer Guide* on the Developer Portal.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

`sendalert <alert_action_name> [results_link=<url>] [results_path=<path>] [param.<name>=<"value">...]`

Required arguments

`alert_action_name`

Syntax: `<alert_action_name>`

Description: The name of the alert action configured in the `alert_actions.conf` file

Optional arguments

`results_link`

Syntax: `results_link=<url>`

Description: Set the URL link to the search results.

`results_path`

Syntax: `results_path=<path>`

Description: Set the location to the file containing the search results.

`param.<name>`

Syntax: `param.<name>=<"value">`

Description: The parameter name and value. You can use this name and value pair to specify a variety of things, such as a threshold value, a team name, or the text of a message.

Usage

When you use the `sendalert` command in an **ad hoc search**, the command might be called multiple times if there are a large number of search results. This occurs because previewing the search results on the Statistics tab is enabled by default. If you are using an ad hoc search to test the `sendalert` command, testing turn off preview to avoid the command being called multiple times.

When the `sendalert` command is included in a saved search, such as a scheduled report or a scheduled search, the command is called only one time.

Capability required

To use this command, you must have a role with the `run_sendalert` capability. See [Define roles on the Splunk platform with capabilities](#).

Search results format

When the `sendalert` command is used in a search or in an alert action, the search results are stored in an archive file in the `dispatch` directory using the CSV format. The file name is `results.csv.gz`. The default format for the search results is SRS, a Splunk-specific binary format for the search results. The CSV format for the archive file is used so that scripts can process the results file. The default SRS format is not designed to be parsed by scripts.

The archived search results format is controlled through the `forceCsvResults` setting. This setting is in the [default] stanza in the `alert_actions.conf` file.

Examples

Example 1: Invoke an alert action without any arguments. The alert action script handles checking whether there are necessary parameters that are missing and report the error appropriately.

```
... | sendalert myaction
```

Example 2: Trigger the hipchat custom alert action and pass in room and message as custom parameters.

```
... | sendalert hipchat param.room="SecOps" param.message="There is a security problem!"
```

Example 3: Trigger the servicenow alert option.

```
... | sendalert servicenow param.severity="3" param.assigned_to="DevOps" param.short_description="Splunk Alert: this is a potential security issue"
```

sendemail

Description

Use the `sendemail` command to generate email notifications. You can email search results to specified email addresses.

You must have a Simple Mail Transfer Protocol (SMTP) server available to send email. An SMTP server is not included with the Splunk instance.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

The required syntax is in **bold**:

```
sendemail to=<email_list>
[from=<email_list>]
[cc=<email_list>]
[bcc=<email_list>]
[subject=<string>]
[format=csv | table | raw]
[inline= <bool>]
[sendresults=<bool>]
[sendpdf=<bool>]
[priority=highest | high | normal | low | lowest]
[server=<string>]
[width_sort_columns=<bool>]
[graceful=<bool>]
[content_type=html | plain]
[message=<string>]
[sendcsv=<bool>]
[use_ssl=<bool>]
[use_tls=<bool>]
[pdfview=<string>]
[papersize=letter | legal | ledger | a2 | a3 | a4 | a5]
[paperorientation=portrait | landscape]
[maxinputs=<int>]
[maxtime=<int> m | s | h | d]
[footer=<string>]
```

Required arguments

to

Syntax: `to=<email_list>`

Description: List of email addresses to send search results to. Specify email addresses in a comma-separated and quoted list. For example: "alex@email.com, maria@email.com, wei@email.com"

The set of domains to which you can send emails can be restricted by the Allowed Domains setting on the Email Settings page. For example, that setting could restrict you to sending emails only to addresses in your organization's email domain.

For more information, see Email notification action in the *Alerting Manual*.

Optional arguments

bcc

Syntax: bcc=<email_list>

Description: Blind courtesy copy line. Specify email addresses in a comma-separated and quoted list.

cc

Syntax: cc=<email_list>

Description: Courtesy copy line. Specify email addresses in a comma-separated and quoted list.

content_type

Syntax: content_type=html | plain

Description: The format type of the email.

Default: The default value for the `content_type` argument is set in the [email] stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `html`.

format

Syntax: format=csv | raw | table

Description: Specifies how to format inline results.

Default: The default value for the `format` argument is set in the [email] stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `table`.

footer

Syntax: footer=<string>

Description: Specify an alternate email footer.

Default: The default footer is:

**If you believe you've received this email in error, please see your Splunk administrator.
splunk > the engine for machine data.**

To force a new line in the footer, use Shift+Enter.

from

Syntax: from=<email_list>

Description: Email address from line.

Default: "splunk@<hostname>"

inline

Syntax: inline=<boolean>

Description: Specifies whether to send the results in the message body or as an attachment. By default, an attachment is provided as a CSV file. See the [Usage](#) section.

Default: The default value for the `inline` argument is set in the [email] stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `false`.

graceful

Syntax: graceful=<boolean>

Description: If set to true, no error is returned if sending the email fails for whatever reason. The remainder of the search continues as if the sendemail command was not part of the search. If `graceful=false` and sending the email fails, the search returns an error.

Default: false

maxinputs

Syntax: maxinputs=<integer>

Description: Sets the maximum number of search results sent via alerts per invocation of the command. The `sendemail` command is invoked repeatedly in increments according to the `maxinputs` argument until the search is complete and all of the results have been displayed. Do not change the value of `maxinputs` unless you know what you are doing.

Default: 50000

maxtime

Syntax: maxtime=<integer>m | s | h | d

Description: The maximum amount of time that the execution of an action is allowed to take before the action is aborted.

Example: 2m

Default: no limit

message

Syntax: message=<string>

Description: Specifies the message sent in the email.

Default: The default message depends on which other arguments are specified with the `sendemail` command.

- ◊ If `sendresults=false` the message defaults to "Search complete."
- ◊ If `sendresults=true`, `inline=true`, and either `sendpdf=false` or `sendcsv=false`, message defaults to "Search results."
- ◊ If `sendpdf=true` or `sendcsv=true`, message defaults to "Search results attached."

paperorientation

Syntax: paperorientation=portrait | landscape

Description: The orientation of the paper.

Default: portrait

papersize

Syntax: papersize=letter | legal | ledger | a2 | a3 | a4 | a5

Description: Default paper size for PDFs. Acceptable values: letter, legal, ledger, a2, a3, a4, a5.

Default: letter

pdfview

Syntax: pdfview=<string>

Description: Name of a `view.xml` file to send as a PDF. For example, `mydashboard.xml`, `search.xml`, or `foo.xml`. Generally this is the name of a dashboard, but it could also be the name of a single page application or some other object. Specify the name only. Do not specify the filename extension. The `view.xml` files are located in `<SPLUNK_HOME>/data/ui/views`.

priority

Syntax: priority=highest | high | normal | low | lowest

Description: Set the priority of the email as it appears in the email client. Lowest or 5, low or 4, high or 2, highest or 1.

Default: normal or 3

sendcsv

Syntax: sendcsv=<boolean>

Description: Specify whether to send the results with the email as an attached CSV file or not.

Default: The default value for the `sendcsv` argument is set in the `[email]` stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `false`.

sendpdf

Syntax: sendpdf=<boolean>

Description: Specify whether to send the results with the email as an attached PDF or not. For more information about generating PDFs, see "Generate PDFs of your reports and dashboards" in the Reporting Manual.

Default: The default value for the `sendpdf` argument is set in the `[email]` stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `false`.

sendresults

Syntax: sendresults=<boolean>

Description: Determines whether the results should be included with the email. See the [Usage](#) section.

Default: The default value for the `sendresults` argument is set in the `[email]` stanza of the `alert_actions.conf` file. The default value for a new or upgraded Splunk installation is `false`.

server

Syntax: server=<host>[:<port>]

Description: If the SMTP server is not local, use this argument to specify the SMTP mail server to use when sending emails. The <host> can be either the hostname or the IP address. You have the option to specify the SMTP <port> that the Splunk instance should connect to.

If you set `use_ssl=true`, you must specify both <host> and <port> in the `server` argument.

This setting takes precedence over the `mailserver` setting in the `alert_actions.conf` file. The default setting for `mailserver` is `localhost:25`.

If an alert action is configured to send an email notification when an alert triggers, the `sendemail` command might not be able to use the server you specify in the `server` argument. The values in the Email domains setting on the Email Settings page might restrict the server you can use. The `sendemail` command uses the Mail host that is set on the Email Settings page. For more information, see [Email notification action in the Alerting Manual](#).

Default: localhost

subject

Syntax: subject=<string>

Description: Specifies the subject line.

Default: "Splunk Results"

use_ssl

Syntax: use_ssl=<boolean>

Description: Specifies whether to use SSL when communicating with the SMTP server. When set to `true`, you must also specify both the <host> and <port> in the `server` argument.

Default: false

use_tls

Syntax: use_tls=<boolean>

Description: Specify whether to use TLS (transport layer security) when communicating with the SMTP server (starttls).

Default: false

width_sort_columns

Syntax: width_sort_columns=<boolean>

Description: This is only valid for plain text emails. Specifies whether the columns should be sorted by their width.

Default: true

Usage

If you set `sendresults=true` and `inline=false` and do not specify `format`, a CSV file is attached to the email.

If you use fields as tokens in your sendemail messages, use the rename command to remove curly brace characters such as { and } from them before they are processed by the sendemail command. The sendemail command cannot interpret curly brace characters when they appear in tokens such as \$results\$.

Capability requirements

To use `sendemail`, your role must have the `schedule_search` and `list_settings` capabilities.

Examples

1: Send search results to the specified email

Send search results to the specified email. By default, the results are formatted as a table.

```
... | sendemail to="elvis@splunk.com" sendresults=true
```

2: Send search results in table format

Send search results in a raw format with the subject "myresults".

```
... | sendemail to="elvis@splunk.com, john@splunk.com" format=raw subject=myresults server=mail.splunk.com  
sendresults=true
```

3. Include a PDF attachment, a message, and raw inline results

Send an email notification with a PDF attachment, a message, and raw inline results.

```
index=_internal | head 5 | sendemail to@example@splunk.com server=mail.example.com subject="Here is an  
email from Splunk" message="This is an example message" sendresults=true inline=true format=raw sendpdf=true
```

4: Use email notification tokens with the sendemail command

You can use the `eval` command in conjunction with email notification tokens to customize your search results emails. The search in the following example sends an email to `sample@splunk.com` with a custom message that says `sample` sendemail message body.

```
|makergesults |eval custommessage="sample sendemail message body" |eval dest="sample@splunk.com" |sendemail  
to="$result.dest$" message="$result.custommessage$"
```

See Use tokens in email notifications in the Splunk Cloud Platform *Alerting Manual*.

set

Description

Performs set operations on **subsearches**.

Syntax

The required syntax is in **bold**.

```
| set (union | diff | intersect) subsearch1 subsearch2
```

Required arguments

union | diff | intersect

Syntax: union | diff | intersect

Description: Performs two subsearches, then executes the specified set operation on the two sets of search results.

Operation	Description
union	Returns a set that combines the results generated by the two subsearches. Provides results that are common to both subsets only once.
diff	Returns a set that combines the results generated by the two subsearches and excludes the events common to both. Does not indicate which subsearch the results originated from.
intersect	Returns a set that contains results common to both subsearches.

subsearch

Syntax: "[" <string> "]"

Description: Specifies a subsearch. Subsearches must be enclosed in square brackets. For more information about subsearch syntax, see "About subsearches" in the *Search Manual*.

Usage

The `set` command is an **event-generating command**. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search.

Results

The `set` command considers results to be the same if all of fields that the results contain match. Some internal fields generated by the search, such as `_serial`, vary from search to search. You need to filter out some of the fields if you are using the `set` command with raw events, as opposed to transformed results such as those from a `stats` command. Typically in these cases, all fields are the same from search to search.

Output limitations

There is a limit on the quantity of results that come out of the invoked subsearches that the `set` command receives to operate on. If this limit is exceeded, the input result set to the `diff` command is silently truncated.

If you have Splunk Enterprise, you can adjust this limit by editing the `limits.conf` file and changing the `maxout` value in the `[subsearch]` stanza. If this value is altered, the default quantity of results coming from a variety of subsearch scenarios are altered. Note that very large values might cause extensive stalls during the 'parsing' phase of a search, which is when subsearches run. The default value for this limit is 10000.

Only users with file system access, such as system administrators, can edit the configuration files. Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make the changes in the local directory.

See How to edit a configuration file.

If you have Splunk Cloud Platform and want to edit a configuration file, file a Support ticket.

Result rows limitations

By default the `set` command attempts to traverse a maximum of 50000 items from each subsearch. If the number of input results from either search exceeds this limit, the `set` command silently ignores the remaining events. By default, the `maxout` setting for subsearches in `limits.conf` prevents the number of results from exceeding this limit.

This maximum is controlled by the `maxresultrows` setting in the `[set]` stanza in the `limits.conf` file. Increasing this limit can result in more memory usage.

Only users with file system access, such as system administrators, can edit the configuration files. Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make the changes in the local directory.

See How to edit a configuration file.

If you have Splunk Cloud Platform and want to edit a configuration file, file a Support ticket.

Examples

Example 1:

Return values of "URL" that contain the string "404" or "303" but not both.

```
| set diff [search 404 | fields url] [search 303 | fields url]
```

Example 2:

Return all urls that have 404 errors and 303 errors.

```
| set intersect [search 404 | fields url] [search 303 | fields url]
```

Note: When you use the `fields` command in your subsearches, it does not filter out internal fields by default. If you do not want the `set` command to compare internal fields, such as the `_raw` or `_time` fields, you need to explicitly exclude them from the subsearches:

```
| set intersect [search 404 | fields url | fields - *_] [search 303 | fields url | fields - *_]
```

See also

[append](#), [appendcols](#), [appendpipe](#), [join](#), [diff](#)

setfields

This feature is deprecated.

The `setfields` command is deprecated in the Splunk platform as of version 9.0.0. It might be removed in a future version. See the Release Notes.

Description

Sets the field values for all results to a common value.

Sets the value of the given fields to the specified values for each event in the result set. Delimit multiple definitions with commas. Missing fields are added, present fields are overwritten.

Whenever you need to change or define field values, you can use the more general purpose `eval` command. See usage of an **eval expression** to set the value of a field in Example 1.

Syntax

```
setfields <setfields-arg>, ...
```

Required arguments

<setfields-arg>

Syntax: string=<string>, ...

Description: A key-value pair, with the value quoted. If you specify multiple key-value pairs, separate each pair with a comma. Standard key cleaning is performed. This means all non-alphanumeric characters are replaced with `'_'` and leading `'_'` are removed.

Examples

Example 1:

Specify a value for the ip and foo fields.

```
... | setfields ip="10.10.10.10", foo="foo bar"
```

To do this with the `eval` command:

```
... | eval ip="10.10.10.10" | eval foo="foo bar"
```

See also

[eval](#), [fillnull](#), [rename](#)

sichart

Summary indexing is a method you can use to speed up long-running searches that *do not* qualify for report acceleration, such as searches that use commands that are not **streamable** before the reporting command. For more information, see "About report acceleration and summary indexing" and "Use summary indexing for increased reporting efficiency" in the *Knowledge Manager Manual*.

Description

The summary indexing version of the `chart` command. The `sichart` command populates a summary index with the statistics necessary to generate a chart visualization. For example, it can create a column, line, area, or pie chart. After you populate the summary index, you can use the `chart` command with the exact same search that you used with the `sichart` command to search against the summary index.

Syntax

Required syntax is in **bold**.

```
sichart
[sep=<string>]
[format=<string>]
[cont=<bool>]
[limit=<int>]
[agg=<stats-agg-term>]
( <stats-agg-term> | <sparkline-agg-term> | "("<eval-expression>"")" )...
[ BY <field> [<bins-options>...] [<split-by-clause>] ] | [ OVER <field> [<bins-options>...] [BY <split-by-clause>] ]
```

For syntax descriptions, refer to the `chart` command.

Usage

Supported functions

You can use a wide range of functions with the `sichart` command. For general information about using functions, see [Statistical and charting functions](#).

- ◊ For a list of functions by category, see [Function list by category](#)
- ◊ For an alphabetical list of functions, see [Alphabetical list of functions](#)

Examples

Example 1:

Compute the necessary information to later do 'chart avg(foo) by bar' on summary indexed results.

```
... | sichtet avg(foo) by bar
```

See also

[chart](#), [collect](#), [overlap](#), [sirare](#), [sistats](#), [sitimechart](#), [sitop](#)

sirare

Summary indexing is a method you can use to speed up long-running searches that *do not* qualify for report acceleration, such as searches that use commands that are not **streamable** before the reporting command. For more information, see "About report acceleration and summary indexing" and "Use summary indexing for increased reporting efficiency" in the *Knowledge Manager Manual*.

Description

The `sirare` command is the summary indexing version of the `rare` command, which returns the least common values of a field or combination of fields. The `sirare` command populates a summary index with the statistics necessary to generate a rare report. After you populate the summary index, use the regular `rare` command with the exact same search string as the `rare` command search to report against it.

Syntax

```
sirare [<top-options>...] <field-list> [<by-clause>]
```

Required arguments

<field-list>

Syntax: <string>...

Description: Comma-delimited list of field names.

Optional arguments

<by-clause>

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

<top-options>

Syntax: countfield=<string> | limit=<int> | percentfield=<string> | showcount=<bool> | showperc=<bool>

Description: Options that specify the type and number of values to display. These are the same <top-options> used by the `rare` and `top` commands.

Top options

countfield

Syntax: countfield=<string>

Description: Name of a new field to write the value of count.

Default: "count"

limit

Syntax: limit=<int>

Description: Specifies how many tuples to return, "0" returns all values.

percentfield

Syntax: percentfield=<string>

Description: Name of a new field to write the value of percentage.

Default: "percent"

showcount

Syntax: showcount=<bool>

Description: Specify whether to create a field called "count" (see "countfield" option) with the count of that tuple.

Default: true

showpercent

Syntax: showpercent=<bool>

Description: Specify whether to create a field called "percent" (see "percentfield" option) with the relative prevalence of that tuple.

Default: true

Examples

Example 1:

Compute the necessary information to later do 'rare foo bar' on summary indexed results.

```
... | sirare foo bar
```

See also

[collect](#), [overlap](#), [sichart](#), [sistats](#), [sitimechart](#), [sitop](#)

sistats

Description

The `sistats` command is one of several commands that you can use to create summary indexes. Summary indexing is one of the methods that you can use to speed up searches that take a long time to run.

The `sistats` command is the summary indexing version of the `stats` command, which calculates aggregate statistics over the dataset.

The `sistats` command populates a summary index. You must then create a report to generate the summary statistics. See the [Usage](#) section.

Syntax

`sistats [allnum=<bool>] [delim=<string>] (<stats-agg-term> | <sparkline-agg-term>) [<by clause>]`

- For descriptions of each of the arguments in this syntax, refer to the `stats` command.
- For information about functions that you can use with the `sistats` command, see [Statistical and charting functions](#).

Usage

The summary indexes exist separately from your main indexes.

After you create the summary index, create a report by running a search against the summary index. You use the exact same search string that you used to populate the summary index, substituting the `stats` command for the `sistats` command, to create your reports.

For more information, see [About report acceleration and summary indexing](#) and [Use summary indexing for increased reporting efficiency](#) in the *Knowledge Manager Manual*.

Statistical functions that are not applied to specific fields

With the exception of the `count` function, when you pair the `sistats` command with functions that are not applied to specific fields or `eval` expressions that resolve into fields, the search head processes it as if it were applied to a wildcard for all fields. In other words, when you have `| sistats avg` in a search, it returns results for `| sistats avg(*)`.

This "implicit wildcard" syntax is officially deprecated, however. Make the wildcard explicit. Write `| sistats <function>(*)` when you want a function to apply to all possible fields.

Memory and sistats search performance

A pair of `limits.conf` settings strike a balance between the performance of `sistats` searches and the amount of memory they use during the search process, in RAM and on disk. If your `sistats` searches are consistently slow to complete you can adjust these settings to improve their performance, but at the cost of increased search-time memory usage, which can lead to search failures.

If you have Splunk Cloud Platform, you need to file a Support ticket to change these settings.

For more information, see [Memory and stats search performance](#) in the *Search Manual*.

Examples

Example 1:

Create a summary index with the statistics about the average, for each hour, of any unique field that ends with the string "lay". For example, `delay`, `xdelay`, `relay`, etc.

```
... | sistats avg(*lay) BY date_hour
```

To create a report, run a search against the summary index using this search

```
index=summary | stats avg(*lay) BY date_hour
```

See also

[collect](#), [overlap](#), [sichart](#), [sirare](#), [sitop](#), [sitimechart](#)

For a detailed explanation and examples of summary indexing, see [Use summary indexing for increased reporting efficiency](#) in the *Knowledge Manager Manual*.

sitimechart

Summary indexing is a method you can use to speed up long-running searches that *do not* qualify for report acceleration, such as searches that use commands that are not **streamable** before the transforming command. For more information, see "About report acceleration and summary indexing" and "Use summary indexing for increased reporting efficiency" in the *Knowledge Manager Manual*.

Description

The `sitimechart` command is the summary indexing version of the `timechart` command, which creates a time-series chart visualization with a corresponding table of statistics. The `sitimechart` command populates a summary index with the statistics necessary to generate a timechart report. After you use an `sitimechart` search to populate the summary index, use the regular `timechart` command with the exact same search string as the `sitimechart` search to report against the summary index.

Syntax

The required syntax is in **bold**.

```
sitimechart
[sep=<string>]
[partial=<bool>]
[cont=<bool>]
[limit=<int>]
[agg=<stats-agg-term>]
[<bin-options>... ]
<single-agg> [BY <split-by-clause>] | <eval-expression> BY <split-by-clause>
```

When specifying `sitimechart` command arguments, either `<single-agg>` or `<eval-expression> BY <split-by-clause>` is required.

For descriptions of each of these arguments, see the [timechart command](#).

Usage

Supported functions

You can use a wide range of functions with the `sitimechart` command. For general information about using functions, see [Statistical and charting functions](#).

- ◊ For a list of functions by category, see [Function list by category](#)
- ◊ For an alphabetical list of functions, see [Alphabetical list of functions](#)

Examples

Example 1:

Use the `collect` command to populate a summary index called `mysummary` with the statistics about CPU usage organized by host,

```
... | sitimechart avg(cpu) BY host | collect index=mysummary
```

The collect command adds the results of a search to a summary index that you specify. You must create the summary index before you invoke the collect command.

Then use the `timechart` command with the same search to generate a timechart report.

```
index=mysummary | timechart avg(cpu) BY host
```

See also

[collect](#), [overlap](#), [sichart](#), [sirare](#), [sistats](#), [sitop](#)

sitop

Summary indexing is a method you can use to speed up long-running searches that *do not* qualify for report acceleration, such as searches that use commands that are not **streamable** before the reporting command. For more information, see *Overview of summary-based search acceleration* and *Use summary indexing for increased reporting efficiency* in the *Knowledge Manager Manual*.

Description

The `sitop` command is the summary indexing version of the `top` command, which returns the most frequent value of a field or combination of fields. The `sitop` command populates a summary index with the statistics necessary to generate a top report. After you populate the summary index, use the regular `top` command with the exact same search string as the `sitop` command search to report against it.

Syntax

```
sitop [<N>] [<top-options>...] <field-list> [<by-clause>]
```

Note: This is the exact same syntax as that of the `top` command.

Required arguments

<field-list>

Syntax: <field>, ...

Description: Comma-delimited list of field names.

Optional arguments

<N>

Syntax: <int>

Description: The number of results to return.

<top-options>

Syntax: countfield=<string> | limit=<int> | otherstr=<string> | percentfield=<string> | showcount=<bool> | showperc=<bool> | useother=<bool>

Description: Options for the `sitop` command. See [Top options](#).

<by-clause>

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

Top options

countfield

Syntax: countfield=<string>

Description: The name of a new field that the value of count is written to.

Default: count

limit

Syntax: limit=<int>

Description: Specifies how many tuples to return, "0" returns all values.

Default: "10"

otherstr

Syntax: otherstr=<string>

Description: If useother is true, specify the value that is written into the row representing all other values.

Default: "OTHER"

percentfield

Syntax: percentfield=<string>

Description: Name of a new field to write the value of percentage.

Default: "percent"

showcount

Syntax: showcount=<bool>

Description: Specify whether to create a field called "count" (see "countfield" option) with the count of that tuple.

Default: true

showperc

Syntax: showperc=<bool>

Description: Specify whether to create a field called "percent" (see "percentfield" option) with the relative prevalence of that tuple.

Default: true

useother

Syntax: useother=<bool>

Description: Specify whether or not to add a row that represents all values not included due to the limit cutoff.

Default: false

Examples

Example 1:

Compute the necessary information to later do 'top foo bar' on summary indexed results.

```
... | sitop foo bar
```

Example 2:

Populate a summary index with the top source IP addresses in a scheduled search that runs daily:

```
eventtype=firewall | sitop src_ip
```

Save the search as, "Summary - firewall top src_ip".

Later, when you want to retrieve that information and report on it, run this search over the past year:

```
index=summary search_name="summary - firewall top src_ip" |top src_ip
```

Additionally, because this search specifies the search name, it filters out other data that have been placed in the summary index by other summary indexing searches.

See also

[collect](#), [overlap](#), [sichart](#), [sirare](#), [sistats](#), [sitimechart](#)

snowincident

The `snowincident` command is used with the Splunk Add-on for ServiceNow.

For information about this command, see Use custom generating search commands for the Splunk Add-on for ServiceNow in *Splunk Add-on for ServiceNow*.

snowincidentstream

The `snowincidentstream` command is used with the Splunk Add-on for ServiceNow.

For information about this command, see Use custom streaming search commands for the Splunk Add-on for ServiceNow in *Splunk Add-on for ServiceNow*.

snowevent

The `snowevent` command is used with the Splunk Add-on for ServiceNow.

For information about this command, see Use custom generating search commands for the Splunk Add-on for ServiceNow in *Splunk Add-on for ServiceNow*.

snoweventstream

The `snoweventstream` command is used with the Splunk Add-on for ServiceNow.

For information about this command, see Use custom streaming search commands for the Splunk Add-on for ServiceNow in *Splunk Add-on for ServiceNow*.

sort

Description

The `sort` command sorts all of the results by the specified fields. Results missing a given field are treated as having the smallest or largest possible value of that field if the order is descending or ascending, respectively.

If the first argument to the `sort` command is a number, then at most that many results are returned, in order. If no number is specified, the default limit of 10000 is used. If the number 0 is specified, all of the results are returned. See the `count` argument for more information.

Syntax

The required syntax is in **bold**.

```
sort
[<count>]
<sort-by-clause>...
[desc]
```

Required arguments

<sort-by-clause>

Syntax: [- | +] <sort-field>, (- | +) <sort-field> ...

Description: List of fields to sort by and the sort order. Use a minus sign (-) for descending order and a plus sign (+) for ascending order. When specifying more than one field, separate the field names with commas. See **Sort field options**.

Optional arguments

<count>

Syntax: <int> | limit=<int>

Description: Specify the number of results to return from the sorted results. If no count is specified, the default limit of 10000 is used. If 0 is specified, all results are returned. You can specify the count using an integer or precede the count with a label, for example `limit=10`.

Using sort 0 might have a negative impact performance, depending on how many results are returned.

Default: 10000

desc

Syntax: d | desc

Description: Reverses the order of the results. If multiple fields are specified, reverses the order of the values in the fields in the order in which the fields are specified. For example, if there are three fields specified, the `desc` argument reverses the order of the values in the first field. For each set of duplicate values in the first field, reverses the order of the corresponding values in the second field. For each set of duplicate values in the second field, reverses the order of the corresponding values in the third field.

Sort field options

<sort-field>

Syntax: <field> | auto(<field>) | str(<field>) | ip(<field>) | num(<field>)

Description: Options you can specify with <sort-field>.

<field>

Syntax: <string>

Description: The name of field to sort.

auto

Syntax: auto(<field>)

Description: Determine automatically how to sort the values of the field.

ip

Syntax: ip(<field>)

Description: Interpret the values of the field as IP addresses.

num

Syntax: num(<field>)

Description: Interpret the values of the field as numbers.

str

Syntax: str(<field>)

Description: Interpret the values of the field as strings and order the values alphabetically.

Usage

The `sort` command is a dataset processing command. See [Command types](#).

By default, `sort` tries to automatically determine what it is sorting. If the field contains numeric values, the collating sequence is numeric. If the field contains on IP address values, the collating sequence is for IP addresses. Otherwise, the collating sequence is in lexicographical order. Some specific examples are:

- Alphabetic strings are sorted lexicographically.
- Punctuation strings are sorted lexicographically.
- Numeric data is sorted as you would expect for numbers and the sort order is specified as ascending or descending.
- Alphanumeric strings are sorted based on the data type of the first character. If the string starts with a number, the string is sorted numerically based on that number alone. Otherwise, strings are sorted lexicographically.
- Strings that are a combination of alphanumeric and punctuation characters are sorted the same way as alphanumeric strings.

The sort order is determined between each pair of values that are compared at any one time. This means that for some pairs of values, the order might be lexicographical, while for other pairs the order might be numerical.

Results in descending order	Description
10.1	This set of values are sorted numerically because the values are all numeric.
9.1	
9.1.a	
10.1.a	This set of values are sorted lexicographically because the values are alphanumeric strings.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

Custom sort order

You can specify a custom sort order that overrides the lexicographical order. See the blog Order Up! Custom Sort Orders.

Basic examples

1. Use the sort field options to specify field types

Sort the results by the `ipaddress` field in ascending order and then sort by the `url` field in descending order.

```
... | sort num(ipaddress), -str(url)
```

2. Specifying the number of results to sort

Sort first 100 results in descending order of the "size" field and then by the "source" value in ascending order. This example specifies the type of data in each of the fields. The "size" field contains numbers and the "source" field contains strings.

```
... | sort 100 -num(size), +str(source)
```

3. Specifying descending and ascending sort orders

Sort results by the `_time` field in ascending order and then by the `"host"` value in descending order.

```
... | sort _time, -host
```

4. Changing the time format of events for sorting

Change the format of the event's time and sort the results in descending order by the Time field that is created with the `eval` command.

```
... | bin _time span=60m | eval Time=strftime(_time, "%m/%d %H:%M %Z") | stats avg(time_taken) AS AverageResponseTime BY Time | sort - Time
```

(Thanks to Splunk user Ayn for this example.)

5. Return the most recent event

Return the most recent event:

```
... | sort 1 -_time
```

6. Use a label with the <count>

You can use a label to identify the number of results to return: Return the first 12 results, sorted by the "host" field in descending order.

```
... | sort limit=12 host
```

Extended example

1. Specify a custom sort order

Sort a table of results in a specific order, such as days of the week or months of the year, that is not lexicographical or numeric. For example, suppose you have a search that produces the following table:

Day	Total
Friday	120
Monday	93
Tuesday	124
Thursday	356
Weekend	1022
Wednesday	248

Sorting on the day field (Day) returns a table sorted alphabetically, which does not make much sense. Instead, you want to sort the table by the day of the week, Monday to Friday, with the Weekend at the end of the list.

To create a custom sort order, you first need to create a field called `sort_field` that defines the order. Then you can sort on that field.

```
... | eval wd=lower(Day) | eval sort_field=case(wd=="monday",1, wd=="tuesday",2, wd=="wednesday",3, wd=="thursday",4, wd=="friday",5, wd=="weekend",6) | sort sort_field | fields - sort_field
```

This search uses the `eval` command to create the `sort_field` and the `fields` command to remove `sort_field` from the final results table.

The results look something like this:

Day	Total
Monday	93
Tuesday	124
Wednesday	248
Thursday	356
Friday	120
Weekend	1022

(Thanks to Splunk users Ant1D and Ziegfried for this example.)

For additional custom sort order examples, see the blog Order Up! Custom Sort Orders and the Extended example in the

`rangemap` command.

See also

[reverse](#)

spath

Description

The `spath` command enables you to extract information from the structured data formats XML and JSON. The command stores this information in one or more fields. The command also highlights the syntax in the displayed events list.

You can also use the `spath()` function with the `eval` command. For more information, see the [evaluation functions](#).

Syntax

```
spath [input=<field>] [output=<field>] [path=<datapath> | <datapath>]
```

Optional arguments

input

Syntax: `input=<field>`

Description: The field to read in and extract values from.

Default: `_raw`

output

Syntax: `output=<field>`

Description: If specified, the value extracted from the path is written to this field name.

Default: If you do not specify an `output` argument, the value for the `path` argument becomes the field name for the extracted value.

path

Syntax: `path=<datapath> | <datapath>`

Description: The location path to the value that you want to extract. The location path can be specified as `path=<datapath>` or as just `datapath`. If you do not specify the `path=`, the first unlabeled argument is used as the location path. A location path is composed of one or more location steps, separated by periods. An example of this is `vendorProductSet.product.desc`. A location step is composed of a field name and an optional index surrounded by curly brackets. The index can be an integer, to refer to the position of the data in an array (this differs between JSON and XML), or a string, to refer to an XML attribute. If the index refers to an XML attribute, specify the attribute name with an `@` symbol.

Usage

The `spath` command is a distributable streaming command. See [Command types](#).

Location path omitted

When used with no `path` argument, the `spath` command runs in "auto-extract" mode. By default, when the `spath` command is in "auto-extract" mode, it finds and extracts all the fields from the first 5,000 characters in the input field. These fields default to `_raw` if another input source is not specified. If a path is provided, the value of this path is extracted to a field named by the path or to a field specified by the output argument, if the output argument is provided.

A location path contains one or more location steps

A location path contains one or more location steps, each of which has a context that is specified by the location steps that precede it. The context for the top-level location step is implicitly the top-level node of the entire XML or JSON document.

The location step is composed of a field name and an optional array index

The location step is composed of a field name and an optional array index indicated by curly brackets around an integer or a string.

Array indices mean different things in XML and JSON. For example, JSON uses zero-based indexing. In JSON, `product.desc{3}` refers to the **fourth** element of the `desc` child of the `product` element. In XML, this same path refers to the **third** `desc` child of `product`.

Using wildcards in place of an array index

The `spath` command lets you use wildcards to take the place of an array index in JSON. Now, you can use the location path `entities.hashtags{} .text` to get the text for all of the hashtags, as opposed to specifying `entities.hashtags{0} .text`, `entities.hashtags{1} .text`, and so on. The referenced path, here `entities.hashtags`, has to refer to an array for this to make sense. Otherwise, you get an error just like with regular array indices.

This also works with XML. For example, `catalog.book` and `catalog.book{}` are equivalent. Both get you all the books in the catalog.

Overriding the spath extraction character limit

By default, the `spath` command extracts all the fields from the first 5,000 characters in the input field. If your events are longer than 5,000 characters and you want to extract all of the fields, you can override the extraction character limit for all searches that use the `spath` command. To change this character limit for all `spath` searches, change the `extraction_cutoff` setting in the `limits.conf` file to a larger value.

If you change the default `extraction_cutoff` setting, you must also change the setting to the same value in all `limits.conf` files across all search head and indexer tiers.

Splunk Cloud Platform

To change the `limits.conf` `extraction_cutoff` setting, use one of the following methods:

- ◊ The Configure limits page in Splunk Web. For more information, see [Configure limits using Splunk Web](#) in the *Splunk Cloud Platform Admin Manual*.
- ◊ The Admin Config Service (ACS) command line interface (CLI). For more information, see [Administer Splunk Cloud Platform using the ACS CLI](#) in the *Splunk Cloud Platform Admin Config Service Manual*.
- ◊ The Admin Config Service (ACS) API. For more information, see [Manage limits.conf configurations](#) in [Splunk Cloud Platform in the Splunk Cloud Platform Admin Config Service Manual](#).

Splunk Enterprise

To change the `extraction_cutoff` setting, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can edit configuration files.
- ◊ Review the steps in [How to edit a configuration file](#) in the Splunk Enterprise *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local` if you are using *nix, or `%SPLUNK_HOME%\etc\system\local` if you are using Windows.
2. In the `[spath]` stanza, add the line `extraction_cutoff = <value>` set to the value you want as the extraction cutoff.
3. If your deployment includes search head or indexer clusters, repeat the previous steps on every indexer peer node or search head cluster member. See [Use the deployer to distribute apps and configuration updates](#) in Splunk Enterprise *Distributed Search* and [Update common peer configurations and apps](#) in Splunk Enterprise *Managing Indexers and Clusters of Indexers* for information about changing the `limits.conf` setting across search head and indexer clusters.

JSON data used with the `spath` command must be well-formed

To use the `spath` command to extract JSON data, ensure that the JSON data is well-formed. For example, string literals other than the literal strings `true`, `false` and `null` must be enclosed in double quotation marks ("). For a full reference on the JSON data format, see the [JSON Data Interchange Syntax](#) standard at <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.

Alternatives to the `spath` command

If you are using `autokv` or index-time field extractions, the path extractions are performed for you at index time.

You do not need to explicitly use the `spath` command to provide a path.

If you are using `indexed_extractions=JSON` or `KV_MODE=JSON` in the `props.conf` file, then you don't need to use the `spath` command.

Basic examples

1. Specify an output field and path

This example shows how to specify an output field and path.

```
... | spath output=myfield path=vendorProductSet.product.desc
```

2. Specify just the <datapath>

For the `path` argument, you can specify the location path with or without the `path=`. In this example the `<datapath>` is `server.name`.

```
... | spath output=myfield server.name
```

3. Specify an output field and path based on an array

For example, you have this array.

```
{  
    "vendorProductSet" : [1,2]  
}
```

To specify the output field and path, use this syntax.

```
... | spath output=myfield path=vendorProductSet{1}
```

4. Specify an output field and a path that uses a nested array

For example, you have this nested array.

```
{  
    "vendorProductSet" : {  
        "product" : [  
            {"desc" : 1},  
            {"locDesc" : 2}  
        ]  
    }  
}
```

To specify the output and path from this nested array, use this syntax.

```
... | spath output=myfield path=vendorProductSet.product{}.locDesc
```

5. Specify the output field and a path for an XML attribute

Use the @ symbol to specify an XML attribute. Consider the following XML list of books and authors.

```
<?xml version="1.0"?>  
<purchases>  
    <book>  
        <author>Martin, George R.R.</author>  
        <title yearPublished=1996>A Game of Thrones</title>  
        <title yearPublished=1998>A Clash of Kings</title>  
    </book>  
    <book>  
        <author>Clarke, Susanna</author>  
        <title yearPublished=2004>Jonathan Strange and Mr. Norrell</title>  
    </book>  
    <book>  
        <author>Kay, Guy Gavriel</author>  
        <title yearPublished=1990>Tigana</title>  
    </book>  
    <book>  
        <author>Bujold, Lois McMaster</author>  
        <title yearPublished=1986>The Warrior's Apprentice</title>  
    </book>  
</purchases>
```

Use this search to return the path for the book and the year it was published.

```
... | spath output=dates path=purchases.book.title{@yearPublished} | table dates
```

In this example, the output is a single multivalue result that lists all of the years the books were published.

Extended examples

1: GitHub

As an administrator of a number of large Git repositories, you want to:

- See who has committed the most changes and to which repository
- Produce a list of the commits submitted for each user

Suppose you are Indexing JSON data using the GitHub PushEvent webhook. You can use the `spath` command to extract fields called `repository`, `commit_author`, and `commit_id`:

```
... | spath output=repository path=repository.url  
... | spath output=commit_author path=commits{}.author.name  
... | spath output=commit_id path=commits{}.id
```

To see who has committed the most changes to a repository, run the search.

```
... | top commit_author by repository
```

To see the list of commits by each user, run this search.

```
... | stats values(commit_id) by commit_author
```

2: Extract a subset of a XML attribute

This example shows how to extract values from XML attributes and elements.

```
<vendorProductSet vendorID="2">  
  <product productID="17" units="mm" >  
    <prodName nameGroup="custom">  
      <locName locale="all">API 01209</locName>  
    </prodName>  
    <desc descGroup="custom">  
      <locDesc locale="es">Precios</locDesc>  
      <locDesc locale="fr">Prix</locDesc>  
      <locDesc locale="de">Preise</locDesc>  
      <locDesc locale="ca">Preus</locDesc>  
      <locDesc locale="pt">Preços</locDesc>  
    </desc>  
  </product>
```

To extract the values of the `locDesc` elements (Precios, Prix, Preise, etc.), use:

```
... | spath output=locDesc path=vendorProductSet.product.desc.locDesc
```

To extract the value of the `locale` attribute (es, fr, de, etc.), use:

```
... | spath output=locDesc.locale path=vendorProductSet.product.desc.locDesc{@locale}
```

To extract the attribute of the 4th locDesc (ca), use:

```
... | spath path=vendorProductSet.product.desc.locDesc{4}{@locale}
```

3: Extract and expand JSON events with multi-valued fields

The `mvexpand` command only works on one multivalued field. This example walks through how to expand a JSON event that has more than one multivalued field into individual events for each field value. For example, given this event with `sourcetype=json`:

```
{
  "widget": {
    "text": [
      {
        "data": "Click here",
        "size": 36
      },
      {
        "data": "Learn more",
        "size": 37
      },
      {
        "data": "Help",
        "size": 38
      }
    ]
  }
}
```

First, start with a search to extract the fields from the JSON. Because no `path` argument is specified, the `spath` command runs in "auto-extract" mode and extracts all of the fields from the first 5,000 characters in the input field. The fields are then renamed and placed in a table.

```
sourcetype=json | spath | rename widget.text.size AS size, widget.text.data AS data | table _time,size,data
```

_time	size	data
2018-10-18 14:45:46.000 BST	36	Click here
	37	Learn more
	38	Help

Then, use the eval function, `mvzip()`, to create a new multivalued field named `x`, with the values of the `size` and `data`:

```
sourcetype=json | spath | rename widget.text.size AS size, widget.text.data AS data | eval
x=mvzip(data,size) | table _time,data,size,x
```

_time	data	size	x
2018-10-18 14:45:46.000 BST	Click here	36	Click here,36
	Learn more	37	Learn more,37
	Help	38	Help,38

Now, use the [mvexpand command](#) to create individual events based on x and the eval function mvindex() to redefine the values for data and size.

```
sourcetype=json | spath | rename widget.text.size AS size, widget.text.data AS data | eval  
x=mvzip(data,size) | mvexpand x | eval x = split(x,",") | eval data=mvindex(x,0) | eval size=mvindex(x,1) |  
table _time,data, size
```

_time	data	size
2018-10-18 14:45:46.000 BST	Click here	36
2018-10-18 14:45:46.000 BST	Learn more	37
2018-10-18 14:45:46.000 BST	Help	38

(Thanks to Splunk user G. Zaimi for this example.)

See also

[extract](#), [kvform](#), [multikv](#), [regex](#), [rex](#), [xmlkv](#), [xpath](#)

stats

Description

Calculates aggregate statistics, such as average, count, and sum, over the results set. This is similar to SQL aggregation. If the `stats` command is used without a `BY` clause, only one row is returned, which is the aggregation over the entire incoming result set. If a `BY` clause is used, one row is returned for each distinct value specified in the `BY` clause.

The `stats` command can be used for several SQL-like operations. If you are familiar with SQL but new to SPL, see [Splunk SPL for SQL users](#).

Difference between stats and eval 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.

Time	Event
7/18/15 6:20:56 ...PM	182.236.164.11 - - [18/July/2015:18:20:56] " GET /cart.do? action=addtocart&itemId=EST-14...
7/18/15 6:21:04 ...PM	182.236.164.11 - - [18/July/2015:18:21:04] " POST /oldlink? itemId=EST18&JSESSIONID=SD6SL8FF10...

Result set before
stats command

... | stats count(eval(method="GET")) as GET by host

host	GET
www1	8413
www2	4654

Result set after
stats command

Syntax

Simple:

stats (stats-function(*field*) [AS *field*])... [BY *field-list*]

Complete:

Required syntax is in **bold**.

```
| stats
[partitions=<num>]
[allnum=<bool>]
[delim=<string>]
( <stats-agg-term>... | <sparkline-agg-term>... )
[<by-clause>]
[<dedup_splitvals>]
```

Required arguments

stats-agg-term

Syntax: <stats-func>(<evalued-field> | <wc-field>) [AS <wc-field>]

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names. For more information on eval expressions, see Types of eval expressions in the *Search Manual*.

sparkline-agg-term

Syntax: <sparkline-agg> [AS <wc-field>]

Description: A sparkline aggregation function. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in the field name.

Optional arguments

allnum

Syntax: allnum=<bool>

Description: If true, computes numerical statistics on each field if and only if all of the values of that field are numerical.

Default: false

by-clause

Syntax: BY <field-list>

Description: The name of one or more fields to group by. You cannot use a wildcard character to specify multiple fields with similar names. You must specify each field separately. The `BY` clause returns one row for each distinct value in the `BY` clause fields. If no `BY` clause is specified, the `stats` command returns only one row, which is the aggregation over the entire incoming result set.

dedup_splitvals

Syntax: dedup_splitvals=<boolean>

Description: Specifies whether to remove duplicate values in multivalued `BY` clause fields.

Default: false

delim

Syntax: delim=<string>

Description: Specifies how the values in the `list()` or `values()` aggregation are delimited.

Default: a single space

partitions

Syntax: partitions=<num>

Description: Partitions the input data based on the split-by fields for multithreaded reduce. The `partitions` argument runs the reduce step (in parallel reduce processing) with multiple threads in the same search process on the same machine. Compare that with parallel reduce, using the `redistribute` command, that runs the reduce step in parallel on multiple machines.

When `partitions=0`, the value of the `partitions` argument is the same as the value of the `default_partitions` setting in the `limits.conf` file.

Default: 0. Set to the same value as the `default_partitions` setting in the `limits.conf` file, which is 1 by default.

Stats function options

stats-func

Syntax: The syntax depends on the function that you use. Refer to the table below.

Description: Statistical and charting functions that you can use with the `stats` command. Each time you invoke the `stats` command, you can use one or more functions. However, you can only use one `BY` clause. See [Usage](#).

The following table lists the supported functions by type of function. Use the links in the table to see descriptions and examples for each function. For an overview about using functions with commands, see [Statistical and charting functions](#).

Type of function	Supported functions and syntax			
Aggregate functions	<code>avg()</code>	<code>exactperc<num>()</code>	<code>perc<num>()</code>	<code>sum()</code>

Type of function	Supported functions and syntax			
	count() distinct_count() estdc() estdc_error()	max() median() min() mode()	range() stdev() stdevp()	sumsq() upperperc<num>() var() varp()
Event order functions	first()	last()		
Multivalue stats and chart functions	list()	values()		
Time functions	earliest() earliest_time()	latest() latest_time()	rate()	

Sparkline function options

Sparklines are inline charts that appear within table cells in search results to display time-based trends associated with the primary key of each row. Read more about how to "Add sparklines to your search results" in the Search Manual.

sparkline-agg

Syntax: sparkline (count(<wc-field>), <span-length>) | sparkline (<sparkline-func>(<wc-field>), <span-length>)

Description: A sparkline specifier, which takes the first argument of a aggregation function on a field and an optional timespan specifier. If no timespan specifier is used, an appropriate timespan is chosen based on the time range of the search. If the sparkline is not scoped to a field, only the count aggregator is permitted. You can use wildcard characters in the field name. See the [Usage](#) section.

sparkline-func

Syntax: c() | count() | dc() | mean() | avg() | stdev() | stdevp() | var() | varp() | sum() | sumsq() | min() | max() | range()

Description: Aggregation function to use to generate sparkline values. Each sparkline value is produced by applying this aggregation to the events that fall into each particular time bin.

Usage

The `stats` command is a **transforming command**. See [Command types](#).

Eval expressions with statistical functions

When you use the `stats` command, you must specify either a statistical function or a sparkline function. When you use a statistical function, you can use an eval expression as part of the statistical function. For example:

```
index=* | stats count(eval(status="404")) AS count_status BY sourcetype
```

Statistical functions that are not applied to specific fields

With the exception of the `count` function, when you pair the `stats` command with functions that are not applied to specific fields or eval expressions that resolve into fields, the search head processes it as if it were applied to a wildcard for all fields. In other words, when you have `| stats avg` in a search, it returns results for `| stats avg(*)`.

This "implicit wildcard" syntax is officially deprecated, however. Make the wildcard explicit. Write `| stats <function>(*)` when you want a function to apply to all possible fields.

Numeric calculations

During calculations, numbers are treated as double-precision floating-point numbers, subject to all the usual behaviors of floating point numbers. If the calculation results in the floating-point special value NaN, it is represented as "nan" in your results. The special values for positive and negative infinity are represented in your results as "inf" and "-inf" respectively. Division by zero results in a null field.

There are situations where the results of a calculation contain more digits than can be represented by a floating-point number. In those situations precision might be lost on the least significant digits. For an example of how to correct this, see Example 2 of the [basic examples](#) for the sigfig(X) function.

Ensure correct search behavior when time fields are missing from input data

Ideally, when you run a `stats` search that aggregates results on a time function such as `latest()`, `latest_time()`, or `rate()`, the search should not return results when `_time` or `_origtime` fields are missing from the input data. However, searches that fit this description return results by default, which means that those results might be incorrect or random.

Correct this behavior by changing the `check_for_invalid_time` setting in `limits.conf` file.

Splunk Cloud Platform

To change the `check_for_invalid_time` setting, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `check_for_invalid_time` setting, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can change the `check_for_invalid_time` setting in the `limits.conf` configuration file.
- ◊ Review the steps in How to edit a configuration file in the [Splunk Enterprise Admin Manual](#).
- ◊ You can have configuration files with the same name in your default, local, and app directories. Read Where you can place (or find) your modified configuration files in the [Splunk Enterprise Admin Manual](#).

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file at `$SPLUNK_HOME/etc/system/local`.
2. Under the `[stats]` stanza, set `check_for_invalid_time` to `true`.

When you set `check_for_invalid_time=true`, the `stats` search processor does not return results for searches on time functions when the input data does not include the `_time` or `_origtime` fields.

Functions and memory usage

Some functions are inherently more expensive, from a memory standpoint, than other functions. For example, the `distinct_count` function requires far more memory than the `count` function. The `values` and `list` functions also can consume a lot of memory.

If you are using the `distinct_count` function without a split-by field or with a low-cardinality split-by by field, consider replacing the `distinct_count` function with the the `estdc` function (estimated distinct count). The `estdc` function might result in significantly lower memory usage and run times.

Memory and stats search performance

A pair of `limits.conf` settings strike a balance between the performance of `stats` searches and the amount of memory they use during the search process, in RAM and on disk. If your `stats` searches are consistently slow to complete you can adjust these settings to improve their performance, but at the cost of increased search-time memory usage, which can lead to search failures.

If you use Splunk Cloud Platform, you need to file a Support ticket to change these settings.

For more information, see Memory and stats search performance in the *Search Manual*.

Event order functions

Using the `first` and `last` functions when searching based on time does not produce accurate results.

- To locate the first value based on time order, use the `earliest` function, instead of the `first` function.
- To locate the last value based on time order, use the `latest` function, instead of the `last` function.

For example, consider the following search.

```
index=test sourcetype=testDb | eventstats first(LastPass) as LastPass, last(_time) as mostRecentTestTime BY testCaseId | where startTime==LastPass OR _time==mostRecentTestTime | stats first(startTime) AS startTime, first(status) AS status, first(histID) AS currentHistId, last(histID) AS lastPassHistId BY testCaseId
```

Replace the `first` and `last` functions when you use the `stats` and `eventstats` commands for ordering events based on time. The following search shows the function changes.

```
index=test sourcetype=testDb | eventstats latest(LastPass) AS LastPass, earliest(_time) AS mostRecentTestTime BY testCaseId | where startTime==LastPass OR _time==mostRecentTestTime | stats latest(startTime) AS startTime, latest(status) AS status, latest(histID) AS currentHistId, earliest(histID) AS lastPassHistId BY testCaseId
```

Wildcards in BY clauses

The `stats` command does not support wildcard characters in field values in BY clauses.

For example, you cannot specify `| stats count BY source*`.

Renaming fields

You cannot rename one field with multiple names. For example if you have field A, you cannot rename A as B, A as C. The following example is not valid.

```
... | stats first(host) AS site, first(host) AS report
```

Basic examples

1. Return the average transfer rate for each host

```
sourcetype=access* | stats avg(kbps) BY host
```

2. Search the access logs, and return the total number of hits from the top 100 values of "referer_domain"

Search the access logs, and return the total number of hits from the top 100 values of "referer_domain". The "top" command returns a count and percent value for each "referer_domain".

```
sourcetype=access_combined | top limit=100 referer_domain | stats sum(count) AS total
```

3. Calculate the average time for each hour for similar fields using wildcard characters

Return the average, for each hour, of any unique field that ends with the string "lay". For example, delay, xdelay, relay, etc.

```
... | stats avg(*lay) BY date_hour
```

4. Remove duplicates in the result set and return the total count for the unique results

Remove duplicates of results with the same "host" value and return the total count of the remaining results.

```
... | stats dc(host)
```

5. In a multivalue BY field, remove duplicate values

For each unique value of `mvfield`, return the average value of `field`. Deduplicates the values in the `mvfield`.

```
... | stats avg(field) BY mvfield dedup_splitvals=true
```

Extended examples

1. Compare the difference between using the stats and chart commands

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 **All time** when you run the search.

This search uses the `stats` command to count the number of events for a combination of HTTP status code values and host:

```
sourcetype=access_* | stats count BY status, host
```

The BY clause returns one row for each distinct value in the BY clause fields. In this search, because two fields are specified in the BY clause, every unique combination of status and host is listed on separate row.

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

status	host	count
200	www.google.com	1000

200	www1	11835
200	www2	11186
200	www3	11261
400	www1	233
400	www2	257
400	www3	211
403	www2	228
404	www1	244
404	www2	209

If you click the **Visualization** tab, the `status` field forms the X-axis and the `host` and `count` fields form the data series. The problem with this chart is that the host values (`www1`, `www2`, `www3`) are strings and cannot be measured in a chart.

Substitute the `chart` command for the `stats` command in the search.

```
sourcetype=access_* | chart count BY status, host
```

With the `chart` command, the two fields specified after the `BY` clause change the appearance of the results on the Statistics tab. The `BY` clause also makes the results suitable for displaying the results in a chart visualization.

- The first field you specify is referred to as the <row-split> field. In the table, the values in this field become the labels for each row. In the chart, this field forms the X-axis.
- The second field you specify is referred to as the <column-split> field. In the table, the values in this field are used as headings for each column. In the chart, this field forms the data series.

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

status	www1	www2	www3
200	11835	11186	11261
400	233	257	211
403	0	288	0
404	244	209	237
406	258	228	224
408	267	243	246
500	225	262	246
503	324	299	329
505	242	0	238

If you click the **Visualization** tab, the `status` field forms the X-axis, the values in the `host` field form the data series, and the Y-axis shows the `count`.

2. Use eval expressions to count the different types of requests against each Web server

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 **All time** when you run the search.

Run the following search to use the `stats` command to determine the number of different page requests, GET and POST, that occurred for each Web server.

```
sourcetype=access_* | stats count(eval(method="GET")) AS GET, count(eval(method="POST")) AS POST BY host
```

This example uses `eval` expressions to specify the different field values for the `stats` command to count.

- The first clause uses the `count()` function to count the Web access events that contain the `method` field value `GET`. Then, using the `AS` keyword, the field that represents these results is renamed `GET`.
- The second clause does the same for `POST` events.
- The counts of both types of events are then separated by the web server, using the `BY` clause with the `host` field.

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

host	GET	POST
www1	8431	5197
www2	8097	4815
www3	8338	4654

You can substitute the `chart` command for the `stats` command in this search. You can then click the Visualization tab to see a chart of the results.

3. Calculate a wide range of statistics by a specific field

Count the number of earthquakes that occurred for each magnitude range

This search 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`), etc., for each earthquake recorded.

You can download a current CSV file from the [USGS Earthquake Feeds](#) and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

Run the following search to calculate the number of earthquakes that occurred in each magnitude range. This data set is comprised of events over a 30-day period.

```
source=all_month.csv | chart count AS "Number of Earthquakes" BY mag span=1 | rename mag AS "Magnitude Range"
```

- This search uses `span=1` to define each of the ranges for the magnitude field, `mag`.
- The [rename command](#) is then used to rename the field to "Magnitude Range".

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

Magnitude Range	

Number of Earthquakes	
-1-0	18
0-1	2088
1-2	3005
2-3	1026
3-4	194
4-5	452
5-6	109
6-7	11
7-8	3

Click the **Visualization** tab to see the result in a chart.

Calculate aggregate statistics for the magnitudes of earthquakes in an area

Search for earthquakes in and around California. Calculate the number of earthquakes that were recorded. Use statistical functions to calculate the minimum, maximum, range (the difference between the min and max), and average magnitudes of the recent earthquakes. List the values by magnitude type.

```
source=all_month.csv place=*California* | stats count, max(mag), min(mag), range(mag), avg(mag) BY magType
```

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

magType	count	max(mag)	min(mag)	range(mag)	avg(mag)
H	123	2.8	0.0	2.8	0.549593
MbLg	1	0	0	0	0.0000000
Md	1565	3.2	0.1	3.1	1.056486
Me	2	2.0	1.6	.04	1.800000
MI	1202	4.3	-0.4	4.7	1.226622
Mw	6	4.9	3.0	1.9	3.650000
ml	10	1.56	0.19	1.37	0.934000

Find the mean, standard deviation, and variance of the magnitudes of the recent quakes

Search for earthquakes in and around California. Calculate the number of earthquakes that were recorded. Use statistical functions to calculate the mean, standard deviation, and variance of the magnitudes for recent earthquakes. List the values by magnitude type.

```
source=usgs place=*California* | stats count mean(mag), stdev(mag), var(mag) BY magType
```

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

magType	count	mean(mag)	std(mag)	var(mag)
H	123	0.549593	0.356985	0.127438
MbLg	1	0.000000	0.000000	0.000000
Md	1565	1.056486	0.580042	0.336449
Me	2	1.800000	0.346410	0.120000
MI	1202	1.226622	0.629664	0.396476
Mw	6	3.650000	0.716240	0.513000
ml	10	0.934000	0.560401	0.314049

The `mean` values should be exactly the same as the values calculated using `avg()`.

4. In a table display items sold by ID, type, and name and calculate the revenue for each product

This example uses the sample dataset from the Search Tutorial and a field lookup to add more information to the event data.

- Download the data set from [Add data tutorial](#) and follow the instructions to load the tutorial data.
- Download the CSV file from [Use field lookups tutorial](#) and follow the instructions to set up the lookup definition to add price and productName to the events.

After you configure the field lookup, you can run this search using the time range, **All time**.

Create a table that displays the items sold at the Buttercup Games online store by their ID, type, and name. Also, calculate the revenue for each product.

```
sourcetype=access_* status=200 action=purchase | stats values(categoryId) AS Type, values(productName) AS "Product Name", sum(price) AS "Revenue" by productId | rename productId AS "Product ID" | eval Revenue="$".tostring(Revenue, "commas")
```

This example uses the `values()` function to display the corresponding `categoryId` and `productName` values for each `productId`. Then, it uses the `sum()` function to calculate a running total of the values of the `price` field.

Also, this example renames the various fields, for better display. For the `stats` functions, the renames are done inline with an "AS" clause. The `rename` command is used to change the name of the `product_id` field, since the syntax does not let you rename a split-by field.

Finally, the results are piped into an `eval` expression to reformat the `Revenue` field values so that they read as currency, with a dollar sign and commas.

This returns the following table of results:

Events	Patterns	Statistics (14)	Visualization
20 Per Page ▾			
Product ID	Type	Product Name	Revenue
BS-AG-G09	ARCADE	Benign Space Debris	\$ 3,348.66
CU-PG-G06	SPORTS	Curling 2014	\$ 2,758.62
DB-SG-G01	STRATEGY	Mediocre Kingdoms	\$ 5,947.62
DC-SG-G02	STRATEGY	Dream Crusher	\$ 8,237.94
FI-AG-G08	ARCADE	Orville the Wolverine	\$ 5,998.50
FS-SG-G03	STRATEGY	Final Sequel	\$ 4,998.00
MB-AG-G07	ARCADE	Manganelli Bros.	\$ 8,357.91

5. Determine how much email comes from each domain

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).

Find out how much of the email in your organization comes from .com, .net, .org or other top level domains.

The `eval` command in this search contains two expressions, separated by a comma.

```
sourcetype="cisco:esa" mailfrom=* | eval accountname=split(mailfrom,"@"),  
from_domain=mvindex(accountname,-1) | stats count(eval(match(from_domain, "[^\n\r\s]+\.\com")) AS ".com",  
count(eval(match(from_domain, "[^\n\r\s]+\.\net")) AS ".net", count(eval(match(from_domain,  
"[^\n\r\s]+\.\org")) AS ".org", count(eval(NOT match(from_domain, "[^\n\r\s]+\.(com|net|org)")) AS "other"
```

- The first part of this search uses the `eval` command to break up the email address in the `mailfrom` field. The `from_domain` is defined as the portion of the `mailfrom` field after the `@` symbol.
 - ◆ 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_domain` to the second value in the multivalue field `accountname`.
- The results are then piped into the `stats` command. The `count()` function is used to count the results of the `eval` expression.
- The `eval` uses the `match()` function to compare the `from_domain` to a regular expression that looks for the different suffixes in the domain. If the value of `from_domain` matches the regular expression, the `count` is updated for each suffix, `.com`, `.net`, and `.org`. Other domain suffixes are counted as `other`.

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

.com	.net	.org	other
4246	9890	0	3543

6. Search Web access logs for the total number of hits from the top 10 referring domains

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.

This example searches the web access logs and return the total number of hits from the top 10 referring domains.

```
sourcetype=access_* | top limit=10 referer
```

This search uses the `top` command to find the ten most common referer domains, which are values of the `referer` field. Some events might use `referer_domain` instead of `referer`. The `top` command returns a count and percent value for each referer.

referer	count	percent
http://www.buttercupgames.com/category.screen?categoryId=STRATEGY	265	6.032324
http://www.buttercupgames.com	209	4.757569
http://www.google.com	163	3.710448
http://www.buttercupgames.com/category.screen?categoryId=NULL	152	3.460050
http://www.buttercupgames.com/product.screen?productId=SF-BVS-G01	127	2.890963
http://www.buttercupgames.com/category.screen?categoryId=ARCADE	122	2.777145
http://www.buttercupgames.com/category.screen?categoryId=ACCESSORIES	118	2.686092
http://www.buttercupgames.com/product.screen?productId=WC-SH-G04	93	2.117004
http://www.buttercupgames.com/category.screen?categoryId=TEE	93	2.117004
http://www.buttercupgames.com/category.screen?categoryId=SHOOTER	81	1.843842

You can then use the `stats` command to calculate a total for the top 10 referrer accesses.

```
sourcetype=access_* | top limit=10 referer | stats sum(count) AS total
```

The `sum()` function adds the values in the `count` to produce the total number of times the top 10 referrers accessed the web site.

New Search

```
sourcetype=access_* | top limit=10 referer | stats sum(count) AS total
```

✓ 4,393 events (3/15/18 12:00:00.000 AM to 3/16/18 12:00:00.000 AM) No Event Sampling ▾

Events Patterns Statistics (1) Visualization

20 Per Page ▾ Format Preview ▾

total
1423

See also

Functions

[Statistical and charting functions](#)

Commands

- [eventstats](#)
- [rare](#)
- [sistats](#)
- [streamstats](#)
- [top](#)

Blogs

[Getting started with stats, eventstats and streamstats](#)

[Search commands > stats, chart, and timechart](#)

[Smooth operator | Searching for multiple field values](#)

strcat

Description

Concatenates string values from 2 or more fields. Combines together string values and literals into a new field. A destination field name is specified at the end of the `strcat` command.

Syntax

```
strcat [allrequired=<bool>] <source-fields> <dest-field>
```

Required arguments

<dest-field>

Syntax: <string>

Description: A destination field to save the concatenated string values in, as defined by the <source-fields> argument. The destination field is always at the end of the series of source fields.

<source-fields>

Syntax: (<field> | <quoted-str>)...

Description: Specify the field names and literal string values that you want to concatenate. Literal values must be enclosed in quotation marks.

quoted-str

Syntax: "<string>"

Description: Quoted string literals.

Examples: "/" or ":"

Optional arguments

allrequired

Syntax: allrequired=<bool>

Description: Specifies whether or not all source fields need to exist in each event before values are written to the destination field. If `allrequired=f`, the destination field is always written and source fields that do not exist are treated as empty strings. If `allrequired=t`, the values are written to destination field only if all source fields exist.

Default: false

Usage

The `strcat` command is a distributable streaming command. See [Command types](#).

Examples

Example 1:

Add a field called `comboIP`, which combines the source and destination IP addresses. Separate the addresses with a forward slash character.

```
... | strcat sourceIP "/" destIP comboIP
```

Example 2:

Add a field called comboIP, which combines the source and destination IP addresses. Separate the addresses with a forward slash character. Create a chart of the number of occurrences of the field values.

```
host="mailserver" | strcat sourceIP "/" destIP comboIP | chart count by comboIP
```

Example 3:

Add a field called address, which combines the host and port values into the format <host>::<port>.

```
... | strcat host "://" port address
```

See also

[eval](#)

streamstats

Description

Adds cumulative summary statistics to all search results in a streaming manner. The `streamstats` command calculates statistics for each event at the time the event is seen. For example, you can calculate the running total for a particular field. The total is calculated by using the values in the specified field for every event that has been processed, up to the current event.

Syntax

The required syntax is in **bold**.

```
streamstats
[reset_on_change=<bool>]
[reset_before="("<eval-expression>")"]
[reset_after="("<eval-expression>")"]
[current=<bool>]
>window=<int>
[time_window=<span-length>]
[global=<bool>]
[allnum=<bool>]
<stats-agg-term>...
[<by-clause>]
```

Required arguments

`stats-agg-term`

Syntax: `<stats-func>(<evalued-field> | <wc-field>) [AS <wc-field>]`

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names. For more information on eval expressions, see Types of eval expressions in the *Search Manual*.

Optional arguments

allnum

Syntax: allnum=<boolean>

Description: If true, computes numerical statistics on each field only if all of the values in that field are numerical.

Default: false

by-clause

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

current

Syntax: current=<boolean>

Description: If true, the search includes the given, or current, event in the summary calculations. If false, the search uses the field value from the previous event.

Default: true

global

Syntax: global=<boolean>

Description: Used only when the `window` argument is set. Defines whether to use a single window, `global=true`, or to use separate windows based on the `by` clause. If `global=false` and `window` is set to a non-zero value, a separate window is used for each group of values of the field specified in the `by` clause.

Default: true

reset_after

Syntax: reset_after="(<eval-expression>)"

Description: After the streamstats calculations are produced for an event, `reset_after` specifies that all of the accumulated statistics are reset if the `eval-expression` returns `true`. The `eval-expression` must evaluate to true or false. The `eval-expression` can reference fields that are returned by the `streamstats` command. When the `reset_after` argument is combined with the `window` argument, the window is also reset when the accumulated statistics are reset.

Default: false

reset_before

Syntax: reset_before="(<eval-expression>)"

Description: Before the streamstats calculations are produced for an event, `reset_before` specifies that all of the accumulated statistics are reset when the `eval-expression` returns `true`. The `eval-expression` must evaluate to true or false. When the `reset_before` argument is combined with the `window` argument, the window is also reset when the accumulated statistics are reset.

Default: false

reset_on_change

Syntax: reset_on_change=<bool>

Description: Specifies that all of the accumulated statistics are reset when the group by fields change. The reset is as if no previous events have been seen. Only events that have all of the group by fields can trigger a reset. Events that have only some of the group by fields are ignored. When the `reset_on_change` argument is combined with the `window` argument, the window is also reset when the accumulated statistics are reset. See the **Usage** section.

Default: false

time_window

Syntax: time_window=<span-length>

Description: Specifies the window size for the `streamstats` calculations, based on time. The `time_window` argument is limited by range of values in the `_time` field in the events. To use the `time_window` argument, the events must be sorted in either ascending or descending time order. You can use the `window` argument with the `time_window` argument to specify the maximum number of events in a window. For the `<span-length>`, to specify five minutes, use `time_window=5m`. To specify 2 days, use `time_window=2d`.

Default: None. However, the value of the `max_stream_window` attribute in the `limits.conf` file applies. The default value is 10000 events.

window

Syntax: `window=<integer>`

Description: Specifies the number of events to use when computing the statistics.

Default: 0, which means that all previous and current events are used.

Stats function options

stats-func

Syntax: The syntax depends on the function that you use. Refer to the table below.

Description: Statistical and charting functions that you can use with the `streamstats` command. Each time you invoke the `streamstats` command, you can use one or more functions. However, you can only use one `BY` clause. See [Usage](#).

The following table lists the supported functions by type of function. Use the links in the table to see descriptions and examples for each function. For an overview about using functions with commands, see [Statistical and charting functions](#).

Type of function	Supported functions and syntax			
Aggregate functions	<code>avg()</code> <code>count()</code> <code>distinct_count()</code> <code>estdc()</code> <code>estdc_error()</code>	<code>exactperc<int>()</code> <code>max()</code> <code>median()</code> <code>min()</code> <code>mode()</code>	<code>perc<int>()</code> <code>range()</code> <code>stdev()</code> <code>stdevp()</code>	<code>sum()</code> <code>sumsq()</code> <code>upperperc<int>()</code> <code>var()</code> <code>varp()</code>
Event order functions	<code>earliest()</code>	<code>first()</code>	<code>last()</code>	<code>latest()</code>
Multivalue stats and chart functions	<code>list(X)</code>	<code>values(X)</code>		

Usage

The `streamstats` command is a centralized streaming command. See [Command types](#).

The `streamstats` command is similar to the `eventstats` command except that it uses events before the current event to compute the aggregate statistics that are applied to each event. If you want to include the current event in the statistical calculations, use `current=true`, which is the default.

The `streamstats` command is also similar to the `stats` command in that `streamstats` calculates summary statistics on search results. Unlike `stats`, which works on the group of results as a whole, `streamstats` calculates statistics for each event at the time the event is seen.

Statistical functions that are not applied to specific fields

With the exception of the `count` function, when you pair the `streamstats` command with functions that are not applied to specific fields or `eval` expressions that resolve into fields, the search head processes it as if it were applied to a wildcard for all fields. In other words, when you have `| streamstats avg` in a search, it returns results for `| stats avg(*)`.

This "implicit wildcard" syntax is officially deprecated, however. Make the wildcard explicit. Write `| streamstats <function>(*)` when you want a function to apply to all possible fields.

Escaping string values

If your `<eval-expression>` contains a value instead of a field name, you must escape the quotation marks around the value.

The following example is a simple way to see this. Start by using the `makeresults` command to create 3 events. Use the `streamstats` command to produce a cumulative count of the events. Then use the `eval` command to create a simple test. If the value of the `count` field is equal to 2, display `yes` in the `test` field. Otherwise display `no` in the `test` field.

```
| makeresults count=3 | streamstats count | eval test=if(count==2,"yes","no")
```

The results appear something like this:

<code>_time</code>	<code>count</code>	<code>test</code>
2017-01-11 11:32:43	1	no
2017-01-11 11:32:43	2	yes
2017-01-11 11:32:43	3	no

Use the `streamstats` command to reset the count when the match is true. You must escape the quotation marks around the word `yes`. The following example shows the complete search.

```
| makeresults count=3 | streamstats count | eval test=if(count==2,"yes","no") | streamstats count as testCount reset_after="("match(test,\\"yes\\")")"
```

Here is another example. You want to look for the value `session is closed` in the **description** field. Because the value is a string, you must enclose it in quotation marks. You then need to escape those quotation marks.

```
... | streamstats reset_after="("description==\"session is closed\")")"
```

The `reset_on_change` argument

You have a dataset with the field "shift" that contains either the value DAY or the value NIGHT. You run this search:

```
...| streamstats count BY shift reset_on_change=true
```

If the dataset is:

```
shift
DAY
DAY
NIGHT
NIGHT
NIGHT
```

NIGHT
DAY
NIGHT

Running the command with `reset_on_change=true` produces the following streamstats results:

```
shift, count
DAY, 1
DAY, 2
NIGHT, 1
NIGHT, 2
NIGHT, 3
NIGHT, 4
DAY, 1
NIGHT, 1
```

Memory and maximum results

The `streamstats` search processor uses two `limits.conf` settings to determine the maximum number of results that it can store in memory for the purpose of computing statistics.

The `maxresultrows` setting specifies a top limit for the `window` argument. This sets the number of result rows that the `streamstats` command processor can store in memory. The `max_mem_usage_mb` setting limits how much memory the `streamstats` command uses to keep track of information.

When the `max_mem_usage_mb` limit is reached, the `streamstats` command processor stops adding the requested fields to the search results.

Do not set `max_mem_usage_mb=0` as this removes the bounds to the amount of memory the `streamstats` command processor can use. This can lead to search failures.

Prerequisites

- Only users with file system access, such as system administrators, can increase the `maxresultrows` and `max_mem_usage_mb` settings using configuration files.
- Review the steps in How to edit a configuration file in the Splunk Enterprise *Admin Manual*.
- You can have configuration files with the same name in your default, local, and app directories. Read Where you can place (or find) your modified configuration files in the Splunk Enterprise *Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

If you have Splunk Cloud Platform and want to change these limits, file a Support ticket.

Basic examples

1. Compute the average of a field over the last 5 events

For each event, compute the average of the `foo` field over the last 5 events, including the current event.

```
... | streamstats avg(foo) window=5
```

This is similar to using the `trendline` command to compute a simple moving average (SMA), such as `trendline sma5(foo)`.

2. Compute the average of a field, with a by clause, over the last 5 events

For each event, compute the average value of `foo` for each value of `bar` including only 5 events, specified by the window size, with that value of `bar`.

```
... | streamstats avg(foo) by bar window=5 global=f
```

3. For each event, add a count of the number of events processed

This example adds to each event a `count` field that represents the number of events seen so far, including that event. For example, it adds 1 for the first event, 2 for the second event, and so on.

```
... | streamstats count
```

If you did not want to include the current event, you would specify:

```
... | streamstats count current=f
```

4. Apply a time-based window to streamstats

Assume that the `max_stream_window` argument in the `limits.conf` file is the default value of 10000 events.

The following search counts the events, using a time window of five minutes.

```
... | streamstats count time_window=5m
```

This search adds a `count` field to each event.

- If the events are in descending time order (most recent to oldest), the value in the `count` field represents the number of events in the next 5 minutes.
- If the events are in ascending time order (oldest to most recent), the `count` field represents the number of events in the previous 5 minutes.

If there are more events in the time-based window than the value for the `max_stream_window` argument, the `max_stream_window` argument takes precedence. The `count` will never be > 10000 , even if there are actually more than 10,000 events in any 5 minute period.

Extended examples

1. Create events for testing

You can use the `streamstats` command with the `makeresults` command to create a series events. This technique is often used for testing search syntax. The `eval` command is used to create events with different hours. You use 3600, the number of seconds in an hour, in the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600)
```

The `streamstats` command is used to create the `count` field. The `streamstats` command calculates a cumulative count for each event, at the time the event is processed.

The results look something like this:

_time	count
2020-01-09 15:35:14	1
2020-01-09 14:35:14	2
2020-01-09 13:35:14	3
2020-01-09 12:35:14	4
2020-01-09 11:35:14	5

Notice that the hours in the timestamp are 1 hour apart.

You can create additional fields by using the `eval` command.

```
| makeresults count=5 | streamstats count | eval _time=_time-(count*3600) | eval age = case(count=1, 25, count=2, 39, count=3, 31, count=4, 27, count=5, null()) | eval city = case(count=1 OR count=3, "San Francisco", count=2 OR count=4, "Seattle", count=5, "Los Angeles")
```

- The `eval` command is used to create two new fields, `age` and `city`. The `eval` command uses the value in the `count` field.
- The `case` function takes pairs of arguments, such as `count=1, 25`. The first argument is a Boolean expression. When that expression is TRUE, the corresponding second argument is returned.

The results of the search look like this:

_time	age	city	count
2020-01-09 15:35:14	25	San Francisco	1
2020-01-09 14:35:14	39	Seattle	2
2020-01-09 13:35:14	31	San Francisco	3
2020-01-09 12:35:14	27	Seattle	4
2020-01-09 11:35:14		Los Angeles	5

2. Calculate a snapshot of summary statistics

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 **All time** when you run the search.

You want to determine the number of the bytes used over a set period of time. The following search uses the first 5 events. Because search results typically display the most recent event first, the `sort` command is used to sort the 5 events in ascending order to see the oldest event first and the most recent event last. Ascending order enables the `streamstats` command to calculate statistics over time.

```
sourcetype=access_combined* | head 5 | sort _time
```

New Search

sourcetype=access_combined* | head 5 | sort _time

✓ 5 events (before 4/10/18 3:28:07.000 PM) No Event Sampling ▾

Events (5) Patterns Statistics Visualization

Format Timeline ▾ - Zoom Out + Zoom to Selection × Deselect

List ▾ Format 20 Per Page ▾

Time	Event
4/9/18 6:20:54.000 PM	182.236.164.11 - - [09/Apr/2018:18:20:54] "GET /category.screen?categoryId=ACCESSORIES&JSESSIONID=pgames.com/oldlink?itemId=EST-17" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.1 host = www1 source = tutorialdata.zip:/www1/access.log sourcetype = access_combined_wcookie
4/9/18 6:20:55.000 PM	182.236.164.11 - - [09/Apr/2018:18:20:55] "POST /oldlink?itemId=EST-18&JSESSIONID=SD6SL8FF18ADFF53" "t.screen?productId=SF-BVS-G01" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (host = www1 source = tutorialdata.zip:/www1/access.log sourcetype = access_combined_wcookie
4/9/18 6:20:56.000 PM	182.236.164.11 - - [09/Apr/2018:18:20:56] "GET /cart.do?action=addtocart&itemId=EST-15&productId=B! "http://www.buttercupgames.com/oldlink?itemId=EST-15" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5" 506 host = www1 source = tutorialdata.zip:/www1/access.log sourcetype = access_combined_wcookie
4/9/18 6:22:15.000 PM	91.205.189.15 - - [09/Apr/2018:18:22:15] "GET /category.screen?categoryId=SHOOTER&JSESSIONID=SD6SL host = www2 source = tutorialdata.zip:/www2/access.log sourcetype = access_combined_wcookie
4/9/18 6:22:16.000 PM	91.205.189.15 - - [09/Apr/2018:18:22:16] "GET /oldlink?itemId=EST-14&JSESSIONID=SD6SL7FF7ADFF53113 "(itemId=EST-14" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/ host = www2 source = tutorialdata.zip:/www2/access.log sourcetype = access_combined_wcookie

< Hide Fields All Fields

SELECTED FIELDS

- a host 2
- a source 2
- a sourcetype 1

INTERESTING FIELDS

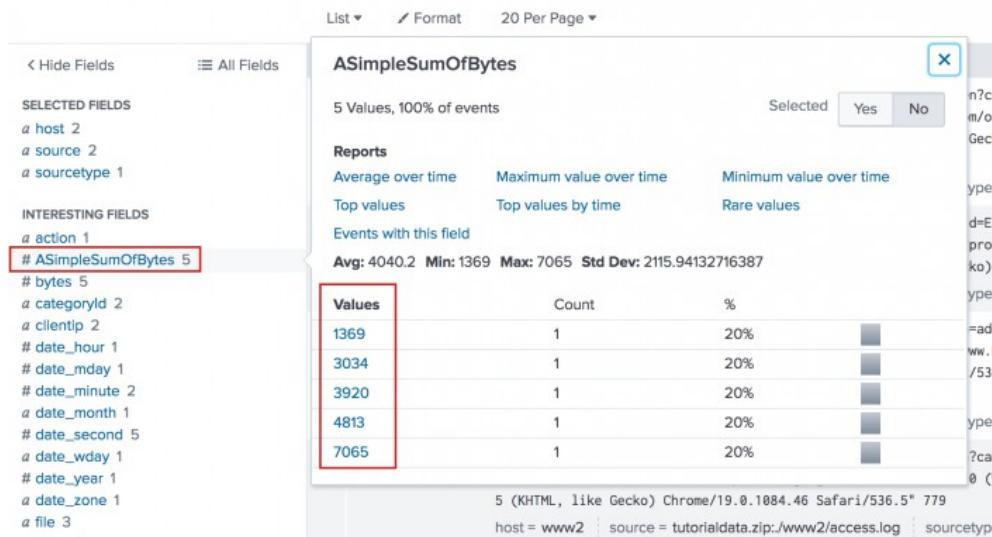
- a action 1
- # bytes 5
- a categoryid 2
- a clientip 2
- # date_hour 1
- # date_mday 1
- # date_minute 2
- a date_month 1
- # date_second 5
- a date_wday 1
- # date_year 1
- a date_zone 1
- a file 3

» Fields 4

Add the `streamstats` command to the search to generate a running total of the bytes over the 5 events and organize the results by `clientip`.

```
sourcetype=access_combined* | head 5 | sort _time | streamstats sum(bytes) AS ASimpleSumOfBytes BY clientip
```

When you click on the `ASimpleSumOfBytes` field in the list of **Interesting fields**, an information window shows the cumulative sum of the bytes, as shown in this image:



The `streamstats` command aggregates the statistics to the original data, which means that all of the original data is accessible for further calculations.

Add the `table` command to the search to display the only the values in the `_time`, `clientip`, `bytes`, and `ASimpleSumOfBytes` fields.

```
sourcetype=access_combined* | head 5 | sort _time | streamstats sum(bytes) as ASimpleSumOfBytes by clientip | table _time, clientip, bytes, ASimpleSumOfBytes
```

Each event shows the timestamp for the event, the `clientip`, and the number of `bytes` used. The `ASimpleSumOfBytes` field shows a cumulative summary of the bytes for each `clientip`.

The screenshot shows the search results table with the following data:

_time	clientip	bytes	ASimpleSumOfBytes
2018-04-09 18:20:54	182.236.164.11	3920	3920
2018-04-09 18:20:55	182.236.164.11	893	4813
2018-04-09 18:20:56	182.236.164.11	2252	7065
2018-04-09 18:22:15	91.205.189.15	1369	1369
2018-04-09 18:22:16	91.205.189.15	1665	3034

3. Calculate the running total of distinct users over time

Each day you track unique users, and you would like to track the cumulative count of distinct users. This example calculates the running total of distinct users over time.

```
eventtype="download" | bin _time span=1d as day | stats values(clientip) as ips dc(clientip) by day | streamstats dc(ips) as "Cumulative total"
```

The `bin` command breaks the time into days. The `stats` command calculates the distinct users (`clientip`) and user count per day. The `streamstats` command finds the running distinct count of users.

This search returns a table that includes: `day`, `ips`, `dc(clientip)`, and `Cumulative total`.

4. Calculate hourly cumulative totals

This example uses `streamstats` to produce hourly cumulative totals.

```
... | timechart span=1h sum(bytes) as SumOfBytes | streamstats global=f sum(*) as accu_total_*
```

This search returns 3 columns: `_time`, `SumOfBytes`, and `accu_total_SumOfBytes`.

The `timechart` command buckets the events into spans of 1 hour and counts the total values for each category. The `timechart` command also fills NULL values, so that there are no missing values. Then, the `streamstats` command is used to calculate the accumulated total.

This example uses `streamstats` to produce hourly cumulative totals for category values.

```
... | timechart span=1h sum(value) as total by category | streamstats global=f | addtotals | accum Total | rename Total as accu_total
```

5. Calculate when a DHCP IP lease address changed for a specific MAC address

This example uses `streamstats` to figure out when a DHCP IP lease address changed for a MAC address, 54:00:00:00:00:00.

```
source=dhcp MAC=54:00:00:00:00:00 | head 10 | streamstats current=f last(DHCP_IP) as new_dhcp_ip last(_time) as time_of_change by MAC
```

You can also clean up the presentation to display a table of the DHCP IP address changes and the times the occurred.

```
source=dhcp MAC=54:00:00:00:00:00 | head 10 | streamstats current=f last(DHCP_IP) as new_dhcp_ip last(_time) as time_of_change by MAC | where DHCP_IP!=new_dhcp_ip | convert ctime(time_of_change) as time_of_change | rename DHCP_IP as old_dhcp_ip | table time_of_change, MAC, old_dhcp_ip, new_dhcp_ip
```

For more details, refer to the Splunk Blogs post for this example.

See also

Commands

`accum`
`autoregress`
`delta`

[fillnull](#)
[eventstats](#)
[makeresults](#)
[trendline](#)

Blogs

Getting started with stats, eventstats and streamstats

table

Description

The `table` command returns a table that is formed by only the fields that you specify in the arguments. Columns are displayed in the same order that fields are specified. Column headers are the field names. Rows are the field values. Each row represents an event.

The `table` command is similar to the [fields](#) command in that it lets you specify the fields you want to keep in your results. Use `table` command when you want to retain data in tabular format.

With the exception of a scatter plot to show trends in the relationships between discrete values of your data, you should not use the `table` command for charts. See [Usage](#).

Syntax

`table <wc-field-list>`

Arguments

`<wc-field-list>`

Syntax: `<wc-field> ...`

Description: A list of valid field names. The list can be space-delimited or comma-delimited. You can use the asterisk (*) as a wildcard to specify a list of fields with similar names. For example, if you want to specify all fields that start with "value", you can use a wildcard such as `value*`.

Usage

The `table` command is a **transforming command**. See [Command types](#).

Visualizations

To generate visualizations, the search results must contain numeric, datetime, or aggregated data such as count, sum, or average.

Command type

The `table` command is a non-streaming command. If you are looking for a streaming command similar to the `table` command, use the [fields](#) command.

Field renaming

The `table` command doesn't let you rename fields, only specify the fields that you want to show in your tabulated results. If you're going to rename a field, do it before piping the results to `table`.

Truncated results

The `table` command truncates the number of results returned based on settings in the `limits.conf` file. In the [search] stanza, if the value for the `truncate_report` parameter is 1, the number of results returned is truncated.

The number of results is controlled by the `max_count` parameter in the [search] stanza. If `truncate_report` is set to 0, the `max_count` parameter is not applied.

Examples

Example 1

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.

Search for recent earthquakes in and around California and display only the time of the quake (`time`), where it occurred (`place`), and the quake's magnitude (`mag`) and depth (`depth`).

```
source=all_month.csv place=*California | table time, place, mag, depth
```

This search reformats your events into a table and displays only the fields that you specified as arguments. The results look something like this:

time	place	mag	depth
2023-03-06T06:45:17.427Z	0 km S of Carnelian Bay, California	0.2	8
2023-03-06T12:49:26.451Z	35 km NE of Independence, California	0.7	0
2023-03-07T09:22:15.281Z	16 km ENE of Doyle, California	0.4	11
2023-03-07T09:37:03.042Z	Northern California	0.4	0
2023-03-07T16:41:29.557Z	27 km ENE of Herlong, California	1	0
2023-03-07T20:57:11.181Z	259 km W of Ferndale, California	3.3	16.554

Example 2

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.

Show the date, time, coordinates, and magnitude of each recent earthquake in Northern California.

```
source=all_month.csv place="Northern California" | rename latitude as lat longitude as lon locationSource as locSource | table time, place, lat, lon, locS*
```

This example begins with a search for all recent earthquakes in Northern California (`place="Northern California"`).

Then the events are piped into the `rename` command to change the names of the coordinate fields, from `latitude` and `longitude` to `lat` and `lon`. The `locationSource` field is also renamed to `locSource`. (The `table` command doesn't let you rename or reformat fields, only specify the fields that you want to show in your tabulated results.)

Finally, the results are piped into the `table` command, which specifies both coordinate fields with `lat` and `lon`, the date and time with `time`, and `locSource` using the asterisk wildcard. The results look something like this:

time	place	lat	lon	locSource
2023-03-03T13:32:16.019Z	Northern California	39.3547	-120.0101	nn
2023-03-07T09:37:03.042Z	Northern California	39.6117	-120.7116	nn
2023-03-09T03:56:40.162Z	Northern California	39.3561	-120.0133	nn
2023-03-01T09:37:57.283Z	Northern California	39.5293	-120.3513	nn
2023-02-21T05:18:39.039Z	Northern California	39.6726	-120.642	nn

Example 3

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 **All time** when you run the search.

Search for IP addresses and classify the network they belong to.

```
sourcetype=access_* | dedup clientip | eval network=if(cidrmatch("192.0.0.0/16", clientip), "local", "other") | table clientip, network
```

This example searches for Web access data and uses the `dedup` command to remove duplicate values of the IP addresses (`clientip`) that access the server. These results are piped into the `eval` command, which uses the `cidrmatch()` function to compare the IP addresses to a subnet range (192.0.0.0/16). This search also uses the `if()` function, which specifies that if the value of `clientip` falls in the subnet range, then the `network` field is given the value `local`. Otherwise, the `network` field is `other`.

The results are then piped into the `table` command to show only the distinct IP addresses (`clientip`) and the network classification (`network`). The results look something like this:

clientip	network
192.0.1.51	other
192.168.11.33	other
192.168.11.44	other
192.168.11.35	other
192.1.2.40	other
192.1.2.35	other

clientip	network
192.0.1.39	local

Example 4

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 **All time** when you run the search.

Create a table with the fields host, action, and all fields that start with date_m*.

```
sourcetype=access_* | table host action date_m*
```

The results look something like this:

host	action	date_mday	date_minute	date_month
www1		20	51	july
www1		20	48	july
www1		20	48	july
www1	addtocart	20	48	july
www1		20	48	july

See Also

[fields](#)

tags

Description

Annotates specified fields in your search results with **tags**. If there are fields specified, only annotates tags for those fields. Otherwise, this command looks for tags for all fields. See About tags and aliases in the *Knowledge Manager Manual*.

Syntax

The required syntax is in **bold**.

```
tags
[outputfield=<field>]
[inclname=<bool>]
[inclvalue=<bool>]
[allowed_tags=<string>]
<field-list>
```

Required arguments

None.

Optional arguments

allowed_tags

Syntax: allowed_tags=<string> | allowed_tags="<string-list>"

Description: If specified, returns only the tag names in the `allowed_tags` argument. You can specify multiple tags using a comma-separated, double-quoted string. For example: `allowed_tags="host, sourcetype"`.

Default: None

<field-list>

Syntax: <field> <field> ...

Description: Specify the fields that you want to output the tags from. The tag names are written to the `outputfield`.

Default: All fields

inlname

Syntax: inlname=true | false

Description: If `outputfield` is specified, this controls whether or not the event field name is added to the output field, along with the tag names. Specify `true` to include the field name.

Default: false

inclvalue

Syntax: inclvalue=true | false

Description: If `outputfield` is specified, controls whether or not the event field value is added to the output field, along with the tag names. Specify `true` to include the event field value.

Default: false

outputfield

Syntax: outputfield=<field>

Description: If specified, the tag names for all of the fields are written to this one new field. If not specified, a new field is created for each field that contains tags. The tag names are written to these new fields using the naming convention `tag_name::<field>`. In addition, a new field is created called `tags` that lists all of the tag names in all of the fields.

Default: New fields are created and the tag names are written to the new fields.

Usage

The `tags` command is a distributable streaming command. See [Command types](#).

Viewing tag information

To view the tags in a table format, use a command before the `tags` command such as the `stats` command. Otherwise, the fields output from the `tags` command appear in the list of **Interesting fields**. See [Examples](#).

Using the <outputfield> argument

If `outputfield` is specified, the tag names for the fields are written to this field. By default, the tag names are written in the format `<field>::<tag_name>`. For example, `sourcetype::apache`.

If `outputfield` is specified, the `inlname` and `inclvalue` arguments control whether or not the field name and field values are added to the `outputfield`. If both `inlname` and `inclvalue` are set to `true`, then the format is `<field>::<value>::<tag_name>`. For example, `sourcetype::access_combined_wcookie::apache`.

Examples

1. Results using the default settings

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 **All time** when you run the search.

This search looks for web access events and counts those events by host.

```
sourcetype=access_* | stats count by host
```

The results look something like this:

host	count
www1	13628
www2	12912
www3	12992

When you use the `tags` command without any arguments, two new fields are added to the results `tag` and `tag::host`.

```
sourcetype=access_* | stats count by host | tags
```

The results look something like this:

host	count	tag	tag::host
www1	13628	tag2	tag2
www2	12912	tag1	tag1
www3	12992		

There are no tags for `host=www3`.

Add the `sourcetype` field to the `stats` command BY clause.

```
sourcetype=access_* | stats count by host sourcetype | tags
```

The results look something like this:

host	sourcetype	count	tag	tag:host	tag::sourcetype
www1	access_combined_wcookie	13628	apache	tag2	apache

host	sourcetype	count	tag	tag:host	tag::sourcetype
			tag2		
www2	access_combined_wcookie	12912	apache tag1	tag1	apache
www3	access_combined_wcookie	12992	apache		apache

The `tag` field list all of the tags used in the events that contain the combination of host and sourcetype.

The `tag::host` field list all of the tags used in the events that contain that host value.

The `tag::sourcetype` field list all of the tags used in the events that contain that sourcetype value.

2. Specifying a list of fields

Return the tags for the `host` and `eventtype` fields.

```
... | tags host eventtype
```

3. Specifying an output field

Write the tags for all fields to the new field `test`.

```
... | stats count by host sourcetype | tags outputfield=test
```

The results look like this:

host	sourcetype	count	test
www1	access_combined_wcookie	13628	apache tag2
www2	access_combined_wcookie	12912	apache tag1
www3	access_combined_wcookie	12992	apache

4. Including the field names in the search results

Write the tags for the `host` and `sourcetype` fields into the `test` field. New fields are returned in the output using the format `host::<tag>` or `sourcetype::<tag>`. Include the field name in the output.

```
... | stats count by host sourcetype | tags outputfield=test inclname=t
```

The results look like this:

host	sourcetype	count	test
www1	access_combined_wcookie	13628	sourcetype::apache

host	sourcetype	count	test
			host::tag2
www2	access_combined_wcookie	12912	sourcetype::apache host::tag1
www3	access_combined_wcookie	12992	sourcetype::apache

5. Identifying a specific a list of tags to return

Write the "error" and "group" tags for the `host` field into the `test` field. New fields are returned in the output using the format `host::<tag>`. Include the field name in the output.

```
index=main | tags outputfield=test inclname=t allowed_tags="error, group" host
```

If you don't have a command before the `tags` command that organizes the results in a table format, you will see the output of the `tags` command in the **Interesting fields** list, as shown in the following image:



Notice that the **tag** field in the list of **Interesting fields** shows that there are 3 tag values. Because the search specified that only the `error` and `group` tags should be returned to the `test` output field, those are the only tag values that appear in the image.

See also

Related information

- [About tags and aliases in the Knowledge Manager Manual](#)
- [Tag field-value pairs in Search in the Knowledge Manager Manual](#)
- [Define and manage tags in Settings in the Knowledge Manager Manual](#)

Commands

[eval](#)

tail

Description

Returns the last N number of specified results. The events are returned in reverse order, starting at the end of the result set. The last 10 events are returned if no integer is specified

Syntax

tail [<N>]

Required arguments

None.

Optional arguments

<N>

Syntax: <int>

Description: The number of results to return.

Default: 10

Usage

The `tail` command is a dataset processing command. See [Command types](#).

Examples

Example 1:

Return the last 20 results in reverse order.

```
... | tail 20
```

See also

[head](#), [reverse](#)

timechart

Description

Creates a time series chart with corresponding table of statistics.

A timechart is a statistical aggregation applied to a field to produce a chart, with time used as the X-axis. You can specify a split-by field, where each distinct value of the split-by field becomes a series in the chart. If you use an `eval` expression, the split-by clause is required. With the `limit` and `agg` options, you can specify series filtering. These options are ignored if you specify an explicit where-clause. If you set `limit=0`, no series filtering occurs.

Syntax

The required syntax is in **bold**.

```
timechart
[sep=<string>]
[format=<string>]
```

```
[partial=<bool>]
[cont=<bool>]
[limit=<chart-limit-opt>]
[agg=<stats-agg-term>]
[<bin-options>... ]
( (<single-agg> [BY <split-by-clause>] ) | (<eval-expression>) BY <split-by-clause> )
[<dedup_splitvals>]
```

Required arguments

When specifying `timechart` command arguments, either `<single-agg>` or `<eval-expression>` BY `<split-by-clause>` is required.

eval-expression

Syntax: `<math-exp> | <concat-exp> | <compare-exp> | <bool-exp> | <function-call>`

Description: A combination of literals, fields, operators, and functions that represent the value of your destination field. For these evaluations to work, your 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.

Additionally, the search can concatenate the two operands if they are both strings. When concatenating values with a period '.' the search treats both values as strings, regardless of their actual data type.

single-agg

Syntax: `count | <stats-func>(<field>)`

Description: A single aggregation applied to a single field, including an evaluated field. For `<stats-func>`, see [Stats function options](#). No wildcards are allowed. The field must be specified, except when using the `count` function, which applies to events as a whole.

split-by-clause

Syntax: `<field> (<tc-options>)... [<where-clause>]`

Description: Specifies a field to split the results by. If field is numerical, default discretization is applied. Discretization is defined with the `<tc-options>`. Use the `<where-clause>` to specify the number of columns to include. See the [tc options](#) and the [where clause](#) sections in this topic.

Optional arguments

agg=<stats-agg-term>

Syntax: `agg=(<stats-func> (<evalued-field> | <wc-field>) [AS <wc-field>])`

Description: A statistical aggregation function. See [Stats function options](#). The function can be applied to an eval expression, or to a field or set of fields. Use the AS clause to place the result into a new field with a name that you specify. You can use wild card characters in field names.

bin-options

Syntax: `bins | minspan | span | <start-end> | aligntime`

Description: Options that you can use to specify discrete bins, or groups, to organize the information. The `bin-options` set the maximum number of bins, not the target number of bins. See the [Bin options](#) section in this topic.

Default: `bins=100`

cont

Syntax: `cont=<bool>`

Description: Specifies whether the chart is continuous or not. If set to `true`, the Search application fills in the time gaps.

Default: true

dedup_splitvals

Syntax: dedup_splitvals=<boolean>

Description: Specifies whether to remove duplicate values in multivalued <split-by-clause> fields.

Default: false

fixedrange

Syntax: fixedrange=<bool>

Description: Specifies whether or not to enforce the earliest and latest times of the search. Setting `fixedrange=false` allows the `timechart` command to constrict or expand to the time range covered by all events in the dataset.

Default: true

format

Syntax: format=<string>

Description: Used to construct output field names when multiple data series are used in conjunction with a split-by-field. `format` takes precedence over `sep` and allows you to specify a parameterized expression with the `stats` aggregator and function (`AGG`) and the value of the split-by-field (`VAL`).

limit

Syntax: limit=(top | bottom)<int>

Description: Specifies a limit for the number of distinct values of the split-by field to return. If set to `limit=0`, all distinct values are used. Setting `limit=N` or `limit=topN` keeps the N highest scoring distinct values of the split-by field. Setting `limit=bottomN` keeps the lowest scoring distinct values of the split-by field. All other values are grouped into 'OTHER', as long as `useother` is not set to false. The scoring is determined as follows:

- ◊ If a single aggregation is specified, the score is based on the sum of the values in the aggregation for that split-by value. For example, for `timechart avg(host) BY <field>`, the `avg(host)` values are added up for each value of `<field>` to determine the scores.
- ◊ If multiple aggregations are specified, the score is based on the frequency of each value of `<field>`. For example, for `timechart avg(host) max(amount) BY <field>`, the top scoring values for `<field>` are the most common values of `<field>`.

Ties in scoring are broken lexicographically, based on the value of the split-by field. For example, 'AMOUNT' takes precedence over 'amount', which takes precedence over 'host'. See [Usage](#).

Default: top10

partial

Syntax: partial=<bool>

Description: Controls if partial time bins should be retained or not. Only the first and last bin can be partial.

Default: True. Partial time bins are retained.

sep

Syntax: sep=<string>

Description: Used to construct output field names when multiple data series are used in conjunctions with a split-by field. This is equivalent to setting `format` to `AGG<sep>VAL`.

Stats function options

stats-func

Syntax: The syntax depends on the function that you use. See [Usage](#).

Description: Statistical functions that you can use with the `timechart` command. Each time you invoke the `timechart` command, you can use one or more functions. However, you can only use one `BY` clause.

Bin options

bins

Syntax: bins=<int>

Description: Sets the *maximum number* of bins to discretize into. This does not set the target number of bins. It finds the smallest bin size that results in no more than N distinct bins. Even though you specify a number such as 300, the resulting number of bins might be much lower.

Default: 100

minspan

Syntax: minspan=<span-length>

Description: Specifies the smallest span granularity to use automatically inferring span from the data time range. See [Usage](#).

span

Syntax: span=<log-span> | span=<span-length> | span=<snap-to-time>

Description: Sets the size of each bin, using either a log-based span, a span length based on time, or a span that snaps to a specific time. For descriptions of each of these options, see [Span options](#).

The starting time of a bin might not match your local timezone. see [Usage](#).

<start-end>

Syntax: end=<num> | start=<num>

Description: Sets the minimum and maximum extents for numerical bins. Data outside of the [start, end] range is discarded.

aligntime

Syntax: aligntime=(earliest | latest | <time-specifier>)

Description: Align the bin times to something other than base UNIX time (epoch 0). The `aligntime` option is valid only when doing a time-based discretization. Ignored if `span` is in days, months, or years.

Span options

<log-span>

Syntax: [<num>]log[<num>]

Description: Sets to log-based span. The first number is a coefficient. The second number is the base. If the first number is supplied, it must be a real number ≥ 1.0 and <base>. Base, if supplied, must be real number > 1.0 (strictly greater than 1).

<span-length>

Syntax: <int>[<timescale>]

Description: A span of each bin, based on time. If the timescale is provided, this is used as a time range. If not, this is an absolute bin length.

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <week> | <month> | <quarter> | <subseconds>

Description: Timescale units.

Default: <sec>

Timescale	Valid syntax	Description
-----------	--------------	-------------

Timescale	Valid syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<week>	w week weeks	Time scale in weeks.
<month>	mon month months	Time scale in months.
<quarter>	q qtr qtrs quarter quarters	Time scale in quarters.
<subseconds>	us ms cs ds	Time scale in microseconds (us), milliseconds (ms), centiseconds (cs), or deciseconds (ds)

<snap-to-time>

Syntax: [+|-] [<time_integer>] <relative_time_unit>@<snap_to_time_unit>

Description: A span of each bin, based on a relative time unit and a snap to time unit. The <snap-to-time> must include a relative_time_unit, the @ symbol, and a snap_to_time_unit. The offset, represented by the plus (+) or minus (-) is optional. If the <time_integer> is not specified, 1 is the default. For example, if you specify w as the relative_time_unit, 1 week is assumed.

tc options

The <tc-option> is part of the <split-by-clause>.

tc-option

Syntax: <bin-options> | usenull=<bool> | useother=<bool> | nullstr=<string> | otherstr=<string>

Description: Timechart options for controlling the behavior of splitting by a field.

bin-options

See the [Bin options](#) section in this topic.

nullstr

Syntax: nullstr=<string>

Description: If usenull=true, specifies the label for the series that is created for events that do not contain the split-by field.

Default: NULL

otherstr

Syntax: otherstr=<string>

Description: If useother=true, specifies the label for the series that is created in the table and the graph.

Default: OTHER

usenull

Syntax: usenull=<bool>

Description: Controls whether or not a series is created for events that do not contain the split-by field. The label for the series is controlled by the nullstr option.

Default: true

useother

Syntax: useother=<bool>

Description: You specify which series to include in the results table by using the <agg>, <limit>, and <where-clause> options. The `useother` option specifies whether to merge all of the series not included in the results table into a single new series. If `useother=true`, the label for the series is controlled by the `otherstr` option.

Default: true

where clause

The <where-clause> is part of the <split-by-clause>. The <where-clause> is comprised of two parts, a single aggregation and some options. See [Where clause examples](#).

where clause

Syntax: <single-agg> <where-comp>

Description: Specifies the criteria for including particular data series when a field is given in the <tc-by-clause>. The most common use of this option is to look for spikes in your data rather than overall mass of distribution in series selection. The default value finds the top ten series by area under the curve. Alternately one could replace sum with max to find the series with the ten highest spikes. Essentially the default is the same as specifying `where sum in top10`. The <where-clause> has no relation to the `where` command.

<where-comp>

Syntax: <wherein-comp> | <wherethresh-comp>

Description: Specify either a grouping for the series or the threshold for the series.

<wherein-comp>

Syntax: (in |notin) (top | bottom)<int>

Description: A grouping criteria that requires the aggregated series value be in or not in some top or bottom group.

<wherethresh-comp>

Syntax: (< | >) [" "] <num>

Description: A threshold criteria that requires the aggregated series value be greater than or less than some numeric threshold. You can specify the threshold with or without a space between the sign and the number.

Usage

The `timechart` command is a **transforming command**. See [Command types](#).

bins and span arguments

The `timechart` command accepts either the `bins` argument OR the `span` argument. If you specify both `bins` and `span`, `span` is used. The `bins` argument is ignored.

If you do not specify either `bins` or `span`, the `timechart` command uses the default `bins=100`.

Default time spans

If you use the predefined time ranges in the time range picker, and do not specify the `span` argument, the following table shows the default span that is used.

Time range	Default span
Last 15 minutes	10 seconds
Last 60 minutes	1 minute
Last 4 hours	5 minutes
Last 24 hours	30 minutes
Last 7 days	1 day
Last 30 days	1 day
Previous year	1 month

(Thanks to Splunk users MuS and Martin Mueller for their help in compiling this default time span information.)

Spans used when minspan is specified

When you specify a `minspan` value, the span that is used for the search must be equal to or greater than one of the span threshold values in the following table. For example, if you specify `minspan=15m` that is equivalent to 900 seconds. The minimum span that can be used is 1800 seconds, or 30 minutes.

Span threshold	Time equivalents
1 second	
5 seconds	
10 seconds	
30 seconds	
60 seconds	1 minute
300 seconds	5 minutes
600 seconds	10 minutes
1800 seconds	30 minutes
3600 seconds	1 hour
86400 seconds	1 day
2592000 seconds	30 days

Bin time spans and local time

The `span` argument always rounds down the starting date for the first bin. There is no guarantee that the bin start time used by the `timechart` command corresponds to your local timezone. In part this is due to differences in daylight savings time for different locales. To use day boundaries, use `span=1d`. Do not use `span=86400s`, or `span=1440m`, or `span=24h`.

Bin time spans versus per_* functions

The functions, `per_day()`, `per_hour()`, `per_minute()`, and `per_second()` are aggregator functions and are not responsible for setting a time span for the resultant chart. These functions are used to get a consistent scale for the data when an explicit span is not provided. The resulting span can depend on the search time range.

For example, `per_hour()` converts the field value so that it is a rate per hour, or `sum()/(<hours in the span>)`. If your chart

span ends up being 30m, it is sum()*2.

If you want the span to be 1h, you still have to specify the argument `span=1h` in your search.

You can do `per_hour()` on one field and `per_minute()` (or any combination of the functions) on a different field in the same search.

Subsecond bin time spans

Subsecond `span` timescales—time spans that are made up of decisconds (ds), centiseconds (cs), milliseconds (ms), or microseconds (us)—should be numbers that divide evenly into a second. For example, 1s = 1000ms. This means that valid millisecond `span` values are 1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, or 500ms. In addition, `span = 1000ms` is not allowed. Use `span = 1s` instead.

Split-by fields

If you specify a split-by field, ensure that you specify the `bins` and `span` arguments *before* the split-by field. If you specify these arguments after the split-by field, Splunk software assumes that you want to control the bins on the split-by field, not on the time axis.

If you use `chart` or `timechart`, you cannot use a field that you specify in a function as your split-by field as well. For example, you will not be able to run:

```
... | chart sum(A) by A span=log2
```

However, you can work around this with an `eval` expression, for example:

```
... | eval A1=A | chart sum(A) by A1 span=log2
```

Prepending VALUE to the names of some fields that begin with underscore (_)

In timechart searches that include a split-by-clause, when search results include a field name that begins with a leading underscore (`_`), Splunk software prepends the field name with `VALUE` and creates as many columns as there are unique entries in the argument of the BY clause. Prepending the string with `VALUE` distinguishes the field from internal fields and avoids naming a column with a leading underscore, which ensures that the field is not hidden in the output schema like most internal fields.

For example, consider the following search:

```
index="_internal" OR index="_audit" | timechart span=1m sum(linecount) by index
```

The results look something like this:

<code>_time</code>	<code>VALUE_audit</code>	<code>VALUE_internal</code>
2023-06-26 21:00:00	1	586
2023-06-26 21:01:00	1	295
2023-06-26 21:02:00	1	555

_time	VALUE_audit	VALUE_internal
-------	-------------	----------------

The columns are displayed in the search results as `VALUE_audit` and `VALUE_internal`.

Supported functions

You can use a wide range of functions with the `timechart` command. For general information about using functions, see [Statistical and charting functions](#).

- For a list of functions by category, see [Function list by category](#)
- For an alphabetical list of functions, see [Alphabetical list of functions](#)

Functions and memory usage

Some functions are inherently more expensive, from a memory standpoint, than other functions. For example, the `distinct_count` function requires far more memory than the `count` function. The `values` and `list` functions also can consume a lot of memory.

If you are using the `distinct_count` function without a split-by field or with a low-cardinality split-by by field, consider replacing the `distinct_count` function with the `estdgc` function (estimated distinct count). The `estdgc` function might result in significantly lower memory usage and run times.

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

You can specify a custom sort order that overrides the lexicographical order. See the blog Order Up! Custom Sort Orders.

Basic Examples

1. Chart the product of the average "CPU" and average "MEM" for each "host"

For each minute, compute the product of the average "CPU" and average "MEM" for each "host".

```
... | timechart span=1m eval(avg(CPU) * avg(MEM)) BY host
```

2. Chart the average of `cpu_seconds` by processor

This example uses an eval expression that includes a statistical function, `avg` to calculate the average of `cpu_seconds` field, rounded to 2 decimal places. The results are organized by the values in the `processor` field. When you use a eval expression with the `timechart` command, you must also use `BY` clause.

```
... | timechart eval(round(avg(cpu_seconds),2)) BY processor
```

3. Chart the average of "CPU" for each "host"

For each minute, calculate the average value of "CPU" for each "host".

```
... | timechart span=1m avg(CPU) BY host
```

4. Chart the average "cpu_seconds" by "host" and remove outlier values

Calculate the average "cpu_seconds" by "host". Remove outlying values that might distort the timechart axis.

```
... | timechart avg(cpu_seconds) BY host | outlier action=tf
```

5. Chart the average "thruput" of hosts over time

```
... | timechart span=5m avg(thruput) BY host
```

6. Chart the eventtypes by source_ip

For each minute, count the eventtypes by `source_ip`, where the count is greater than 10.

```
sshd failed OR failure | timechart span=1m count(eventtype) BY source_ip usenull=f WHERE count>10
```

7. Align the chart time bins to local time

Align the time bins to 5am (local time). Set the span to 12h. The bins will represent 5am - 5pm, then 5pm - 5am (the next day), and so on.

```
... | timechart _time span=12h aligntime=@d+5h
```

8. In a multivalue BY field, remove duplicate values

For each unique value of `mvfield`, return the average value of `field`. Deduplicates the values in the `mvfield`.

```
... | timechart avg(field) BY mvfield dedup_splitval=true
```

9. Rename fields prepended with VALUE

To rename fields with leading underscores that are prepended with `VALUE`, add the following command to your search:

```
... | rename VALUE_* as *
```

The columns in your search results now display without the leading `VALUE_` in the field name.

Extended Examples

1. Chart revenue for the different products

This example uses the sample dataset from the Search Tutorial and a field lookup to add more information to the event data. To try this example for yourself:

- Download the `tutorialdata.zip` file from [this topic in the Search Tutorial](#) and follow the instructions to upload the file to your Splunk deployment.
- Download the `Prices.csv.zip` file from [this topic in the Search Tutorial](#) and follow the instructions to set up your field lookup.

- Use the time range **Yesterday** when you run the search.

The tutorialdata.zip file includes a `productId` field that is the catalog number for the items sold at the Buttercup Games online store. The field lookup uses the `prices.csv` file to add two new fields to your events: `productName`, which is a descriptive name for the item, and `price`, which is the cost of the item.

Chart the revenue for the different products that were purchased yesterday.

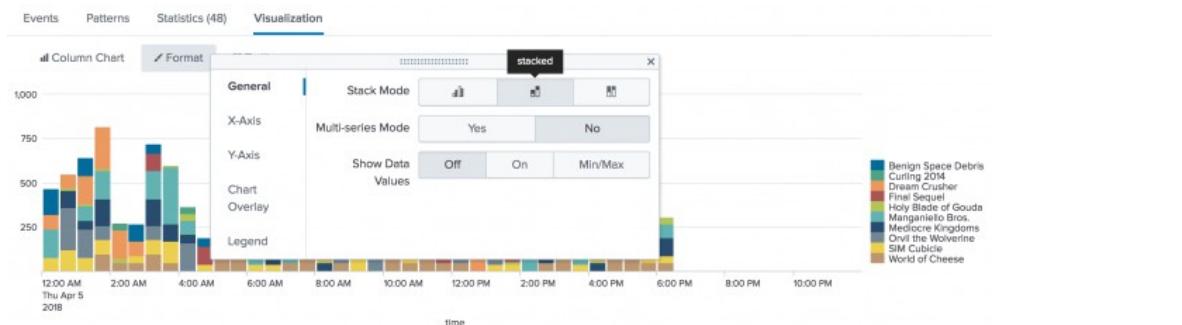
```
sourcetype=access_* action=purchase | timechart per_hour(price) by productName usenull=f useother=f
```

- This example searches for all purchase events (defined by the `action=purchase`).
- The results are piped into `timechart` command.
- The `per_hour()` function sums up the values of the `price` field for each `productName` and organizes the total by time.

This search produces the following table of results in the Statistics tab. To format the numbers to the proper digits for currency, click the format icon in the column heading. On the **Number Formatting** tab, select the **Precision**.

Events		Patterns	Statistics (48)	Visualization
100 Per Page		Format	Preview	
<code>_time</code>	Benign Space Debris	Curling Dream Final Holy Blade Manganiello Mediocre Orvil the Wolverine	Kingdoms	Orville the Wolverine
2018-04-05 00:00:00	149.94			
2018-04-05 00:30:00			99.960000	239.940000
2018-04-05 01:00:00	99.96		49.980000	159.960000
2018-04-05 01:30:00			149.940000	79.980000
2018-04-05 02:00:00	39			
2018-04-05 02:30:00	99.96		149.940000	79.980000
2018-04-05 03:00:00	49.98		99.960000	
2018-04-05 03:30:00				
2018-04-05 04:00:00	39.980000	35.940000 79.980000 49.980000 159.960000		

Click the **Visualization** tab. If necessary, change the chart to a column chart. On the **Format** menu, the General tab contains the Stack Mode option where you can change the chart to a stacked chart.



After you create this chart, you can position your mouse pointer over each section to view more metrics for the product purchased at that hour of the day.

Notice that the chart does not display the data in hourly spans. Because a span is not provided (such as span=1hr), the per_hour() function converts the value so that it is a sum per hours in the time range (which in this example is 24 hours).

2. Chart daily purchases by product type

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

Chart the number of purchases made daily for each type of product.

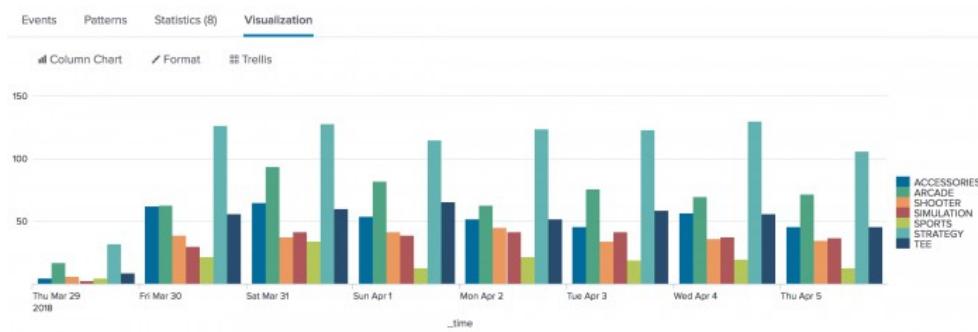
```
sourcetype=access_* action=purchase | timechart span=1d count by categoryId usenull=f
```

- This example searches for all purchases events, defined by the `action=purchase`, and pipes those results into the `timechart` command.
- The `span=1day` argument buckets the count of purchases over the week into daily chunks.
- The `usenull=f` argument ignore any events that contain a NULL value for `categoryId`.

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

_time	ACCESSORIES	ARCADE	SHOOTER	SIMULATION	SPORTS	STRATEGY	TEE
2018-03-29	5	17	6	3	5	32	9
2018-03-30	62	63	39	30	22	127	56
2018-03-31	65	94	38	42	34	128	60
2018-04-01	54	82	42	39	13	115	66
2018-04-02	52	63	45	42	22	124	52
2018-04-03	46	76	34	42	19	123	59
2018-04-04	57	70	36	38	20	130	56
2018-04-05	46	72	35	37	13	106	46

Click the **Visualization** tab. If necessary, change the chart to a column chart.



Compare the number of different items purchased each day and over the course of the week.

3. Display results in 1 week intervals

This search 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), etc., for each earthquake recorded.

You can download a current CSV file from the **USGS Earthquake Feeds** and upload the file to your Splunk instance. This example uses the **All Earthquakes** data from the past 30 days.

This search counts the number of earthquakes in Alaska where the magnitude is greater than or equal to 3.5. The results are organized in spans of 1 week, where the week begins on Monday.

```
source=all_month.csv place=*alaska* mag>=3.5 | timechart span=w@w1 count BY mag
```

- The <by-clause> is used to group the earthquakes by magnitude.
- You can only use week spans with the snap-to span argument in the `timechart` command. For more information, see [Specify a snap to time unit](#).

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

_time	3.5	3.6	3.7	3.8	4	4.1	4.1	4.3	4.4	4.5	OTHER
2018-03-26	3	3	2	2	3	1	0	2	1	1	1
2018-04-02	5	7	2	0	3	2	1	0	0	1	1
2018-04-09	2	3	1	2	0	2	1	1	0	1	2
2018-04-16	6	5	0	1	2	2	2	0	0	2	1
2018-04-23	2	0	0	0	0	2	1	2	2	0	1

4. Count the revenue for each item over time

This example uses the sample dataset from the Search Tutorial and a field lookup to add more information to the event data. Before you run this example:

- Download the data set from [this topic in the Search Tutorial](#) and follow the instructions to upload it to your Splunk deployment.
- Download the `Prices.csv.zip` file from [this topic in the Search Tutorial](#) and follow the instructions to set up your field lookup.

The original data set includes a `productId` field that is the catalog number for the items sold at the Buttercup Games online store. The field lookup adds two new fields to your events: `productName`, which is a descriptive name for the item, and `price`, which is the cost of the item.

Count the total revenue made for each item sold at the shop over the **last 7 days**. This example shows two different searches to generate the calculations.

Search 1

The first search uses the `span` argument to bucket the times of the search results into 1 day increments. The search then uses the `sum()` function to add the `price` for each `productName`.

```
sourcetype=access_* action=purchase | timechart span=1d sum(price) by productName usenull=f
```

Search 2

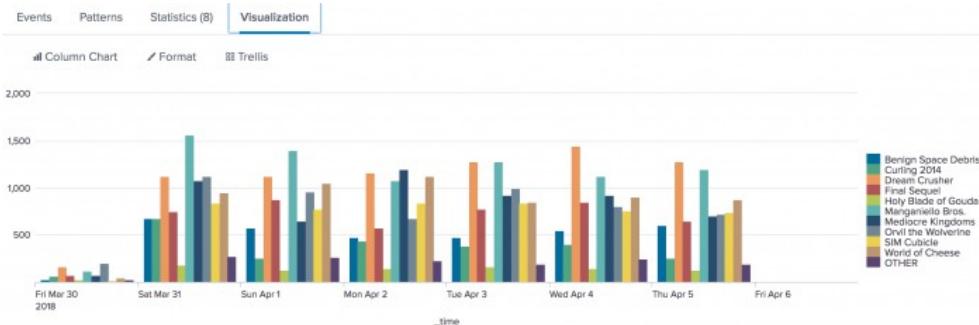
This second search uses the `per_day()` function to calculate the total of the `price` values for each day.

```
sourcetype=access_* action=purchase | timechart per_day(price) by productName usenull=f
```

Both searches produce similar results. Search 1 produces values with two decimal places. Search 2 produces values with six decimal places. The following image shows the results from Search 1.

_time	Benign ✓ Space Debris	Curling 2014	Dream Crusher	Final Sequel	Holy ✓ Blade of Gouda	Manganelli Bros.	Mediocre Kingdoms	Orville ✓ the Wolverine	SIM ✓ Cubicle	World ✓ of Cheese	OTHER
2018-03-30	24.99	59.97	159.96	74.97	23.96	199.97	74.97	199.95	19.99	49.98	22.95
2018-03-31	674.73	679.66	1119.72	749.70	179.70	1559.61	1074.57	1119.72	839.58	949.62	274.38
2018-04-01	574.77	259.87	1119.72	874.65	125.79	1399.65	649.74	959.76	779.61	1049.58	261.41
2018-04-02	474.81	439.78	1159.71	574.77	149.75	1079.73	1199.52	679.83	839.58	1124.55	227.49
2018-04-03	474.81	379.81	1279.68	774.69	161.73	1279.68	924.63	999.75	839.58	849.66	190.58
2018-04-04	549.78	399.80	1439.64	849.66	149.75	1119.72	924.63	799.80	759.62	899.64	242.45
2018-04-05	599.76	259.87	1279.68	649.74	131.78	1199.70	699.72	719.82	739.63	874.65	195.56

Click the **Visualization** tab. If necessary, change the chart to a column chart.



Now you can compare the total revenue made for items purchased each day and over the course of the week.

5. Chart product views and purchases for a single day

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.

Chart a single day's views and purchases at the Buttercup Games online store.

```
sourcetype=access_* | timechart per_hour(eval(method="GET")) AS Views, per_hour(eval(action="purchase")) AS Purchases
```

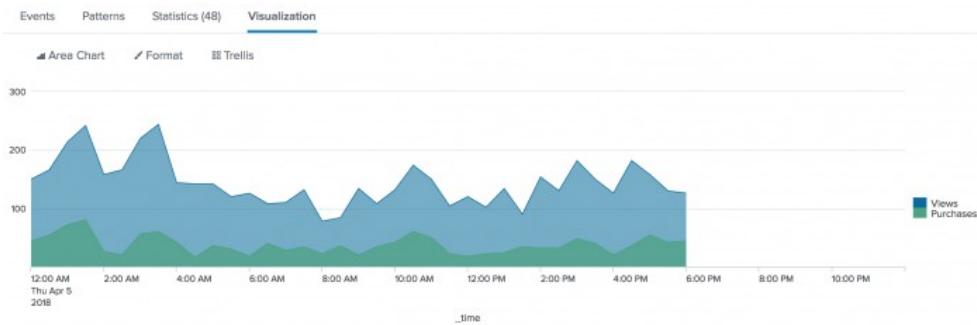
- This search uses the `per_hour()` function and `eval` expressions to search for page views (`method=GET`) and purchases (`action=purchase`).
- The results of the `eval` expressions are renamed as `Views` and `Purchases`, respectively.

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

_time	Views	Purchases
2018-04-05 00:00:00	150.000000	44.000000
2018-04-05 00:30:00	166.000000	54.000000

_time	Views	Purchases
2018-04-05 01:00:00	214.000000	72.000000
2018-04-05 01:30:00	242.000000	80.000000
2018-04-05 02:00:00	158.000000	26.000000
2018-04-05 02:30:00	166.000000	20.000000
2018-04-05 03:00:00	220.000000	56.000000

Click the **Visualization** tab. Format the results as an area chart.



The difference between the two areas indicates that many of the views did not become purchases. If all of the views became purchases, you would expect the areas to overlay on top each other completely. There would be no difference between the two areas.

Where clause examples

These examples use the `where` clause to control the number of series values returned in the time-series chart.

Example 1: Show the 5 most rare series based on the minimum count values. All other series values will be labeled as "other".

```
index=_internal | timechart span=1h count by source WHERE min in bottom5
```

Example 2: Show the 5 most frequent series based on the maximum values. All other series values will be labeled as "other".

```
index=_internal | timechart span=1h count by source WHERE max in top5
```

These two searches return six data series: the five top or bottom series specified and the series labeled `other`. To hide the "other" series, specify the argument `useother=f`.

Example 3: Show the source series count of INFO events, but only where the total number of events is larger than 100. All other series values will be labeled as "other".

```
index=_internal | timechart span=1h sum(eval(if(log_level=="INFO",1,0))) by source WHERE sum > 100
```

Example 4: Using the where clause with the count function measures the total number of events over the period. This yields results similar to using the sum function.

The following two searches returns the sources series with a total count of events greater than 100. All other series values will be labeled as "other".

```
index=_internal | timechart span=1h count by source WHERE count > 100
```

```
index=_internal | timechart span=1h count by source WHERE sum > 100
```

See also

Commands

- [bin](#)
- [chart](#)
- [sitimechart](#)
- [timewrap](#)

Blogs

- Search commands > stats, chart, and timechart

timewrap

Description

Displays, or wraps, the output of the `timechart` command so that every period of time is a different series.

You can use the `timewrap` command to compare data over specific time period, such as day-over-day or month-over-month. You can also use the `timewrap` command to compare multiple time periods, such as a two week period over another two week period. See [Timescale options](#).

Syntax

The required syntax is in **bold**.

```
timewrap
<timewrap-span>
[align=now | end]
[series=relative | exact | short]
[time_format=<str>]
```

Required arguments

`timewrap-span`

Syntax: [<int>]<timescale>

Description: A span of each bin, based on time. The `timescale` is required. The `int` is not required. If <int> is not specified, 1 is assumed. For example if `day` is specified for the timescale, `1day` is assumed. See [Timescale options](#).

Optional arguments

align

Syntax: align=now | end

Description: Specifies if the wrapping should be aligned to the current time or the end time of the search.

Default: end

series

Syntax: series=relative | exact | short

Description: Specifies how the data series is named. If `series=relative` and `timewrap-span` is set to week, the field names are `latest_week`, `1week_before`, `2weeks_before`, and so forth. If `series=exact`, use the `time_format` argument to specify a custom format for the series names. If `series=short`, the field names are an abbreviated version of the field names used with `series=relative`. With `series=short`, the field names are abbreviated to "s" followed by a number representing the period of time. For example, if `timewrap-span` is set to week, the field names are `s0`, `s1`, `s2` and so forth. The field `s0` represents the latest week. The field `s1` represents 1 week before the latest week.

Default: relative

time_format

Syntax: time_format=<str>

Description: Use with `series=exact` to specify a custom name for the series. The `time_format` is designed to be used with the [time format variables](#). For example, if you specify `time_format="week of %d/%m/%y"`, this format appears as `week of 13/2/17` and `week of 20/2/17`. If you specify `time_format=week of %b %d`, this format appears as `week of Feb 13` and `week of Feb 20`. See the [Usage](#) section.

Default: None

Timescale options

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <week> | <month> | <quarter> | <year>

Description: Time scale units.

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<week>	w week weeks	Time scale in weeks.
<month>	m mon month months	Time scale in months.
<quarter>	qtr quarter quarters	Time scale in quarters
<year>	y yr year years	Time scale in years.

The `timewrap` command uses the abbreviation `m` to refer to months. Other commands , such as `timechart` and `bin` use the abbreviation `m` to refer to minutes.

Usage

The `timewrap` command is a reporting command.

You must use the `timechart` command in the search before you use the `timewrap` command.

The wrapping is based on the end time of the search. If you specify the time range of `All time`, the wrapping is based on today's date. You see this in the timestamps for the `_time` field and in the data series names.

Field names with a timechart BY clause

If you use a BY clause in the `timechart` command part of your search, the field names generated by the `timewrap` command are appended to the field names generated with the BY clause. For example, suppose you have a search that includes `BY categoryId` in the `timechart` command and the results look something like this:

<code>_time</code>	<code>ACCESSORIES</code>	<code>SPORTS</code>	<code>STRATEGY</code>
2020-05-21	5	17	32
2020-05-22	62	22	127
2020-05-23	65	34	128
2020-05-24	5	17	32
2020-05-25	62	22	127
2020-05-26	65	34	128

When you add the `timewrap` command, such as `| timewrap w series=short`, the series field names are appended to the category ID names from the `timechart` BY clause.

The output looks something like this:

<code>_time</code>	<code>ACCESSORIES_s1</code>	<code>SPORTS_s1</code>	<code>STRATEGY_s1</code>	<code>ACCESSORIES_s0</code>	<code>SPORTS_s0</code>	<code>STRATEGY_s0</code>
2020-05-21				5	17	32
2020-05-22				62	22	127
2020-05-23				65	34	128
2020-05-24				5	17	32
2020-05-25	62	22	127	17	54	39
2020-05-26	65	34	128			

Using the time_format argument

If you do not include any time specifiers with the `time_format` argument, all of the data series display the same name and are compressed into each other.

Examples

1. Compare week over week

Display a timechart that has a span of 1 day for each count in a week over week comparison table. Each table column, which is the series, is 1 week of time.

```
... | timechart count span=1d | timewrap 1week
```

2. Compare today, yesterday, and average for the week

To compare a few days with the weekly average, you need to calculate the daily totals, calculate the weekly average, and remove the days you don't want to use. For example:

```
... | timechart count span=1h | timewrap d series=short | addtotals s* | eval 7dayavg=Total/7.0 | table _time, _span, s0, s1, 7dayavg | rename s0 as now, s1 as yesterday
```

- Use the `timewrap` command to generate results over the last 7 days.
- By using the `series=short` argument, field names are generated in the output which start with "s", making it easy to create totals using the `addtotals` command.
- Use the `addtotals` and `eval` commands to calculate the average over those 7 days.
- The `table` command is used to cut out days 3-7 so that only today, yesterday, and the weekly average are returned.
- The `rename` command is used to rename the fields.

The output looks something like this:

_time	now	yesterday	7dayavg
2020-02-20 15:00	0	0	0.0
2020-02-20 16:00	0	0	0.29
2020-02-20 17:00	0	0	0.0
2020-02-20 18:00	0	0	0.0
2020-02-20 19:00	0	0	0.57
2020-02-20 20:00	0	0	0.0
2020-02-20 21:00	0	0	0.29
2020-02-20 22:00	0	0	1.1

3. Compare a day of the week to the same day of the previous weeks

You can compare a day of the week to the same day of the weeks by specifying a filter at the end of the search. For example, to compare Wednesdays your search would be like this:

```
... | timechart count span=1h | timewrap w | where strftime(_time, "%A") == "Wednesday"
```

The output looks something like this:

_time	4weeks_before	3weeks_before	2weeks_before	1week_before	latest_week
2020-02-19 00:00	0	1	4	0	1

<code>_time</code>	<code>4weeks_before</code>	<code>3weeks_before</code>	<code>2weeks_before</code>	<code>1week_before</code>	<code>latest_week</code>
2020-02-19 01:00	2	0	0	0	1
2020-02-19 02:00	3	5	7	2	0
2020-02-19 03:00	6	4	0	1	2
2020-02-19 04:00	9	0	4	0	0
2020-02-19 05:00	2	8	7	3	1
2020-02-19 06:00	4	2	7	0	1
2020-02-19 07:00	6	9	2	2	0

If you change the timechart span to 1d instead of 1h, your output will look like this:

<code>_time</code>	<code>4weeks_before</code>	<code>3weeks_before</code>	<code>2weeks_before</code>	<code>1week_before</code>	<code>latest_week</code>
2020-02-19	32	29	31	8	6

See also

[timechart](#)

tojson

Description

Converts events into JSON objects. You can specify which fields get converted by identifying them through exact match or through wildcard expressions. You can also apply specific JSON datatypes to field values using datatype functions. The `tojson` command converts multivalue fields into JSON arrays.

When fields are specifically named in a `tojson` search, the command generates JSON objects that are limited to the values of just those named fields. If no fields are specified for `tojson`, `tojson` generates JSON objects for all fields that would otherwise be returned by the search.

Syntax

Required syntax is in **bold**.

```
| tojson
[<tojson-function>]...
[default_type=<datatype>]
[fill_null=<boolean>]
[include_internal=<boolean>]
[output_field=<string>]
```

Optional arguments

`tojson-function`

Syntax: [auto | bool | json | none | num | str](<wc-field>)...

Description: Applies JSON datatype functions to values of named fields. See [Usage](#) for details about how `tojson` interprets these datatype functions, and how `tojson` applies datatypes to field values when it converts events into JSON objects.

If you provide no fields, the `tojson` processor creates JSON objects for each event that include all available fields. In other words, it applies `none(*)` to the search.

Default: `none(*)`

`default_type`

Syntax: `default_type=<datatype>`

Description: Specifies the datatype that the `tojson` processor should apply to fields that aren't specifically associated with a datatype function.

Default: `none`

`fill_null`

Syntax: `fill_null=<boolean>`

Description: When set to true, `tojson` outputs a literal `null` value when `tojson` skips a value. For example, normally, when `tojson` tries to apply the `json` datatype to a field that does not have proper JSON formatting, `tojson` skips the field. However, if `fill_null=true`, the `tojson` processor outputs a `null` value

Default: `false`

`include_internal`

Syntax: `include_internal=<boolean>`

Description: When set to true, `tojson` includes **internal fields** such as `_time`, `_index`, or `_raw` in its JSON object output.

Default: `false`

`output_field`

Syntax: `output_field=<string>`

Description: Specifies the name of the field to which the `tojson` search processor writes the output JSON objects.

Default: `_raw`

Usage

The `tojson` command is a **streaming command**, which means it operates on each event as it is returned by the search. See [Types of commands](#).

Apply JSON datatypes to field values

The `tojson` command applies JSON datatypes to field values according to logic encoded in its datatype functions.

You can assign specific datatype functions to fields when you write a `tojson` search. Alternatively, you can name a set of fields without associating them with datatype functions, and then identify a `default_type` that `tojson` can apply to those unaffiliated fields.

If you do not specify any fields for the `tojson` command, the `tojson` returns JSON objects for each field that can possibly be returned by the search at that point, and applies the `none` datatype function to the values of those fields. The `none` datatype function applies the numeric datatype to field values that are purely numeric, and applies the string datatype to all other field values.

The following table explains the logic that the various datatype functions use to apply datatypes to the values of the fields

with which they are associated.

Datatype function	Conversion logic
auto	<p>Converts all values of the specified field into JSON-formatted output. Automatically determines the field datatypes.</p> <ul style="list-style-type: none"> If the value is numeric, the JSON output has a numeric output and includes a literal numeric. If the value is the string <code>true</code> or <code>false</code> the JSON output has a Boolean type. If the value is a literal <code>null</code> the JSON output has a null type and includes a <code>null</code> value. If the value is a string other than the previously mentioned strings, <code>tojson</code> examines the string. If it is proper JSON, <code>tojson</code> outputs a nested JSON object. If it is not proper JSON, <code>tojson</code> includes the string in the output.
bool	<p>Converts valid values of the specified field to the Boolean datatype, and skips invalid values, using string validation.</p> <ul style="list-style-type: none"> If the value is a number, <code>tojson</code> outputs <code>false</code> only if that value is 0. Otherwise <code>tojson</code> outputs <code>false</code>. If the value is a string, <code>tojson</code> outputs <code>false</code> only if the value is <code>false</code>, <code>f</code>, or <code>no</code>. The <code>tojson</code> processor outputs <code>true</code> only if the value is code <code>true</code>, <code>t</code>, or <code>yes</code>. If the value does not fit into those two sets of strings, it is skipped. The validation for the <code>bool</code> datatype function is case insensitive. This means that it also interprets <code>FALSE</code>, <code>False</code>, <code>F</code>, and <code>NO</code> as <code>false</code>.
json	<p>Converts all values of the specified field to the JSON type, using string validation. Skips values with invalid JSON.</p> <ul style="list-style-type: none"> If the value is a number, <code>tojson</code> outputs that number. If the value is a string, <code>tojson</code> outputs the string as a JSON block. If the value is invalid JSON, <code>tojson</code> skips it.
none	<p>Outputs all values for the specified field in the JSON type. Does not apply string validation.</p> <ul style="list-style-type: none"> If the value is a number, <code>tojson</code> outputs a numeric datatype in the JSON block. If the value is a string, <code>tojson</code> outputs a string datatype.
num	<p>Converts all values of the specified field to the numeric type, using string validation.</p> <ul style="list-style-type: none"> If the value is a number, <code>tojson</code> outputs that value and gives it the numeric datatype. If the value is a string, <code>tojson</code> attempts to parse the string as a number. If it cannot, it skips the value.
str	<p>Converts all values of the specified field into the string datatype, using string validation. The <code>tojson</code> processor applies the string type to all values of the specified field, even if they are numbers, Boolean values, and so on.</p>

When a field includes multivalues, `tojson` outputs a JSON array and applies the datatype function logic to each element of the array.

Examples

1. Convert all events returned by a search into JSON objects

This search of `index=_internal` converts all events it returns for its time range into JSON-formatted data. Because the search string does not assign datatype functions to specific fields, by default `tojson` applies the `none` datatype function to all fields returned by the search. This means all of their values get either the numeric or string datatypes.

```
index=_internal | tojson
```

For example, say you start with events that look like this:

```
12-18-2020 18:19:25.601 +0000 INFO Metrics - group=thruput, name=thruput, instantaneous_kbps=5.821,
```

```
instantaneous_eps=27.194, average_kbps=5.652, total_k_processed=444500.000, kb=180.443, ev=843,
load_average=19.780
```

After being processed by `tojson`, such events have JSON formatting like this:

```
{
  [-]
    component: Metrics
    date_hour: 18
    date_mday: 18
    date_minute: 22
    date_month: december
    date_second: 9
    date_wday: friday
    date_year: 2020
    date_zone: 0
    event_message: group=thruput, name=thruput, instantaneous_kbps=2.914, instantaneous_eps=13.903,
average_kbps=5.062, total_k_processed=398412.000, kb=90.338, ev=431, load_average=14.690
    group: thruput
    host: sh1
    index: _internal
    linecount: 1
    log_level: INFO
    name: thruput
    punct: --_:::_+__-_=_,_=_.,_=_.,_=_.,_=_.,_=_.
    source: /opt/splunk/var/log/splunk/metrics.log
    sourcetype: splunkd
    splunk_server: idx2
    timeendpos: 29
    timestamppos: 0
}
```

2. Specify different datatypes for 'date' fields

The following search of the `_internal` index converts results into JSON objects that have only the `date_*` fields from each event. The numeric datatype is applied to all `date_hour` field values. The string datatype is applied to all other date field values.

```
index=_internal | tojson num(date_hour) str(date_*)
```

This search produces JSON objects like this:

```
{
  [-]
    date_hour: 18
    date_mday: 18
    date_minute: 28
    date_month: december
    date_second: 45
    date_wday: friday
    date_year: 2020
    date_zone: 0
}
```

Note that all fields that do not start with `date_` have been stripped from the output.

3. Limit JSON object output and apply datatypes to the field values

This search returns JSON objects only for the `name`, `age`, and `isRegistered` fields. It uses the `auto` datatype function to have `tojson` automatically apply appropriate JSON datatypes to the values of those fields.

```
... | tojson auto(name) auto(age) auto(isRegistered)
```

4. Convert all events into JSON objects and apply appropriate datatypes to all field values

This search converts all of the fields in each event returned by the search into JSON objects. It uses the `auto` datatype function in conjunction with a wildcard to apply appropriate datatypes to the values of all fields returned by the search.

```
... | tojson auto(*)
```

Notice that this search references the `auto` datatype function, which ensures that Boolean, JSON, and null field values are appropriately typed alongside numeric and string values.

Alternatively, you can use `default_type` to apply the `auto` datatype function to all fields returned by a search:

```
... | tojson default_type=auto
```

5. Apply the Boolean datatype to a specific field

This example generates JSON objects containing values of the `isInternal` field. It uses the `bool` datatype function to apply the Boolean datatype to those field values.

```
... | tojson bool(isInternal)
```

6. Include internal fields and assign a 'null' value to skipped fields

This example demonstrates usage of the `include_internal` and `fill_null` arguments.

```
... | tojson include_internal=true fill_null=true
```

7. Designate a default datatype for a set of fields and write the JSON objects to another field

This search generates JSON objects based on the values of four fields. It uses the `default_type` argument to convert the first three fields to the `num` datatype. It applies the string datatype to a fourth field. Finally, it writes the finished JSON objects to the field `my_JSON_field`.

```
... | tojson age height weight str(name) default_type=num output_field=my_JSON_field
```

See also

Commands

[fromjson](#)

Evaluation functions

[JSON functions](#)

top

Description

Finds the most common values for the fields in the field list. Calculates a count and a percentage of the frequency the values occur in the events. If the <by-clause> is included, the results are grouped by the field you specify in the <by-clause>.

Syntax

```
top [<N>] [<top-options>...] <field-list> [<by-clause>]
```

Required arguments

<field-list>

Syntax: <field>, <field>, ...

Description: Comma-delimited list of field names.

Optional arguments

<N>

Syntax: <int>

Description: The number of results to return.

Default: 10

<top-options>

Syntax: countfield=<string> | limit=<int> | otherstr=<string> | percentfield=<string> | showcount=<bool> | showperc=<bool> | useother=<bool>

Description: Options for the `top` command. See [Top options](#).

<by-clause>

Syntax: BY <field-list>

Description: The name of one or more fields to group by.

Top options

countfield

Syntax: countfield=<string>

Description: For each value returned by the `top` command, the results also return a count of the events that have that value. This argument specifies the name of the field that contains the count. The count is returned by default. If you do not want to return the count of events, specify `showcount=false`.

Default: count

limit

Syntax: limit=<int>

Description: Specifies how many results to return. To return all values, specify zero (0). Specifying `top limit=<int>` is the same as specifying `top N`.

Default: 10

otherstr

Syntax: otherstr=<string>

Description: If `useother=true`, a row representing all other values is added to the results. Use `otherstr=<string>` to specify the name of the label for the row.

Default: OTHER

percentfield

Syntax: `percentfield=<string>`

Description: For each value returned by the `top` command, the results also return a percentage of the events that have that value. This argument specifies the name of the field that contains the percentage. The percentage is returned by default. If you do not want to return the percentage of events, specify `showperc=false`.

Default: percent

showcount

Syntax: `showcount=<bool>`

Description: Specify whether to create a field called "count" (see "countfield" option) with the count of that tuple.

Default: true

showperc

Syntax: `showperc=<bool>`

Description: Specify whether to create a field called "percent" (see "percentfield" option) with the relative prevalence of that tuple.

Default: true

useother

Syntax: `useother=<bool>`

Description: Specify whether or not to add a row that represents all values not included due to the limit cutoff.

Default: false

Usage

The `top` command is a **transforming command**. See [Command types](#).

Default fields

When you use the `top` command, two fields are added to the results: `count` and `percent`.

Field	Description
<code>count</code>	The number of events in your search results that contain the field values that are returned by the <code>top</code> command. See the <code>countfield</code> and <code>showcount</code> arguments.
<code>percent</code>	The percentage of events in your search results that contain the field values that are returned by the <code>top</code> command. See the <code>percentfield</code> and <code>showperc</code> arguments.

Default maximum number of results

By default the `top` command returns a maximum of 50,000 results. This maximum is controlled by the `maxresultrows` setting in the `[top]` stanza in the `limits.conf` file. Increasing this limit can result in more memory usage.

Only users with file system access, such as system administrators, can edit the configuration files. Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make the changes in the local directory.

See How to edit a configuration file.

If you have Splunk Cloud Platform, you need to file a Support ticket to change this limit.

Lexicographic order of results

In searches that use the `limit` option with multiple sets of field lists, only the last lexicographical value of the `<field-list>` is returned in the search results. For example, in the following search, `Orlando` is the only `location` field that is returned because it's the last value when sorted lexicographically.

```
| makeresults | eval location="Orlando Dallas Atlanta" | makemv location | mvexpand location | eval user="Alex Kai Morgan" | makemv user | mvexpand user | top limit=1 location by user
```

The search results look something like this.

user	location	count	percent
Alex	Orlando	1	33.333333
Kai	Orlando	1	33.333333
Morgan	Orlando	1	33.333333

Examples

Example 1: Return the 20 most common values for a field

This search returns the 20 most common values of the "referer" field. The results show the number of events (count) that have that a count of referer, and the percent that each referer is of the total number of events.

```
sourcetype=access_* | top limit=20 referer
```

20 Per Page ▾ Format ▾ Preview ▾

referer	count	percent
http://www.buttercupgames.com/category.screen?categoryId=STRATEGY	2284	5.777598
http://www.buttercupgames.com	1980	5.008601
http://www.google.com	1582	4.001821
http://www.buttercupgames.com/category.screen?categoryId=ARCADE	1372	3.470606
http://www.buttercupgames.com/category.screen?categoryId=NULL	1281	3.240413
http://www.buttercupgames.com/product.screen?productId=SFBVS-G01	1210	3.060811
http://www.buttercupgames.com/category.screen?categoryId=ACCESSORIES	1082	2.737023
http://www.buttercupgames.com/category.screen?categoryId=TEE	993	2.511889
http://www.yahoo.com	766	1.937671

Example 2: Return top values for one field organized by another field

This search returns the top "action" values for each "referer_domain".

```
sourcetype=access_* | top action by referer_domain
```

Because a limit is not specified, this returns all the combinations of values for "action" and "referer_domain" as well as the counts and percentages:

20 Per Page ▾

Format ▾

Preview ▾

referer_domain	action	count	percent
http://www.bing.com	view	46	51.11111
http://www.bing.com	addtocart	21	23.33333
http://www.bing.com	remove	12	13.33333
http://www.bing.com	changequantity	11	12.22222
http://www.buttercupgames.com	purchase	5737	30.120229
http://www.buttercupgames.com	addtocart	5572	29.253951
http://www.buttercupgames.com	view	5054	26.534362
http://www.buttercupgames.com	remove	1359	7.134982
http://www.buttercupgames.com	changequantity	1325	6.956476
http://www.google.com	view	201	50.375940
http://www.google.com	addtocart	96	24.060150

Example 3: Returns the top product purchased for each category

This example uses the sample dataset from the Search Tutorial and a field lookup to add more information to the event data.

- Download the data set from [Add data tutorial](#) and follow the instructions to load the tutorial data.
- Download the CSV file from [Use field lookups tutorial](#) and follow the instructions to set up the lookup definition to add price and productName to the events.

After you configure the field lookup, you can run this search using the time range, **All time**.

This search returns the top product purchased for each category. Do not show the percent field. Rename the count field to "total".

```
sourcetype=access_* status=200 action=purchase | top 1 productName by categoryId showperc=f countfield=total
```

20 Per Page ▾ Format ▾ Preview ▾

categoryId	productName	total
ACCESSORIES	Fire Resistance Suit of Provolone	187
ARCADE	Manganelli Bros.	209
SHOOTER	World of Cheese	245
SIMULATION	SIM Cubicle	246
SPORTS	Curling 2014	138
STRATEGY	Mediocre Kingdoms	238

See also

[rare](#), [sitop](#), [stats](#)

transaction

Description

The transaction command finds transactions based on events that meet various constraints. Transactions are made up of the raw text (the `_raw` field) of each member, the time and date fields of the earliest member, as well as the union of all other fields of each member.

Additionally, the `transaction` command adds two fields to the raw events, `duration` and `eventcount`. The values in the `duration` field show the difference between the timestamps for the first and last events in the transaction. The values in the `eventcount` field show the number of events in the transaction.

See About transactions in the *Search Manual*.

Syntax

The required syntax is in **bold**.

```
transaction
[<field-list>]
[name=<transaction-name>]
[<txn_definition-options>...]
[<memcontrol-options>...]
[<rendering-options>...]
```

Required arguments

None.

Optional arguments

field-list

Syntax: <field> ...

Description: One or more field names. The events are grouped into transactions, based on the unique values in the fields. For example, suppose two fields are specified: `client_ip` and `host`. For each `client_ip` value, a separate transaction is returned for each unique `host` value for that `client_ip`.

memcontrol-options

Syntax: <maxopentxn> | <maxopenevents> | <keepevicted>

Description: These options control the memory usage for your transactions. They are not required, but you can use 0 or more of the options to define your transaction. See [Memory control options](#).

name

Syntax: name=<transaction-name>

Description: Specify the stanza name of a transaction that is configured in the `transactiontypes.conf` file. This runs the search using the settings defined in this stanza of the configuration file. If you provide other transaction definition options (such as `maxpause`) in this search, they overrule the settings in the configuration file.

rendering-options

Syntax: <delim> | <mvlist> | <mvraw> | <>nullstr>

Description: These options control the multivalue rendering for your transactions. They are not required, but you can use 0 or more of the options to define your transaction. See [Multivalue rendering options](#).

txn_definition-options

Syntax: <maxspan> | <maxpause> | <maxevents> | <startswith> | <endswith> | <connected> | <unifyends> | <keeporphans>

Description: Specify the transaction definition options to define your transactions. You can use multiple options to define your transaction.

Txn definition options

connected

Syntax: connected=<bool>

Description: Only relevant if a field or fields list is specified. If an event contains fields required by the transaction, but none of these fields have been instantiated in the transaction (added with a previous event), this opens a new transaction (connected=true) or adds the event to the transaction (connected=false).

Default: true

endswith

Syntax: endswith=<filter-string>

Description: A search or eval expression which, if satisfied by an event, marks the end of a transaction.

keeporphans

Syntax: keeporphans=true | false

Description: Specify whether the transaction command should output the results that are not part of any transactions. The results that are passed through as "orphans" are distinguished from transaction events with a _txn_orphan field, which has a value of 1 for orphan results.

Default: false

maxspan

Syntax: maxspan=<int>[s | m | h | d]

Description: Specifies the maximum length of time in seconds, minutes, hours, or days for the pause between the events in a transaction. If value is negative, the maxspan constraint is disabled and there is no limit. This setting is accurately documented even though its name is inconsistent with its description.

Default: -1 (no limit)

maxpause

Syntax: maxpause=<int>[s | m | h | d]

Description: Specifies the maximum length of time in seconds, minutes, hours, or days that the events can span. The events in the transaction must span less than the integer specified for maxpause. Events that exceed the maxpause limit are treated as part of a separate transaction. If the value is negative, the maxpause constraint is disabled and there is no limit. This setting is accurately documented even though its name is inconsistent with its description.

Default: -1 (no limit)

maxevents

Syntax: maxevents=<int>

Description: The maximum number of events in a transaction. This constraint is disabled if the value is negative.

Default: 1000

startswith

Syntax: startswith=<filter-string>

Description: A search or eval filtering expression which if satisfied by an event marks the beginning of a new transaction.

unifyends

Syntax: unifyends= true | false

Description: Whether to force events that match startswith/endswith constraint(s) to also match at least one of the fields used to unify events into a transaction.

Default: false

Filter string options

These options are used with the `startswith` and `endswith` arguments.

<filter-string>

Syntax: <search-expression> | (<quoted-search-expression>) | eval(<eval-expression>)

Description: A search or eval filtering expression which if satisfied by an event marks the end of a transaction.

<search-expression>

Description: A valid search expression that does not contain quotes.

<quoted-search-expression>

Description: A valid search expression that contains quotes.

<eval-expression>

Description: A valid eval expression that evaluates to a Boolean.

Memory control options

If you have Splunk Cloud, Splunk Support administers the settings in the `limits.conf` file on your behalf.

keepevicted

Syntax: keepevicted=<bool>

Description: Whether to output evicted transactions. Evicted transactions can be distinguished from non-evicted transactions by checking the value of the 'closed_txn' field. The 'closed_txn' field is set to '0', or false, for evicted transactions and '1', or true for non-evicted, or closed, transactions. The 'closed_txn' field is set to '1' if one of the following conditions is met: maxevents, maxspan, maxpause, startswith. For `startswith`, because the `transaction` command sees events in reverse time order, it closes a transaction when it satisfies the start condition. If none of these conditions is specified, all transactions are output even though all transactions will have 'closed_txn' set to '0'. A transaction can also be evicted when the memory limitations are reached.

Default: false or 0

maxopenevents

Syntax: maxopenevents=<int>

Description: Specifies the maximum number of events (which are) part of open transactions before transaction eviction starts happening, using LRU policy.

Default: The default value for this argument is read from the transactions stanza in the `limits.conf` file.

maxopentxn

Syntax: maxopentxn=<int>

Description: Specifies the maximum number of not yet closed transactions to keep in the open pool before starting to evict transactions, using LRU policy.

Default: The default value for this argument is read from the transactions stanza in the `limits.conf` file.

Multivalue rendering options

delim

Syntax: delim=<string>

Description: Specify a character to separate multiple values. When used in conjunction with the `mvraw=t` argument, represents a string used to delimit the values in the `_raw` field.

Default: " " (whitespace)

mvlist

Syntax: mvlist= true | false | <field-list>

Description: Flag that controls how multivalued fields are processed. When set to `mvlist=true`, the multivalued fields in the transaction are a list of the original events ordered in arrival order. When set to `mvlist=false`, the

multivalued fields in the transaction are a set of unique field values ordered alphabetically. If a comma or space delimited list of fields is provided, only those fields are rendered as lists.

Default: false

`mvraw`

Syntax: `mvraw=<bool>`

Description: Used to specify whether the `_raw` field of the transaction search result should be a multivalued field.

Default: false

`nullstr`

Syntax: `nullstr=<string>`

Description: A string value to use when rendering missing field values as part of multivalued fields in a transaction. This option applies only to fields that are rendered as lists.

Default: `NULL`

Usage

The `transaction` command is a centralized streaming command. See [Command types](#).

In the output, the events in a transaction are grouped together as multiple values in the `Events` field. Each event in a transaction starts on a new line by default.

If there are more than 5 events in a transaction, the remaining events in the transaction are collapsed. A message appears at the end of the transaction which gives you the option to show all of the events in the transaction.

Specifying multiple fields

The Splunk software does not necessarily interpret the transaction defined by multiple fields as a conjunction (`field1 AND field2 AND field3`) or a disjunction (`field1 OR field2 OR field3`) of those fields. If there is a transitive relationship between the fields in the fields list and if the related events appear in the correct sequence, each with a different timestamp, `transaction` command will try to use it. For example, if you searched for

```
... | transaction host cookie
```

You might see the following events grouped into a transaction:

```
event=1 host=a
event=2 host=a cookie=b
event=3 cookie=b
```

Descending time order required

The `transaction` command requires that the incoming events be in descending time order. Some commands, such as `eval`, might change the order or time labeling of events. If one of these commands precedes the `transaction` command, your search returns an error unless you include a `sort` command in your search. The `sort` command must occur immediately before the `transaction` command to reorder the search results in descending time order.

Basic Examples

1. Transactions with the same host, time range, and pause

Group search results that have the same host and cookie value, occur within 30 seconds, and do not have a pause of more than 5 seconds between the events.

```
... | transaction host cookie maxspan=5s maxpause=30s
```

2. Transactions with the same "from" value, time range, and pause

Group search results that have the same value of "from", with a maximum span of 30 seconds, and a pause between events no greater than 5 seconds into a transaction.

```
... | transaction from maxspan=5s maxpause=30s
```

3. Transactions with the same field values

You have events that include an alert_level. You want to create transactions where the level is equal. Using the streamstats command, you can remember the value of the alert level for the current and previous event. Using the transaction command, you can create a new transaction if the alert level is different. Output specific fields to table.

```
... | streamstats window=2 current=t latest(alert_level) AS last earliest(alert_level) AS first | transaction endswith=eval(first!=last) | table _time duration first last alert_level eventcount
```

Extended Examples

1. Transactions of Web access events based on IP address

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.

Define a transaction based on Web access events that share the same IP address. The first and last events in the transaction should be no more than thirty seconds apart and each event should not be longer than five seconds apart.

```
sourcetype=access_* | transaction clientip maxspan=5s maxpause=30s
```

This produces the following events list. The clientip for each event in the transaction is highlighted.

List		Format	20 Per Page	< Prev	1	2	3	4	5	6	7	8	...	Next >
Hide Fields	All Fields	1	Time	Event										
SELECTED FIELDS														
<i>a host</i>	3													
<i>a source</i>	3													
<i>a sourcetype</i>	1													
INTERESTING FIELDS														
<i>a action</i>	5													
<i>a bytes</i>	100+													
<i>a categoryid</i>	8													
<i>a clientip</i>	100+													
<i>#closed_bnn</i>	2													
<i>#date_hour</i>	24													
<i>#date_mday</i>	6													
<i>#date_minute</i>	60													
<i>#date_month</i>	1													
<i>#date_second</i>	60													
<i>#date_wday</i>	6													
<i>#date_year</i>	1													
<i>#date_zone</i>	1													
<i>#duration</i>	17													
<i>#eventcount</i>	24													
<i>#field_match_sum</i>	24													
<i>#file</i>	14													
<i>#ident</i>	1													
<i>#index</i>	1													
<i>#itemid</i>	14													
<i>#JSESSIONID</i>	100+													
<i>#linecount</i>	24													
<i>#method</i>	2													

This search groups events together based on the IP addresses accessing the server and the time constraints. The search results might have multiple values for some fields, such as *host* and *source*. For example, requests from a single IP could come from multiple hosts if multiple people are shopping from the same office. For more information, read the topic *About transactions* in the *Knowledge Manager Manual*.

2. Transaction of Web access events based on host and client IP

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.

Define a transaction based on Web access events that have a unique combination of *host* and *clientip* values. The first and last events in the transaction should be no more than thirty seconds apart and each event should not be longer than five seconds apart.

```
sourcetype=access_* | transaction clientip host maxspan=5s maxpause=30s
```

This search produces the following events list.

List	Format	50 Per Page	< Prev	1	2	3	4	5	6	7	8
i	Time	Event									
>	3/13/18 5:20:44:000 PM	2.229.4.58 - - [13/ Mar/ 2018: 17:28:44] "POST /oldlink?itemID=EST-1&JSESSIONID=SD4SL4FF4ADFF52851 HTTP/1.1" 200 1453 "http://www.bing.com" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 692									
		2.229.4.58 - - [13/ Mar/ 2018: 17:28:45] "GET /oldlink?itemID=EST-1&JSESSIONID=SD4SL4FF4ADFF52851 HTTP/1.1" 200 2452 "http://www.buttercupgames.com/view&itemID=EST-1&productID=WC-SH-T02" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 692									
		2.229.4.58 - - [13/ Mar/ 2018: 17:28:46] "GET /product.screen?productId=WC-SH-A02&JSESSIONID=SD4SL4FF4ADFF52851 HTTP/1.1" 200 1647 "http://www.butchart.do?action=view&itemID=EST-1&productID=WC-SH-A02" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 385									
		Show all 13 lines									
		host = www1 source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie									
>	3/13/18 2:55:52:000 PM	2.229.4.58 - - [13/ Mar/ 2018: 14:55:52] "GET /cart.do?action=view&itemID=EST-7&productId=MB-AG-G07&JSESSIONID=SD0SL3FF4ADFF52015 HTTP/1.1" 200 361 "http://www.yahoo.com" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 787									
		2.229.4.58 - - [13/ Mar/ 2018: 14:55:53] "POST /cart.do?action=changeQuantity&itemID=EST-7&productId=WC-SH-G04&JSESSIONID=SD0SL3FF4ADFF52015 HTTP/1.1" 200 1512 "http://www.buttercupgames.com/productId=MB-AG-T01" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 385									
		Show all 6 lines									
		host = www3 source = tutorialdata.zip://www3/access.log sourcetype = access_combined_wcookie									

Each of these events have a distinct combination of the IP address (`clientip`) values and `host` values within the limits of the time constraints specified in the search.

3. Purchase transactions based on IP address and time range

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.

This search defines a purchase transaction as 3 events from one IP address which occur in a 10 minute span of time.

```
sourcetype=access_* action=purchase | transaction clientip maxpause=10m maxevents=3
```

This search defines a purchase event based on Web access events that have the `action=purchase` value. These results are then piped into the `transaction` command. This search identifies purchase transactions by events that share the same `clientip`, where each session lasts no longer than 10 minutes, and includes no more than 3 events.

This search produces the following events list:

List	Format	50 Per Page	< Prev	1	2	3	4	5	6	7	Next >
< Hide Fields	All Fields	Time	Event								
SELECTED FIELDS		> 3/13/18 6:20:54:000 PM	182.236.164.11 - - [13/ Mar/ 2018: 18:20:54] "POST /cart.do?action=purchase&itemID=EST-6&JSESSIONID=SD0SL8F10ADFF53101 HTTP/1.1" 200 1803 "http://www.buttercupgames.com/cart.do?action=addtocart&itemID=EST-6&categoryID=ARCADE&productId=MB-AG-G07" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 524								
INTERESTING FIELDS			182.236.164.11 - - [13/ Mar/ 2018: 18:20:54] "POST /cart.success.do?JSESSIONID=SD0SL8BFF10ADFF53101 HTTP/1.1" 200 356 "http://www.buttercupgames.com/cart.do?action=purchase&itemID=EST-6" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 220								
a host			host = www1 source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie								
a source											
a sourcetype											
Transaction with 2 events											
>	3/13/18 6:18:57:000 PM	198.35.1.75 - - [13/ Mar/ 2018: 18:18:57] "POST /cart.success.do?JSESSIONID=SD0SL2FF4ADFF53099 HTTP/1.1" 200 613 "http://www.buttercupgames.com/cart.do?action=purchase&itemID=EST-27" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 328									
		198.35.1.75 - - [13/ Mar/ 2018: 18:18:58] "POST /cart.do?action=purchase&itemID=EST-16&JSESSIONID=SD0SL2FF4ADFF53099 HTTP/1.1" 200 821 "http://www.buttercupgames.com/cart.do?action=addtocart&itemID=EST-16&categoryID=SIMULATION&productId=SC-MG-G10" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 328									
		host = www1 source = tutorialdata.zip://www1/access.log sourcetype = access_combined_wcookie									
Transaction with 3 events											

4. Email transactions based on maxevents and endswith

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 defines an email transaction as a group of up to 10 events. Each event contains the same value for the `mid` (message ID), `icid` (incoming connection ID), and `dcid` (delivery connection ID). The last event in the transaction contains a **Message done** string.

```
sourcetype="cisco:esa" | transaction mid dcid icid maxevents=10 endswith="Message done"
```

This search produces the following list of events:

	Time	Event
>	3/19/18 8:15:13.000 PM	Mon Mar 19 20:15:13 2018 Info: MID 19990744 ready 15284 bytes from <notify@example.com> Mon Mar 19 20:15:28 2018 Info: MID 19990744 Message-ID: <f4510803fe2007c65672cddee71dbe20m3.datingvip.coop> Mon Mar 19 20:16:33 2018 Info: MID 19990744 Subject:=?UTF-8?B?4pmIJF1vdS8IYXZlIGEgTnV3IEZsaXJ0IGF0IENo cnlzdGhbIBDaGF0IEmp?==?UTF-8?B?dHkh?= Mon Mar 19 20:16:35 2018 Info: MID 19990744 interim AV verdict using Sophos CLEAN Mon Mar 19 20:17:17 2018 Info: MID 19990744 queued for delivery Show all 7 lines
>	3/19/18 8:14:19.000 PM	duration = 140 host = buttercup-mbpr15.splunk.com source = cisco_esa.txt sourcetype = cisco:esa Mon Mar 19 20:14:19 2018 Info: MID 19990744 Outbreak Filters: verdict negative Mon Mar 19 20:14:30 2018 Info: Start MID 19990744 ICID 26006982 Mon Mar 19 20:14:59 2018 Info: MID 19990744 matched all recipients for per-recipient policy Incoming_Mail_Restrictions in the inbound table Mon Mar 19 20:14:59 2018 Info: Start MID 19990744 ICID 26006982 Mon Mar 19 20:18:16 2018 Info: RPT:Message done ICID 8413624 MID 19990744 duration = 237 host = buttercup-mbpr15.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
>	3/19/18 8:14:06.000 PM	Mon Mar 19 20:14:33 2018 Info: MID 19991601 RID [0] Response '2.6.0 <ca18d15j231h@hcp2mailsec.sample.n et> Queued mail for delivery' Mon Mar 19 20:15:01 2018 Info: MID 19991601 queued for delivery Mon Mar 19 20:15:20 2018 Info: Delivery start DCID 8751684 MID 19991601 to RID [0] Mon Mar 19 20:15:41 2018 Info: Message done DCID 8751684 MID 19991601 to RID [0]

By default, only the first 5 events in a transaction are shown. The first transaction contains 7 events and the last event is hidden. The second and third transactions show the **Message done** string in the last event in the transaction.

5. Email transactions based on maxevents, maxspause, and mvlist

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 defines an email transaction as a group of up to 10 events. Each event contains the same value for the `mid` (message ID), `icid` (incoming connection ID), and `dcid` (delivery connection ID). The first and last events in the transaction should be no more than thirty seconds apart.

```
sourcetype="cisco:esa" | transaction mid dcid icid maxevents=10 maxpause=30s mvlist=true
```

By default, the values of multivalue fields are suppressed in search results with the default setting for `mvlist`, which is false. Specifying `mvlist=true` in this search displays all of the values of the selected fields. This produces the following events list:

<input type="checkbox"/> Hide Fields	<input type="checkbox"/> All Fields	i	Time	Event
SELECTED FIELDS		>	3/19/18 8:18:04.000 PM	Mon Mar 19 20:18:20 2018 Info: MID 19998410 RID [0] Response '2.6.0 <7436c832-4f84-4385-a6a1-98835 8db5a51> Queued mail for delivery'
# duration 31				duration = 0 host = buttercup-mbpr15.sv.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
# host 1				
# source 1				
# sourcetype 1				
INTERESTING FIELDS		>	3/19/18 8:18:04.000 PM	Mon Mar 19 20:18:12 2018 Info: MID 19991599 Subject 'Path for the Ranbaxy cases'
# action 1				duration = 0 host = buttercup-mbpr15.sv.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
# answer 3				
# attachment_name 15		>	3/19/18 8:17:56.000 PM	Mon Mar 19 20:17:56 2018 Info: Start MID 19998413 ICID 26006572
# av_scan_result 1				duration = 0 host = buttercup-mbpr15.sv.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
# av_unscannable_reason 1				
# av_verdict 2		>	3/19/18 8:17:56.000 PM	Mon Mar 19 20:17:56 2018 Info: MID 20038555 queued for delivery
# bounce_mid 2				duration = 0 host = buttercup-mbpr15.sv.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
# cisco_esa_dest 3		>	3/19/18 8:17:46.000 PM	Mon Mar 19 20:17:46 2018 Info: MID 19998410 using engine: CASE spam negative
# closed_be 2				duration = 0 host = buttercup-mbpr15.sv.splunk.com source = cisco_esa.txt sourcetype = cisco:esa
# CSRFKey 1				
# date_hour 5		>	3/19/18 8:17:31.000 PM	Mon Mar 19 20:17:31 2018 Info: MID 19998744 ready 15204 bytes from <notify@example.com>
# date_mday 1				Mon Mar 19 20:17:33 2018 Info: Message finished MID 19998744 done
# date_minute 60				Mon Mar 19 20:17:57 2018 Info: RPC Message done RCID 8413824 MID 19998744
# date_month 1				
# date_range 1				
# date_second 60				

Here you can see that each transaction has a duration that is less than thirty seconds. Also, if there is more than one value for a field, each of the values is listed.

6. Transactions with the same session ID and IP address

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 **All time** when you run the search.

Define a transaction as a group of events that have the same session ID, `JSESSIONID`, and come from the same IP address, `clientip`, and where the first event contains the string, "view", and the last event contains the string, "purchase".

```
sourcetype=access_* | transaction JSESSIONID clientip startswith="view" endswith="purchase" | where duration>0
```

The search defines the first event in the transaction as events that include the string, "view", using the `startswith="view"` argument. The `endswith="purchase"` argument does the same for the last event in the transaction.

This example then pipes the transactions into the `where` command and the `duration` field to filter out all of the transactions that took less than a second to complete. The `where` filter cannot be applied before the `transaction` command because the `duration` field is added by the `transaction` command.

```

< Hide Fields   All Fields
    Time      Event
> 4/10/18 198.35.1.75 - [18/Apr/2018:18:18:58] "GET /cart.do?action=view&itemId=EST-12&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1"
6:18:58.000 PM 486 3987 "http://www.buttercupgames.com/product.screen?productId=SF-BVS-Q01" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 959
Transaction with 2 events
198.35.1.75 - [18/Apr/2018:18:18:59] "POST /cart/doAction?do?SESSIONID=SD10SL2FF4ADF53099 HTTP 1.1" 200 2568 "http://www.buttercupgames.com/cart.doAction?purchase&itemId=EST-16" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 384
duration = 1 | host = www1 | source = tutorialdata.zip:www/access.log | sourcetype = access_combined_wcookie

> 4/10/18 198.35.1.75 - [18/Apr/2018:18:18:59] "GET /product.screen?productId=SF-BVS-Q01&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1"
6:18:55.000 PM 1" 508 2889 "http://www.buttercupgames.com/cart.doAction?view&itemId=EST-14" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 378
198.35.1.75 - [18/Apr/2018:18:18:56] "POST /cart/doAction?addtocart&itemId=EST-27&productId=MB-AD-T01&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1" 200 2615 "http://www.buttercupgames.com/product.screen?productId=MB-AD-T01" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 838
198.35.1.75 - [18/Apr/2018:18:18:56] "GET /product.screen?productId=SC-MU-G10&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1" 288 3675 "http://www.buttercupgames.com/category.screen?categoryId=SIMULATION" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 328
198.35.1.75 - [18/Apr/2018:18:18:56] "GET /cart.doAction?addtocart&itemId=EST-4&productId=FS-SG-G03&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1" 200 115 "http://www.buttercupgames.com/product.screen?productId=FS-SG-G03" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 778
198.35.1.75 - [18/Apr/2018:18:18:57] "POST /cart.doAction?purchase&itemId=EST-27&JSESSIONID=SD10SL2FF4ADF53099 HTTP 1.1" 200 3577 "http://www.buttercupgames.com/cart.doAction?addtocart&itemId=EST-27&categoryItemId=TE&productId=MB-AD-T01" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5" 827
duration = 2 | host = www1 | source = tutorialdata.zip:www/access.log | sourcetype = access_combined_wcookie
Transaction with 5 events

```

You might be curious about why the transactions took a long time, so viewing these events might help you to troubleshoot.

You won't see it in this data, but some transactions might take a long time because the user is updating and removing items from their shopping cart before they completes the purchase. Additionally, this search is run over all events. There is no filtering before the `transaction` command. Anytime you can filter the search before the first pipe, the faster the search runs.

See also

Reference

[About transactions](#)

Commands

[stats](#)

[concurrency](#)

Answers

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

transpose

Description

Returns the specified number of rows (search results) as columns (list of field values), such that each search row becomes a column.

Syntax

The required syntax is in **bold**.

```
transpose
[int]
[column_name=<string>]
[header_field=<field>]
[include_empty=<bool>]
```

Required arguments

None.

Optional arguments

column_name

Syntax: column_name=<string>

Description: The name of the first column that you want to use for the transposed rows. This column contains the names of the fields.

Default: column

header_field

Syntax: header_field=<field>

Description: The field in your results to use for the names of the columns (other than the first column) in the transposed data.

Default: row 1, row 2, row 3, and so on.

include_empty

Syntax: include_empty=<bool>

Description: Specify whether to include (true) or not include (false) fields that contain empty values.

Default: true

int

Syntax: <int>

Description: Limit the number of rows to transpose. To transpose all rows, specify | transpose 0, which indicates that the number of rows to transpose is unlimited.

Default: 5

Usage

When you use the `transpose` command the field names used in the output are based on the arguments that you use with the command. By default the field names are: `column`, `row 1`, `row 2`, and so forth.

Examples

1. *Transpose the results of a chart command*

Use the default settings for the transpose command to transpose the results of a chart command.

Suppose you run a search like this:

```
sourcetype=access_* status=200 | chart count BY host
```

The search produces the following search results:

host	count
www1	11835
www2	11186
www3	11261

When you add the `transpose` command to the end of the search, the results look something like this:

column	row 1	row 2	row 3
host	www1	www2	www3
count	11835	11186	11261

2. Count the number of events by sourcetype and transpose the results to display the 3 highest counts

Count the number of events by sourcetype and display the sourcetypes with the highest count first.

```
index=_internal | stats count by sourcetype | sort -count
```

The screenshot shows a Splunk search interface. At the top, there are tabs for "Events (725,349)", "Patterns", "Statistics (9)", and "Visualization". Below the tabs, there are dropdown menus for "20 Per Page", "Format", and "Preview". A search bar contains the query "sourcetype". The main area displays a list of sourcetypes: splunkd, splunkd_ui_access, splunkd_access, splunk_web_service, splunk_web_access, scheduler, mongod, splunkd_stder, and splunkd_conf. The sourcetypes are listed in descending order of count, with splunkd at the top.

Use the transpose command to convert the rows to columns and show the source types with the 3 highest counts.

```
index=_internal | stats count by sourcetype | sort -count | transpose 3
```

Events (725,611)	Patterns	Statistics (2)	Visualization
20 Per Page ▾	Format ▾	Preview ▾	
column ▾	row 1 ▾	row 2 ▾	
sourcetype	splunkd	splunkd_ui_access	spl
count	712168	7272	

3. Transpose a set of data into a series to produce a chart

This example uses the sample dataset from the Search Tutorial.

- Download the data set from [Add data tutorial](#) and follow the instructions to get the tutorial data into your Splunk deployment.

Search all successful events and count the number of views, the number of times items were added to the cart, and the number of purchases.

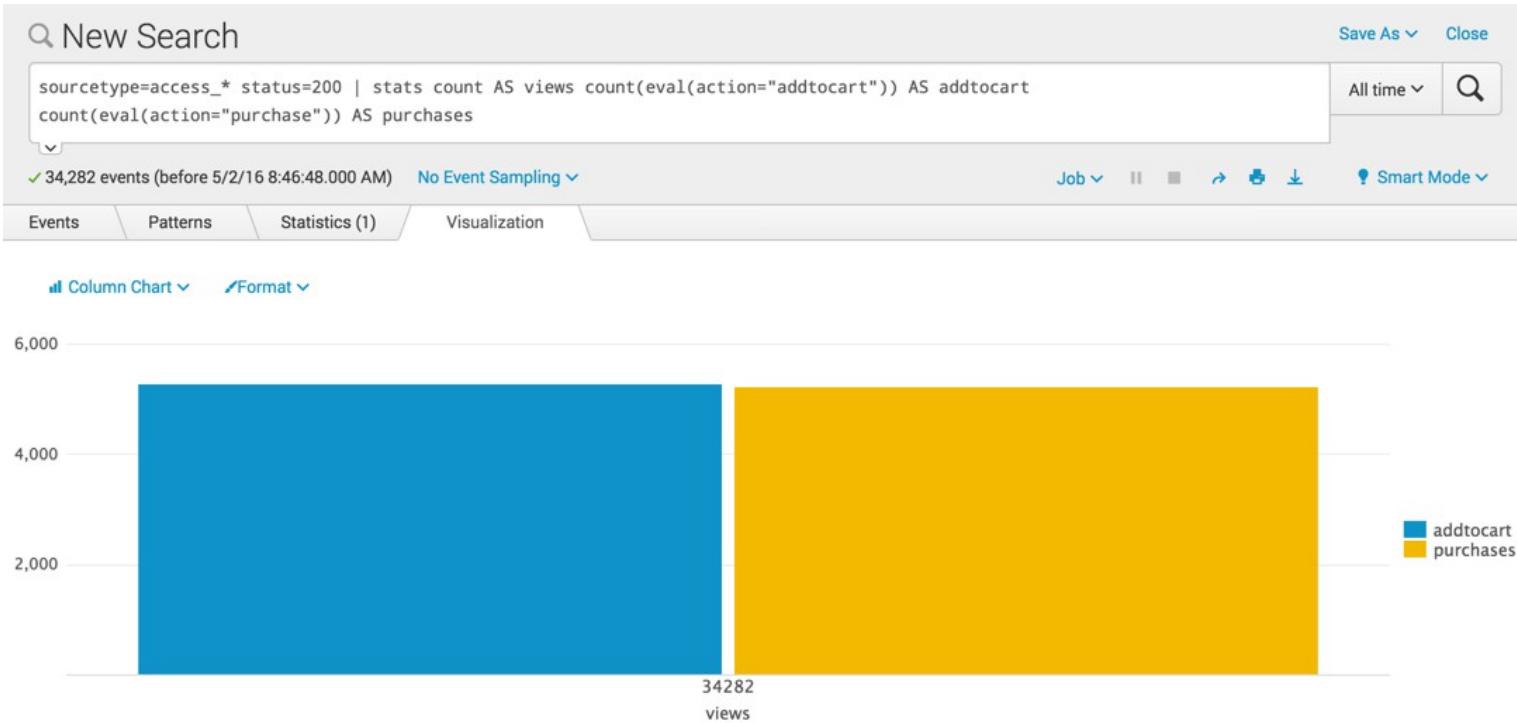
```
sourcetype=access_* status=200 | stats count AS views count(eval(action="addtocart")) AS addtocart  
count(eval(action="purchase")) AS purchases
```

This search produces a single row of data.

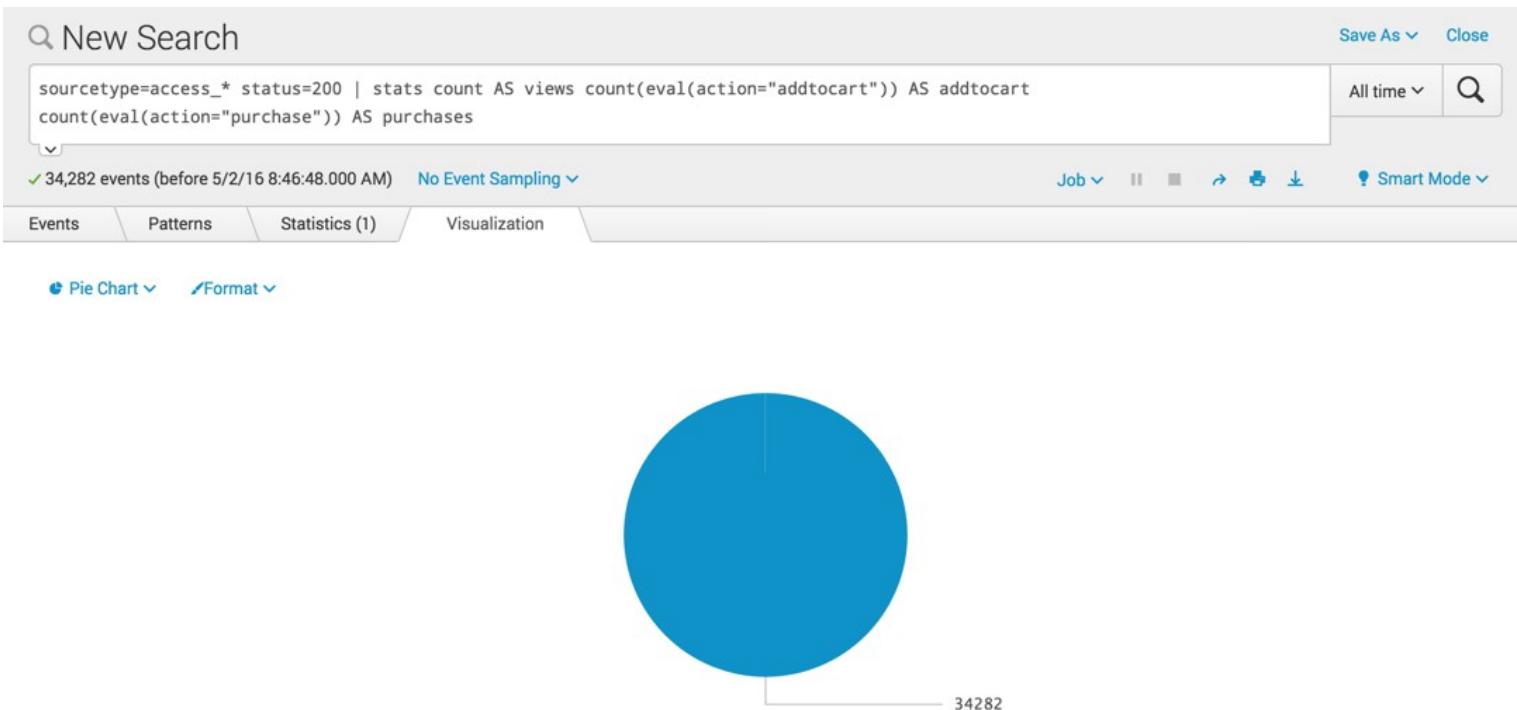
New Search		Save As ▾	Close
sourcetype=access_* status=200 stats count AS views count(eval(action="addtocart")) AS addtocart count(eval(action="purchase")) AS purchases		All time ▾	🔍
✓ 34,282 events (before 5/2/16 8:54:34.000 AM) No Event Sampling ▾		Job ▾	Smart Mode ▾
Events	Patterns	Statistics (1)	Visualization
20 Per Page ▾	Format ▾	Preview ▾	
views ▾	addtocart ▾	purchases ▾	
34282	5292	5224	

The value for count AS views is the total number of the events that match the criteria sourcetype=access_* status=200, or the total count for all actions. The values for addtocart and purchases show the number of events for those specific actions.

When you switch to the Visualization tab, the data displays a chart with the "34282 views" as the X axis label and two columns, one for "addtocart" and one for "purchases". Because the information about the views is placed on the X axis, this chart is confusing.



If you change to a pie chart, you see only the "views".



Use the `transpose` command to convert the columns of the single row into multiple rows.

```
sourcetype=access_* status=200 | stats count AS views count(eval(action="addtocart")) AS addtocart  
count(eval(action="purchase")) AS purchases | transpose
```

New Search

Save As ▾ Close

sourcetype=access_* status=200 | stats count AS views count(eval(action="addtocart")) AS addtocart
count(eval(action="purchase")) AS purchases | transpose

All time ▾

✓ 34,282 events (before 5/2/16 8:55:24.000 AM) No Event Sampling ▾ Job ▾ Smart Mode ▾

Events Patterns Statistics (3) Visualization

20 Per Page ▾

column	row 1
views	34282
addtocart	5292
purchases	5224

Now these rows can be displayed in a column or pie chart where you can compare the values.

New Search

Save As ▾ Close

sourcetype=access_* status=200 | stats count AS views count(eval(action="addtocart")) AS addtocart
count(eval(action="purchase")) AS purchases | transpose

All time ▾

✓ 34,282 events (before 5/2/16 8:55:24.000 AM) No Event Sampling ▾ Job ▾ Smart Mode ▾

Events Patterns Statistics (3) Visualization

Pie Chart

Category	Count
views	34282
addtocart	5292
purchases	5224

In this particular example, using a pie chart is misleading. The views is a total count of all the actions, not just the addtocart and purchases actions. Using a pie chart implies that views is an action like addtocart and purchases. The pie chart implies that the value for views is 1 part of the total, when in fact views is the total.

There are a few ways to fix this issue:

- Use a column chart
- You can remove the `count AS views` criteria from your search
- You can add the `table` command before the `transpose` command in the search, for example:

```
sourcetype=access_* status=200 | stats count AS views count(eval(action="addtocart")) AS addtocart  
count(eval(action="purchase")) AS purchases | table addtocart purchases | transpose
```

See also

Commands

[fields](#)
[stats](#)
[untable](#)
[xseries](#)

trendline

Description

Computes the moving averages of fields: simple moving average (sma), exponential moving average (ema), and weighted moving average (wma). The output is written to a new field, which you can specify.

SMA and WMA both compute a sum over the `period` of most recent values. WMA puts more weight on recent values rather than past values. EMA is calculated using the following formula.

```
EMA(t) = alpha * EMA(t-1) + (1 - alpha) * field(t)
```

where `alpha = 2 / (period + 1)` and `field(t)` is the current value of a field.

Syntax

```
trendline ( <trendtype><period>"(<field>)" [AS <newfield>] )...
```

Required arguments

trendtype

Syntax: sma | ema | wma

Description: The type of trend to compute. Current supported trend types include simple moving average (sma), exponential moving average (ema), and weighted moving average (wma).

period

Syntax: <num>

Description: The period over which to compute the trend, an integer between 2 and 10000.

<field>

Syntax: "("<field>"")"

Description: The name of the field on which to calculate the trend.

Optional arguments

<newfield>

Syntax: <field>

Description: Specify a new field name to write the output to.

Default: <trendtype><period>(<field>)

Usage

Examples

Example 1: Computes a five event simple moving average for field 'foo' and writes the result to new field called 'smoothed_foo.' Also, in the same line, computes ten event exponential moving average for field 'bar'. Because no AS clause is specified, writes the result to the field 'ema10(bar)'.

```
... | trendline sma5(foo) AS smoothed_foo ema10(bar)
```

Example 2: Overlay a trendline over a chart of events by month.

```
index="bar" | stats count BY date_month | trendline sma2(count) AS trend | fields * trend
```

See also

[accum](#), [autoregress](#), [delta](#), [streamstats](#)

tscollect

This feature is deprecated.

The `tscollect` command is deprecated in the Splunk platform as of version 7.3.0. Although this command continues to function, it might be removed in a future version. This command has been superseded by data models. See Accelerate data models in the *Knowledge Manager Manual*.

In the version 7.3.0 Release Notes, see Deprecated features.

Description

The `tscollect` command uses indexed fields to create **time series index (tsidx) files** in a namespace that you define. The result tables in these files are a subset of the data that you have already indexed. This then enables you to use the `tstats` command to search and report on these tsidx files instead of searching raw data. Because you are searching on a subset of the full index, the search should complete faster than it would otherwise.

The `tscollect` command creates multiple tsidx files in the same namespace. The command will begin a new tsidx file when it determines that the tsidx file it is currently creating has gotten big enough.

Only users with the `indexes_edit` capability can run this command. See [Usage](#).

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

```
... | tscollect [namespace=<string>] [squashcase=<bool>] [keepresults=<bool>]
```

Optional arguments

keepresults

Syntax: keepresults = true | false

Description: If true, tscollect outputs the same results it received as input. If false, tscollect returns the count of results processed (this is more efficient since it does not need to store as many results).

Default: false

namespace

Syntax: namespace=<string>

Description: Define a location for the tsidx file(s). If namespace is provided, the tsidx files are written to a directory of that name under the main tsidxstats directory (that is, within \$SPLUNK_DB/tsidxstats). These namespaces can be written to multiple times to add new data.

Default: If namespace is not provided, the files are written to a directory within the job directory of that search, and will live as long as the job does. If you have Splunk Enterprise, you can configure the namespace location by editing `indexes.conf` and setting the attribute `tsidxStatsHomePath`.

squashcase

Syntax: squashcase = true | false

Description: Specify whether or not the case for the entire field::value tokens are case sensitive when it is put into the lexicon. To create indexed field tsidx files that are similar to those created by Splunk Enterprise, set squashcase=true for results to be converted to all lowercase.

Default: false

Usage

You must have the `indexes_edit` capability to run the `tscollect` command. By default, the **admin** role has this capability and the **user** and **power** roles do not have this capability.

Examples

Example 1: Write the results table to tsidx files in namespace foo.

```
... | tscollect namespace=foo
```

Example 2: Retrieve events from the main index and write the values of field foo to tsidx files in the job directory.

```
index=main | fields foo | tscollect
```

See also

[collect](#), [stats](#), [tstats](#)

tstats

Description

Use the `tstats` command to perform statistical queries on indexed fields in `tsidx` files. The indexed fields can be from indexed data or accelerated data models.

Because it searches on index-time fields instead of raw events, the `tstats` command is faster than the `stats` command.

By default, the `tstats` command runs over accelerated and unaccelerated data models.

Syntax

The required syntax is in **bold**.

```
| tstats
[prestats=<bool>]
[local=<bool>]
[append=<bool>]
[summariesonly=<bool>]
[include_reduced_buckets=<bool>]
[allow_old_summaries=<bool>]
[chunk_size=<unsigned int>]
[fillnull_value=<string>]
<stats-func>...
[ FROM datamodel=<data_model_name>.<root_dataset_name> [where nodename =
<root_dataset_name>.<...>.<target_dataset_name>]]
[ WHERE <search-query> | <field> IN (<value-list>)]
[ BY (<field-list> | (PREFIX(<field>))) [span=<timespan>]]
```

Required arguments

<stats-func>

Syntax: (count [<field>] | <function>(PREFIX(<string>) | <field>))... [AS <string>]

Description: Either perform a basic count of a field or perform a function on a field. For a list of the supported functions for the `tstats` command, refer to the table below. You must specify one or more functions. You can apply the function to a field, or to a `PREFIX()` directive if you want to aggregate a raw segment in your indexed events as if it were an extracted field-value pair. You can also rename the result using the `AS` keyword, unless you are in prestats mode (`prestats=true`).

You cannot specify functions without applying them to fields or `eval` expressions that resolve into fields. You cannot use wildcards to specify field names.

See [Usage](#) to learn more about using `PREFIX()`, and about searches you can run to find raw segments in your data.

The following table lists the supported functions by type of function. Use the links in the table to see descriptions and examples for each function. For an overview about using functions with commands, see [Statistical and charting functions](#).

Type of function	Supported functions and syntax			
Aggregate functions	avg() count() distinct_count() estdc()	exactperc<int>() max() median() min() mode()	perc<int>() range() stdev() stdevp()	sum() sumsq() upperperc<int>() var() varp()
Event order functions	first()	last()		
Multivalue stats and chart functions	values()			
Time functions	earliest() earliest_time()	latest() latest_time()	rate()	

Optional arguments

append

Syntax: append=<bool>

Description: When in prestats mode (`prestats=true`), enables `append=true` where the prestats results append to existing results, instead of generating them.

Default: false

allow_old_summaries

Syntax: allow_old_summaries=true | false

Description: Only applies when selecting from an accelerated data model. When you change the constraints that define a data model but the Splunk software has not fully updated the summaries to reflect that change, the summaries may have some data that matches the old definition and some data that matches the new definition. To return results from summary directories only when those directories are up-to-date, set this parameter to `false`. If the data model definition has changed, summary directories that are older than the new definition are not used when producing output from the `tstats` command. This default ensures that the output from `tstats` always reflects your current configuration.

When set to `true`, the `tstats` command uses both current summary data and summary data that was generated prior to the definition change. This is an advanced performance feature for cases where you know that the old summaries are "good enough," meaning the old summary data is close enough to the new summary data that its results are reliable. See [When the data model definition changes and your summaries have not been updated to match it](#) in the Splunk Cloud Platform *Knowledge Manager Manual*.

Default: false

chunk_size

Syntax: chunk_size=<unsigned_int>

Description: Advanced option. This argument controls how many events are retrieved at a time from a single `tsidx` file when the Splunk software processes searches. Lower this setting from its default only when you find a particular `tstats` search is using too much memory, or when it infrequently returns events. This can happen when

a search groups by excessively high-cardinality fields (fields with very large amounts of distinct values). In such situations, a lower `chunk_size` value can make `tstats` searches more responsive, but potentially slower to complete. However, a higher `chunk_size` can help long-running searches to complete faster, with the potential tradeoff of causing the search to be less responsive. For `tstats`, `chunk_size` cannot be set lower than 10000.

Default: 10000000 (10 million)

The default value for the `chunk_size` argument is set by the `chunk_size` setting for the [tstats] stanza in limits.conf. If you have Splunk Cloud Platform, file a Support ticket to change this setting.

fillnull_value

Description: This argument sets a user-specified value that the `tstats` command substitutes for null values for any field within its group-by field list. Null values include field values that are missing from a subset of the returned events as well as field values that are missing from all of the returned events. If you do not provide a `fillnull_value` argument, `tstats` omits rows for events with one or more null field values from its results.

Default: no default value

include_reduced_buckets

Syntax: `include_reduced_buckets=true | false`

Description: This setting only applies when `enableTSIDXReduction=true` in `indexes.conf`. When set to false, the `tstats` command generates results only from index buckets that are not reduced. Set to `true` if you want `tstats` to use results from reduced buckets.

Default: false

local

Syntax: `local=true | false`

Description: If `true`, forces the `tstats` search processor to run only on the search head. This setting is useful for troubleshooting. For example, you can use it to determine whether data on a search head is or has been improperly accelerated. In systems that forward search head data to indexers, this setting may cause the search to produce few or no results. See Best practice: Forward search head data to the indexer layer in Splunk Enterprise *Distributed Search*.

Default: false

prestats

Syntax: `prestats=true | false`

Description: Specifies whether to use the prestats format. The prestats format is a Splunk internal format that is designed to be consumed by commands that generate aggregate calculations. When using the prestats format you can pipe the data into the `chart`, `stats`, or `timechart` commands, which are designed to accept the prestats format. When `prestats=true`, AS instructions are not relevant. The field names for the aggregates are determined by the command that consumes the prestats format and produces the aggregate output.

`prestats=true` is an advanced setting. Use it only in special circumstances when you need to pass tstats-generated data directly to the `chart`, `stats`, or `timechart` command.

Default: false

summariesonly

Syntax: `summariesonly=<bool>`

Description: When `summariesonly` is set to `false`, if the time range of the `tstats` search exceeds the summarization range for the selected data model, the `tstats` command returns results for the entire time range of the search. It quickly returns results from the summarized data, and returns results more slowly from the raw, unsummarized data that exists outside of the data model summary range.

If an accelerated data model is running behind in its summarization, or if its summarization searches are scheduled infrequently, setting `summariesonly = false` might result in a slower `tstats` search. This is because the data model has more unsummarized data to search through than usual.

When `summariesonly` is set to `true`, the `tstats` search returns results only from summarized data, even when the time range of the search exceeds the summarization range of the data model. This means the search runs fast, but no unsummarized data is included in the search results. If you set `summariesonly` to `true`, the `tstats` won't run over unaccelerated data models. Also, when the `tstats` runs over accelerated data models, it returns events only from the data model's acceleration summary.

You might set `summariesonly = true` if you need to identify the data that is currently summarized in a given data model, or if you value search efficiency over completeness of results. See Using the `summariesonly` argument in the Splunk Cloud Platform *Knowledge Manager Manual*.

Default: `false`

FROM clause arguments

The FROM clause is optional. See [Selecting data](#) for more information about this clause.

datamodel

Syntax: `datamodel=<data_model_name>.<root_dataset_name> [where nodename = <root_dataset_name>.<...>.<target_dataset_name>]`

Description: The name of a data model, concatenated with the name of the root dataset that you are searching. If you wish to filter on a child dataset, you need to use a `where` clause that uses `nodename` to reference a specific child dataset in a dataset hierarchy in the data model. See [Selecting data](#) for more information.

WHERE clause arguments

The optional WHERE clause is used as a filter. You can specify either a search or a field and a set of values with the IN operator.

`<search-query>`

Specify the search criteria to filter on.

`<field> IN (<value-list>)`

For the `field`, specify a list of values to include in the search results.

WHERE clauses in `tstat` searches must contain field-value pairs that are indexed, as well as characters that are not **major breakers** or **minor breakers**. For example, consider the following search:

```
| tstats count WHERE index=_internal sourcetype=splunkd* by sourcetype
```

The results look something like this:

sourcetype	count
splunkd	2602154
splunkd_access	319019
splunkd_conf	19

This search returns valid results because `sourcetype=splunkd*` is an indexed field-value pair and wildcard characters are accepted in the search criteria. The asterisk at the end of the `sourcetype=splunkd*` clause is treated as a wildcard, and is

not regarded as either a major or minor breaker.

BY clause arguments

The BY clause is optional. You cannot use wildcards in the BY clause with the tstats command. See [Usage](#). If you use the BY clause, you must specify a field-list. You can also specify a span.

<field-list>

Syntax: <field>, ...

Description: Specify one or more fields to group results.

PREFIX()

Syntax: PREFIX(<string>)

Description: Specify a raw segment in your indexed events that you want to split by as if it were an extracted field-value pair. See [Usage](#) for more information about the PREFIX() directive, and for a search you can run to find raw segments in your indexed data.

span

Syntax: span=<timespan>

Description: The span of each time bin. If you use the BY clause to group by _time, use the span argument to group the time buckets. You can specify timespans such as BY _time span=1h or BY _time span=5d. If you do not specify a <timespan>, the default is auto, which means that the number of time buckets adjusts to produce a reasonable number of results. For example if initially seconds are used for the <timespan> and too many results are being returned, the <timespan> is changed to a longer value, such as minutes, to return fewer time buckets.

Default: auto

<timespan>

Syntax: auto | <int><timescale>

<timescale>

Syntax: <sec> | <min> | <hr> | <day> | <month>

Description: Time scale units. For the tstats command, <timescale> does not support subseconds.

Default: sec

Time scale	Syntax	Description
<sec>	s sec secs second seconds	Time scale in seconds.
<min>	m min mins minute minutes	Time scale in minutes.
<hr>	h hr hrs hour hours	Time scale in hours.
<day>	d day days	Time scale in days.
<month>	mon month months	Time scale in months.

Usage

The tstats command is a **report-generating command**, except when prestats=true. When prestats=true, the tstats command is an **event-generating command**. See [Command types](#).

Generating commands use a leading pipe character and should be the first command in a search, except when prestats=true.

By default, the `tstats` command runs over accelerated and unaccelerated data models.

Properly indexed fields should appear in the `fields.conf` file. See [Create custom fields at index time](#) in *Getting Data In*.

When you use a statistical function with the `tstats` command, you can't use an eval expression as part of the statistical function. See [Complex aggregate functions](#).

Selecting data

Use the `tstats` command to perform statistical queries on indexed fields in `.tsidx` files. You can select the data for the indexed fields in several ways.

Indexed data

Use a `FROM` clause to specify a data model. If you do not specify a `FROM` clause, the Splunk software selects from index data in the same way as the `search` command. You are restricted to selecting data from your allowed indexes by user role. You control exactly which indexes you select data from by using the `WHERE` clause. If no indexes are mentioned in the `WHERE` clause, the Splunk software uses the default indexes. By default, role-based search filters are applied, but can be turned off in the `limits.conf` file.

An accelerated data model

You can select data from a high-performance analytics store, which is a collection of `.tsidx` data summaries, for an accelerated data model. You can select data from this accelerated data model by using `FROM datamodel=<data_model_name>. <root_dataset_name>`.

When you select a data model for a `tstats` search, you also have to select the root dataset within that data model that you intend to search. You cannot select all of the root datasets within a data model at once.

Search filters cannot be applied to accelerated data models. This includes both role-based and user-based search filters.

A child dataset in an accelerated data model

You can select data from a child dataset within an accelerated data model. Use a `WHERE` clause to specify the `nodename` of the child dataset. The `nodename` argument indicates where the target dataset is in the data model hierarchy. The syntax looks like this:

```
... | tstats <stats-func> FROM datamodel=<data_model_name>. <root_dataset_name> where  
nodename=<root_dataset_name>. <...>. <target_dataset_name>
```

For example, say you have a data model with three root datasets, each with their own dataset hierarchies.

```
ButtercupGamesPromos  
- NYC (BaseEvent)  
  - TShirtStore (NYC)  
    - FashionShows (TShirtStore)  
    - Giveaways (TShirtStore)  
- Chicago (BaseEvent)  
  - BeerAndBrautsPopup (Chicago)  
    - BeerSales (BeerAndBrautsPopup)  
    - BrautSales (BeerAndBrautsPopup)  
- Tokyo (BaseSearch)  
  - GiantRobotBattles (Tokyo)  
    - UFORobotGrendizer (GiantRobotBattles)
```

```
- MechaGodzilla (GiantRobotBattles)
```

With this hierarchy, if you wanted to run a `tstats` search that selects from the dataset containing records of the MechaGodzilla giant robot battles staged by the Tokyo office, you would use the following search:

```
... | tstats count FROM datamodel=ButtercupGamesPromos.Tokyo where  
node name=Tokyo.GiantRobotBattles.MechaGodzilla
```

Search filters cannot be applied to accelerated data model datasets. This includes both role-based and user-based search filters.

Filtering data using the WHERE clause

You can use the optional WHERE clause to filter queries with the `tstats` command in much the same ways as you use it with the `search` command. For example, WHERE supports the same time arguments, such as `earliest=-1y`, with the `tstats` command and the `search` command.

WHERE clauses used in `tstats` searches can contain only indexed fields. Fields that are extracted at search time are not supported. If you don't know which of your fields are indexed, run a search on a specific index using the `walklex` command.

Grouping data by _time

You can provide any number of BY fields. If you are grouping by `_time`, supply a timespan with `span` for grouping the time buckets, for example `...BY _time span=1h` or `...BY _time span=3d`.

Limitations

Tstats and tsidx bucket reduction

`tstats` searches over indexes that have undergone tsidx bucket reduction will return incorrect results.

For more information see Reduce tsidx disk usage in *Managing indexers and clusters of indexers*.

Sparkline charts

You can generate sparkline charts with the `tstats` command only if you specify the `_time` field in the BY clause and use the `stats` command to generate the actual sparkline. For example:

```
| tstats count from datamodel=Authentication.Authentication BY _time, Authentication.src span=1h | stats  
sparkline(sum(count),1h) AS sparkline, sum(count) AS count BY Authentication.src
```

Multiple time ranges

The `tstats` command is unable to handle multiple time ranges. This is because the `tstats` command is a generating command and doesn't perform post-search filtering, which is required to return results for multiple time ranges.

The following example of a search using the `tstats` command on events with relative times of 5 seconds to 1 second in the past displays a warning that the results may be incorrect because the `tstats` command doesn't support multiple time ranges.

```
| tstats count where index="_internal" (earliest ==5s latest==4s) OR (earliest==3s latest==1s)
```

If you want to search events in multiple time ranges, use another command such as `stats`, or use multiple `tstats` commands with `append` as shown in the following example.

```
| tstats prestats=t count where index=_internal earliest=-5s latest=-4s | tstats prestats=t append=true  
count where index=_internal earliest=-3s latest=-2s | stats count
```

The results in this example look something like this.

count
264

Wildcard characters

The `tstats` command does not support wildcard characters in field values in aggregate functions or BY clauses.

For example, you cannot specify `| tstats avg(foo*)` or `| tstats count WHERE host=x BY source*`.

Aggregate functions include `avg()`, `count()`, `max()`, `min()`, and `sum()`. For more information, see [Aggregate functions](#).

Any results returned where the aggregate function or BY clause includes a wildcard character are only the most recent few minutes of data that has not been summarized. Include the `summariesonly=t` argument with your `tstats` command to return only summarized data.

Statistical functions must have named fields

With the exception of `count`, the `tstats` command supports only statistical functions that are applied to fields or `eval` expressions that resolve into fields. For example, you cannot specify `| tstats sum` or `| tstats sum()`. Instead the `tstats` syntax requires that at least one field argument be provided for the function: `| tstats sum(<field>)`.

Nested eval expressions not supported

You cannot use `eval` expressions inside aggregate functions with the `tstats` command.

For example, `| tstats count(eval(...))` is not supported.

While nested eval expressions are supported with the `stats` command, they are not supported with the `tstats` command.

Complex aggregate functions

The `tstats` command does not support complex aggregate functions such as
`...count(eval('Authentication.action'=="failure"))`.

Consider the following query. This query will not return accurate results because complex aggregate functions are not supported by the `tstats` command.

```
| tstats count(eval(server.status=200)) from datamodel=internal_server.server where  
nodename=server.splunkdaccess by server.status uri
```

Instead, separate out the aggregate functions from the eval functions, as shown in the following search.

```
| tstats count from datamodel=internal_server.server where nodename=server.splunkdaccess by server.status, uri | eval success=if('server.status'=="200", count, 0) | stats sum(success) as success by uri
```

The results from this search look something like this:

uri	success
//services/cluster/config?output_mode=json	0
//services/cluster/config?output_mode=json	2862
/services/admin/kvstore-collectionstats?count=0	1
/services/admin/transforms-lookup?count=0&getsize=true	1

Limitations of CIDR matching with tstats

As with the `search` command, you can use the `tstats` command to filter events with CIDR match on fields that contain IPv4 and IPv6 addresses. However, unlike the `search` command, the `tstats` command may not correctly filter strings containing non-numeric wildcard octets. As a result, your searches may return unpredictable results.

If you are filtering fields with a CIDR match using the `tstats` command in a BY clause, you can work around this issue and correctly refilter your results by appending your search with a `search` command, `regex` command, or WHERE clause. Unfortunately, you can't use this workaround if the search doesn't include the filtered field in a BY clause.

Example of using CIDR match with tstats in a BY clause

Let's take a look at an example of how you could use CIDR match with the `tstats` command in a BY clause. Say you create a file called `data.csv` containing the following lines:

```
ip,description
1.2.3.4,"An IP address"
5.6.7.8,"Another IP address"
this.is.a.hostname,"A hostname"
this.is.another.hostname,"Another hostname"
```

Then follow these steps:

1. Upload the file and set the sourcetype to `csv`, which ensures that all fields in the file are indexed as required by the `tstats` command.
2. Run the following search against the index you specified when you uploaded the file. This example uses the main index.

```
| tstats count where index=main source=*data.csv ip="0.0.0.0/0" by ip
```

The results look like this:

ip	count
1.2.3.4	1
5.6.7.8	1
this.is.a.hostname	1
this.is.another.hostname	1

Even though only two addresses are legitimate IP addresses, all four rows of addresses are displayed in the results. Invalid IP addresses are displayed along with the valid IP addresses because the `tstats` command uses string matching

to satisfy search requests and doesn't directly support IP address-based searches. The `tstats` command does its best to return the correct results for CIDR search clauses, but the `tstats` search may return more results than you want if the source data contains mixed IP and non-IP data such as host names.

To make sure your searches only return the results you want, make sure that your data set is clean and only contains data in the correct format. If that is not possible, use the `search` command or WHERE clause to do post-filtering of the search results. For example, the following search using the `search` command displays correct results because the piped `search` command further filters the results from the `tstats` command.

```
| tstats count where index=main source=*data.csv ip="0.0.0.0/0" by ip | search ip="0.0.0.0/0"
```

Alternatively, you can use the WHERE clause to filter your results, like this.

```
| tstats count where index=main source=*data.csv ip="0.0.0.0/0" by ip | WHERE cidrmatch("0.0.0.0/0", ip)
```

Both of these searches using the `search` command and the WHERE clause return only the valid IP addresses in the results, which look like this:

ip	count
1.2.3.4	1
5.6.7.8	1

The `tstats` command doesn't respect the `srchTimeWin` parameter

The `tstats` command doesn't respect the `srchTimeWin` parameter in the `authorize.conf` file and other role-based access controls that are intended to improve search performance. This is because the `tstats` command is already optimized for performance, which makes parameters like `srchTimeWin` irrelevant.

For example, say you previously set the `srchTimeWin` parameter on a role for one of your users named Alex, so he is just allowed to run searches back over 1 day. You limited the search time range to prevent searches from running over longer periods of time, which could potentially impact overall system performance and slow down searches for other users. Alex has been running a `stats` search, but didn't notice that he was getting results for just 1 day, even though he specified 30 days. If Alex then changes his search to a `tstats` search, or changes his search in such a way that Splunk software automatically optimizes it to a `tstats` search, the 1 day setting for the `srchTimeWin` parameter no longer applies. As a result, Alex gets many times more results than before, since his search is returning all 30 days of events, not just 1 day of results. This is expected behavior.

Performance

Use `PREFIX()` to aggregate or group by raw tokens in indexed data

The `PREFIX()` directive allows you to search on a raw segment in your indexed data as if it were an extracted field. This causes the search to run over the `tsidx` file in your indexers rather than the log line. This is a practice that can significantly reduce the CPU load on your indexers.

The `PREFIX()` directive is similar to the `CASE()` and `TERM()` directives in that it matches strings in your raw data. You can use `PREFIX()` to locate a recurring **segment** in your raw event data that is actually a key-value pair separated by a delimiter that is also a minor breaker, like `=` or `:`. You give `PREFIX()` the text that precedes the value—the "prefix"—and then it returns the values that follow the prefix. This enables you to group by those values and aggregate them with `tstats` functions. The values can be strings or purely numeric.

For example, say you have indexed segments in your event data that look like `kbps=10` or `kbps=333`. You can isolate the numerical values in these segments and perform aggregations or group-by operations on them by using the `PREFIX()` directive to identify `kbps=` as a common prefix string. Run a `tstats` search with `PREFIX(kbps=)` against your event data and it will return `10` and `333`. These values are perfect for `tstats` aggregation functions that require purely numeric input.

Notice that in this example you need to include the `=` delimiter. If you run `PREFIX(kbps)`, the search returns `=10` and `=333`. Efforts to aggregate on such results may return unexpected results, especially if you are running them through aggregation functions that require purely numeric values.

The text you provide for the `PREFIX()` directive must be in lower case. For example, the `tstats` search processor will fail to process `PREFIX(connectionType=)`. Use `PREFIX(connectiontype=)` instead. It will still match `connectionType=` strings in your events.

The Splunk software separates events into raw segments when it indexes data, using rules specified in `segmenters.conf`. You can run the following search to identify raw segments in your indexed events:

```
| walklex index=<target-index> type=term | stats sum(count) by term
```

You cannot apply the `PREFIX()` directive to segment prefixes and values that contain major breakers such as spaces, square or curly brackets, parentheses, semicolons, or exclamation points.

For more information about the `CASE()` and `TERM()` directives, see [Use CASE\(\) and TERM\(\) to match phrases in the Search Manual](#).

For more information about the segmentation of indexed events, see [About event segmentation in Getting Data In](#)

For more information about minor and major breakers in segments, see [Event segmentation and searching in the Search Manual](#).

Memory and `tstats` search performance

A pair of `limits.conf` settings strike a balance between the performance of `tstats` searches and the amount of memory they use during the search process, in RAM and on disk. If your `tstats` searches are consistently slow to complete you can adjust these settings to improve their performance, but at the cost of increased search-time memory usage, which can lead to search failures.

If you have Splunk Cloud Platform, you need to file a Support ticket to change these settings.

For more information, see [Memory and stats search performance in the Search Manual](#).

Functions and memory usage

Some functions are inherently more expensive, from a memory standpoint, than other functions. For example, the `distinct_count` function requires far more memory than the `count` function. The `values` and `list` functions also can consume a lot of memory.

If you are using the `distinct_count` function without a split-by field or with a low-cardinality split-by field, consider replacing the `distinct_count` function with the `estdgc` function (estimated distinct count). The `estdgc` function might result in significantly lower memory usage and run times.

Examples

1. Get a count of all events in an index

This search tells you how many events there are in the `_internal` index.

```
| tstats count WHERE index=_internal
```

2. Use a filter to get the average

This search returns the average of the field `size` in `myindex`, specifically where `test` is `value2` and the value of `result` is greater than 5. Both `test` and `result` are indexed fields.

```
| tstats avg(size) WHERE index=myindex test=value2 result>5
```

3. Return the count by splitting by source

This search gives the count by source for events with `host=x`.

```
| tstats count WHERE index=myindex host=x by source
```

4. Produce a timechart

This search produces a timechart of all the data in your default indexes with a day granularity. To avoid unpredictable results, the value of the `tstats span` argument should be smaller than or equal to the value of the `timechart span` argument.

```
| tstats prestats=t count WHERE index=_internal BY _time span=1h | timechart span=1d count
```

5. Use summariesonly to get a time range of summarized data

This search uses the `summariesonly` argument to get the time range of the summary for an accelerated data model named `mydm`.

```
| tstats summariesonly=t min(_time) AS min, max(_time) AS max FROM datamodel=mydm | eval prettymin=strftime(min, "%c") | eval prettymax=strftime(max, "%c")
```

6. Find out how much data has been summarized

This search uses `summariesonly` in conjunction with the `timechart` command to reveal the data that has been summarized in 1 hour blocks of time for an accelerated data model called `mydm`.

```
| tstats summariesonly=t prestats=t count FROM datamodel=mydm BY _time span=1h | timechart span=1h count
```

The `span` argument indicates how the events are grouped into buckets or blocks of time, but it doesn't indicate how long the search should run. To run your search over a specific length of time, use the time range picker in the Search app to set the time window for your search. Alternatively, you can include a `WHERE` clause in your search like this, which searches events in 1 hour blocks across a 3 hour time window:

```
| tstats summariesonly=f prestats=t count FROM datamodel=mydm WHERE earliest=-3h BY _time span=1h | timechart span=1h count
```

7. Get a list of values for source returned by the internal log data model

This search uses the `values` statistical function to provide a list of all distinct values for the `source` that is returned by the internal log data model. The list is returned as a multivalue entry.

```
| tstats values(source) FROM datamodel=internal_server
```

The results look something like this:

values(source)
/Applications/Splunk/var/log/splunk/license_usage.log
/Applications/Splunk/var/log/splunk/metrics.log
/Applications/Splunk/var/log/splunk/metrics.log.1
/Applications/Splunk/var/log/splunk/scheduler.log
/Applications/Splunk/var/log/splunk/splunkd.log
/Applications/Splunk/var/log/splunk/splunkd_access.log

If you don't have the `internal_server` data model defined, check under `Settings->Data models` for a list of the data models you have access to.

8. Get a list of values for source returned by the Alerts dataset in the internal log data model

This search uses the `values` statistical function to provide a list of all distinct values for `source` returned by the `Alerts` dataset within the internal log data model.

```
| tstats values(source) FROM datamodel=internal_server where nodename=server.scheduler.alerts
```

9. Get the count and average

This search gets the count and average of a raw, unindexed term using the `PREFIX kbps=`, then splits this by an indexed source and another unindexed term using the `PREFIX group=`.

```
| tstats count avg(PREFIX(kbps=)) where index=_internal by source PREFIX(group=)
```

See also

Commands

[datamodel](#)
[stats](#)
[walklex](#)

typeahead

Description

Returns autosuggest information for a specified prefix that is used to autocomplete word candidates in searches. The maximum number of results returned is based on the value you specify for the `count` argument.

Syntax

The required syntax is in **bold**.

```
| typeahead  
prefix=<string>  
count=<int>  
[collapse=<bool>]  
[endtimeu=<int>]  
[index=<string>]  
[max_servers=<int>]  
[max_time=<int>]  
[starttimeu=<int>]  
[use_cache=<bool>]
```

Required arguments

prefix

Syntax: prefix=<string>

Description: The full search string to return `typeahead` information.

count

Syntax: count=<int>

Description: The maximum number of results to return.

Optional arguments

collapse

Syntax: collapse=<bool>

Description: Specify whether to collapse a term that is a prefix of another term when the event count is the same.

Default: true

endtimeu

Syntax: endtimeu=<int>

Description: Set the end time to N seconds, measured in UNIX time.

Default: now

index-specifier

Syntax: index=<string>

Description: Search the specified index instead of the default index.

max_servers

Syntax: max_servers=<int>

Description: Specify the maximum number of indexers or remote search peers to be used in addition to the search head for `typeahead` searches. If the `max_servers` argument is not specified, the default value is 2, which means Splunk software uses two search peers in addition to any search heads.

Default: 2

max_time

Syntax: max_time=<int>

Description: The maximum time in seconds that the `typeahead` can run. If `max_time=0`, there is no limit.

Default: 1 second

`starttimeu`

Syntax: `starttimeu=<int>`

Description: Set the start time to N seconds, measured in UNIX time.

Default: 0

`use_cache`

Syntax: `use_cache = <boolean>`

Description: Specifies whether the `typeahead` cache will be used if `use_cache` is not specified in the command line or endpoint. When `use_cache` is turned on, Splunk software uses cached search results when running `typeahead` searches, which may have outdated results for a few minutes after you make changes to .conf files.

For more information, see [Typeahead and .conf file updates](#).

Default: true or 1

Usage

The `typeahead` command is a generating command and should be the first command in the search. Generating commands use a leading pipe character.

The `typeahead` command can be targeted to an index and restricted by time.

When you run the `typeahead` command, Splunk software runs internal `typeahead` searches and extracts data from indexes, configurations, and search histories. This information is used to autocomplete word candidates when users type commands in the **Search** bar in Splunk Web. The `typeahead` command extracts data from these sources:

- Indexing field names from indexes.
- Settings in configuration files, such as `props.conf` and `savedsearches.conf`.
- The search history from previous searches in Splunk Web.

The impact of typeahead on search results

The `typeahead` command returns the most common terms found in indexed data with the given prefix. If you use the `typeahead` command with the default settings, the command may not return all search results or the correct search results in the following cases:

- The time to complete the search takes longer than the value specified by the `max_time` argument, which is 1 second, by default.
- Data is indexed on a server that is not randomly chosen, resulting in the exclusion of its data from the search results. This can happen when the value of `max_server` is less than the number of indexers, for example, if `max_server` is set to the default, which is 2, but there are actually 3 indexers.

In addition, the `typeahead` command may not return all of the search results if the `count` argument is set lower than the actual number of results. For example, if the `count` argument is set to 10, the `typeahead` command returns only the top 10 results, even though more results could actually be returned.

Set the number of additional search peers used in a typeahead job

The `max_servers` argument is designed to minimize the workload impact of running `typeahead` search jobs in an indexer clustering environment. For load balancing, the selection of additional search peers for `typeahead` is random.

A setting of 0 removes all limits, causing all available search peers to be used for typeahead search jobs.

The default for the `max_servers` argument is controlled by the `max_servers` setting in `limits.conf`.

Typeahead and .conf file updates

The `typeahead` command uses a cache to run fast searches at the expense of accurate results. As a result, sometimes what is in the cache and shows up in `typeahead` search results may not reflect recent changes to .conf files. This is because it takes 5 or 10 minutes for the cached data to clear, depending on the performance of the server. For example, if you rename a `sourcetype` in the `props.conf` file, it may take a few minutes for that change to display in `typeahead` search results. A `typeahead` search that is run while the cache is being cleared returns the cached data, which is expected behavior.

If you make a change to a .conf file, you can wait a few minutes for the cache to clear to get the most accurate and up-to-date results from your `typeahead` search. Alternatively, you can turn off the `use_cache` argument to clear the cache immediately, which fetches more accurate results, but is a little slower. After you manually clear the cache, you should see the changes to your .conf file reflected in your results when you rerun the `typeahead` search.

For more information, see Rename source types in the Splunk Cloud Platform *Getting Data In* manual.

Typeahead and tsidx bucket reduction

`typeahead` searches over indexes that have undergone tsidx bucket reduction will return incorrect results.

For more information see Reduce tsidx disk usage in *Managing indexers and clusters of indexers*.

Examples

Example 1: Return typeahead information for source

When you run a `typeahead` search, Splunk software extracts information about field definitions from indexes, configurations, and search histories, and displays the relevant information for the specified prefix. For example, say you run the following search for the `source` prefix against the `main` index:

```
| typeahead index=main prefix="source" count=3
```

The `typeahead` command searches the index and shows you what is visible to your users as autocomplete suggestions when they start to type `source` in their searches in Splunk Web. The results look something like this:

content	count	operator
source="access_30DAY.log"	131645	false
source="data.csv"	4	false
source="db_audit_30DAY.csv"	44096	false

Example 2: Return typeahead information for saved searches

You can also run `typeahead` on **saved searches**. For example, say you run this search:

```
|typeahead prefix="savedsearch=" count=3
```

The results look something like this, which tells you what your users see as autocomplete suggestions when they start to type `savedsearch` in the **Search** bar in Splunk Web.

content	count	operator
<code>savedsearch="403_by_clientip"</code>	26	true
<code>savedsearch="Errors in the last 24 hours"</code>	5	true
<code>savedsearch="Errors in the last hour"</code>	2	true

Example 3: Return typeahead information for sourcetypes in the _internal index

When you run the following `typeahead` search, Splunk software returns typeahead information for sourcetypes in the `_internal` index.

```
| typeahead prefix=sourcetype count=5 index=_internal
```

The results look something like this.

content	count	operator
<code>sourcetype</code>	373993	false
<code>sourcetype="mongod"</code>	711	false
<code>sourcetype="scheduler"</code>	2508	false
<code>sourcetype="splunk_btool"</code>	3	false
<code>sourcetype="splunk_intro_disk_objects"</code>	5	false

typelearner

The `typelearner` command is deprecated as of Splunk Enterprise version 5.0. This means that although the command continues to function, it might be removed in a future version. Use the `findtypes` command instead.

Description

Generates suggested event types by taking previous search results and producing a list of potential searches that can be used as event types. By default, the `typelearner` command initially groups events by the value of the grouping-field. The search then unifies and merges these groups based on the keywords they contain.

Syntax

```
typelearner [<grouping-field>] [<grouping-maxlen>]
```

Optional arguments

grouping-field

Syntax: <field>

Description: The field with values for the `typelearner` command to use when initially grouping events.

Default: `punct`, the punctuation seen in `_raw`

grouping-maxlen

Syntax: maxlen=<int>

Description: Determines how many characters in the grouping-field value to look at. If set to negative, the entire value of the grouping-field value is used to group events.

Default: 15

Examples

Example 1:

Have the search automatically discover and apply event types to search results.

```
... | typelearner
```

See also

[typer](#)

typer

Description

Creates an `eventtype` field for search results that match known event types. You must create event types to use this command. See *About event types* in the *Knowledge Manager Manual*.

Syntax

The required syntax is in **bold**.

```
typer
[eventtypes=<string>]
[maxlen=<unsigned_integer>]
```

Required arguments

None.

Optional arguments

eventtypes

Syntax: eventtypes=<string>

Description: Provide a comma-separated list of event types to filter the set of event types that `typer` can return in the `eventtype` field. The `eventtypes` argument filters out all event types except the valid event types in its list. If all of the event types listed for `eventtypes` are invalid, or if no event types are listed, `typer` is disabled and will not return any event types. The `eventtypes` argument accepts wildcards.

Default: No default (by default `typer` returns all available event types)

maxlen

Syntax: maxlen=<unsigned_integer>

Description: By default, the `typer` command looks at the first 10000 characters of an event to determine its event type. Use `maxlen` to override this default. For example, `maxlen=300` restricts `typer` to determining event types from the first 300 characters of events.

Usage

The `typer` command is a distributable streaming command. See [Command types](#).

Changing the default for maxlen

Users with file system access, such as system administrators, can change the default setting for `maxlen`.

Splunk Cloud Platform

To change the `maxlen` default setting, request help from Splunk Support. If you have a support contract, file a new case using the Splunk Support Portal at Support and Services. Otherwise, contact Splunk Customer Support.

Splunk Enterprise

To change the `maxlen` default setting, follow these steps.

Prerequisites

- ◊ Only users with file system access, such as system administrators, can change the `maxlen` default setting using configuration files.
- ◊ Review the steps in How to edit a configuration file in the *Splunk Enterprise Admin Manual*.

Never change or copy the configuration files in the default directory. The files in the default directory must remain intact and in their original location. Make changes to the files in the local directory.

Steps

1. Open or create a local `limits.conf` file for the Search app at `$SPLUNK_HOME/etc/apps/search/local`.
2. Under the `[typer]` stanza, specify the default for the `maxlen` setting.

Examples

Example 1:

Returns a field called `eventtype` which lists the names of the event types associated with the search results.

```
... | typer
```

See also

Commands

[typelearner](#)

Related information

About event types in the *Knowledge Manager Manual*

union

Description

Merges the results from two or more datasets into one dataset. One of the datasets can be a result set that is then piped into the `union` command and merged with a second dataset.

The `union` command appends or merges event from the specified datasets, depending on whether the dataset is streaming or non-streaming and where the command is run. The `union` command runs on indexers in parallel where possible, and automatically interleaves results on the `_time` when processing events. See [Usage](#).

If you are familiar with SQL but new to SPL, see [Splunk SPL for SQL users](#).

Syntax

The required syntax is in **bold**.

```
union
[<subsearch-options>]
<dataset>
[<dataset>...]
```

Required arguments

dataset

Syntax: <dataset-type>:<dataset-name> | <subsearch>

Description: The dataset that you want to perform the union on. The dataset can be either a named or unnamed dataset.

- ◊ A named dataset is comprised of <dataset-type>:<dataset-name>. For <dataset-type> you can specify a **data model**, a **saved search**, or an inputlookup. For example
`datamodel:"internal_server.splunkdaccess".`
- ◊ A subsearch is an unnamed dataset.

When specifying more than one dataset, use a space or a comma separator between the dataset names.

Optional arguments

subsearch-options

Syntax: maxtime=<int> maxout=<int> timeout=<int>

Description: You can specify one set of subsearch-options that apply to all of the subsearches. You can specify one or more of the subsearch-options. These options apply only when the subsearch is treated as a non-streaming search.

- ◊ The `maxtime` argument specifies the maximum number of seconds to run the subsearch before finalizing. The default is 60 seconds.
- ◊ The `maxout` argument specifies the maximum number of results to return from the subsearch. The default is 50000 results. This value is the `maxresultrows` setting is in the [searchresults] stanza in the `limits.conf` file.
- ◊ The `timeout` argument specifies the maximum amount of time, in seconds, to cache the subsearch results. The default is 300 seconds.

Usage

The `union` command is a dataset processing command. See [Command types](#).

How the `union` command processes datasets depends on whether the dataset is a streaming or non-streaming dataset. The type of dataset is determined by the commands that are used to create the dataset. See [Types of commands](#).

There are two types of streaming commands, distributable streaming and centralized streaming. For this discussion about the `union` command, streaming datasets refers to distributable streaming.

A subsearch can be initiated through a search command such as the `union` command. See [Initiating subsearches with search commands](#) in the Splunk Cloud Platform *Search Manual*.

Where the command is run

Whether the datasets are streaming or non-streaming determines if the `union` command is run on the indexers or the search head. The following table specifies where the command is run.

Dataset type	Dataset 1 is streaming	Dataset 1 is non-streaming
Dataset 2 is streaming	Indexers	Search head
Dataset 2 is non-streaming	Search head	Search head

How the command is processed

The type of dataset also determines how the `union` command is processed.

Dataset type	Impact on processing
Centralized streaming or non-streaming	Processed as an <code>append</code> command.
Distributable streaming	Processed as a <code>multisearch</code> command. Placing <code><streaming_dataset1></code> after the <code>union</code> command is more efficient.

Optimized syntax for streaming datasets

With streaming datasets, instead of this syntax:

```
<streaming_dataset1> | union <streaming_dataset2>
```

Your search is more efficient with this syntax:

```
... | union <streaming_dataset1>, <streaming_dataset2>
```

Why unioned results might be truncated

Consider the following search, which uses the `union` command to merge the events from three indexes. Each index contains 60,000 events, for a total of 180,000 events.

```
| union maxout=10000000 [ search index=union_1 ] [ search index=union_2 ] [ search index=union_3 ] | stats count by index
```

This search produces the following union results:

index	count
union_1	60000
union_2	60000
union_3	60000

In this example, all of the subsearches are distributable streaming, so they are unioned by using same processing as the `multisearch` command. All 60,000 results for each index are unioned for a total of 180,000 merged events.

However, if you specify a centralized streaming command, such as the `head` command, in one of the subsearches the results change.

```
| union maxout=10000000 [ search index=union_1 | head 60000 ] [ search index=union_2 ] [ search index=union_3 ] | stats count by index
```

This search produces the following union results for a total of 160,000 merged events.

index	count
union_1	60000
union_2	50000
union_3	50000

Because the `head` command is a centralized streaming command rather than distributable streaming command, any subsearches that follow the `head` command are processed using the `append` command. In other words, when a command forces the processing to the search head, all subsequent commands must also be processed on the search head.

Internally, the search is converted to this:

```
| search index=union_1 | head 60000 | append [ search index=union_2 ] | append [ search index=union_3 ] | stats count by index
```

When the `union` command is used with commands that are non-streaming commands, the default for the `maxout` argument is enforced. The default for the `maxout` argument is 50,000 events. In this example, the default for the `maxout` argument is enforced starting with the subsearch that used the non-streaming command. The default is enforced for any subsequent subsearches.

If the non-streaming command is on the last subsearch, the first two subsearches are processed as streaming. These subsearches are unioned using the `multisearch` command processing. The final subsearch includes a non-streaming command, the `head` command. That subsearch gets unioned using the `append` command processing.

Internally this search is converted to this:

```
| multisearch [ search index=union_1 ] [ search index=union_2 ] | | append [ search index=union_3 | head 60000 ] | stats count by index
```

In this example, the default for the `maxout` argument applies only to the last subsearch. That subsearch returns only 50,000 events instead of the entire set of 60,000 events. The total number events merged is 170,000. 60,000 events for the first and second subsearches and 50,000 events from the last subsearch.

Interleaving results

When two datasets are retrieved from disk in descending time order, which is the default sort order, the `union` command interleaves the results. The interleave is based on the `_time` field. For example, you have the following datasets:

dataset_A

<code>_time</code>	<code>host</code>	<code>bytes</code>
4	mailsrv1	2412
1	dns15	231

dataset_B

<code>_time</code>	<code>host</code>	<code>bytes</code>
3	router1	23
2	dns12	220

Both datasets are descending order by `_time`. When `| union dataset_A, dataset_B` is run, the following dataset is the result.

<code>_time</code>	<code>host</code>	<code>bytes</code>
4	mailsrv1	2412
3	router1	23
2	dns12	220
1	dns15	231

Examples

1. Union events from two subsearches

The following example merges events from `index a` and `index b`. New fields `type` and `mytype` are added in each subsearch using the `eval` command.

```
| union [search index=a | eval type = "foo"] [search index=b | eval mytype = "bar"]
```

2. Union the results of a subsearch to the results of the main search

The following example appends the current results of the main search with the tabular results of errors from the subsearch.

```
... | chart count by category1 | union [search error | chart count by category2]
```

3. Union events from a data model and events from an index

The following example unions a built-in data model that is an internal server log for REST API calls and the events from index a.

```
... | union datamodel:"internal_server.splunkdaccess" [search index=a]
```

4. Specify the subsearch options

The following example sets a maximum of 20,000 results to return from the subsearch. The example specifies to limit the duration of the subsearch to 120 seconds. The example also sets a maximum time of 600 seconds (5 minutes) to cache the subsearch results.

```
... | chart count by category1 | union maxout=20000 maxtime=120 timeout=600 [search error | chart count by category2]
```

See also

Related information

[About subsearches in the *Search Manual*](#)

[About data models in the *Knowledge Manager Manual*](#)

Commands

[search](#)

[inputlookup](#)

uniq

Description

The `uniq` command works as a filter on the search results that you pass into it. This command removes any search result if that result is an exact duplicate of the previous result. This command does not take any arguments.

We do not recommend running this command against a large dataset.

Syntax

```
uniq
```

Examples

Example 1:

Keep only unique results from all web traffic in the past hour.

```
eventtype=webtraffic earliest=-1h@s | uniq
```

See also

[dedup](#)

untabular

Description

Converts results from a tabular format to a format similar to [stats](#) output. This command is the inverse of the [xyseries](#) command.

Syntax

untabular <x-field> <y-name-field> <y-data-field>

Required arguments

<x-field>

Syntax: <field>

Description: The field to use for the x-axis labels or row names. This is the first field in the output.

<y-name-field>

Syntax: <field>

Description: A name for the field to contain the labels for the data series. All of the field names, other than <x-field>, are used as the values for the <y-name-field> field. You can specify any name for this field.

<y-data-field>

Syntax: <field>

Description: A name for the field to contain the data to chart. All of the values from the fields, other than <x-field>, are used as the values for the <y-data-field> field. You can specify any name for this field.

Usage

The `untabular` command is a distributable streaming command. See [Command types](#).

Results with duplicate field values

When you untabular a set of results and then use the `xyseries` command to combine the results, results that contain duplicate values are removed.

You can use the `streamstats` command create unique record numbers and use those numbers to retain all results. See [Extended examples](#).

Basic example

This example uses the sample data from the Search Tutorial. 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 **All time** when you run the search.

To show how to use the `untabular` command, we need results that appear in a table format. Run this search.

```
sourcetype=access_* status=200 action=purchase | top categoryId
```

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

categoryId	count	percent
STRATEGY	806	30.495649
ARCADE	493	18.653046
TEE	367	13.885736
ACCESSORIES	348	13.166856
SIMULATION	246	9.307605
SHOOTER	245	9.269769
SPORTS	138	5.221339

The `top` command automatically adds the count and percent fields to the results.

For each categoryId, there are two values, the count and the percent. When you untable these results, there will be three columns in the output:

- The first column lists the category IDs
- The second column lists the type of calculation: count or percent
- The third column lists the values for each calculation

When you use the `untabular` command to convert the tabular results, you must specify the categoryId field first. You can use any field name you want for the type of calculation and the values. For example:

```
sourcetype=access_* status=200 action=purchase | top categoryId | untabular categoryId calculation value
```

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

categoryId	calculation	value
STRATEGY	count	806
STRATEGY	percent	30.495649
ARCADE	count	493
ARCADE	percent	18.653046
TEE	count	367
TEE	percent	13.885736
ACCESSORIES	count	348
ACCESSORIES	percent	13.166856
SIMULATION	count	246
SIMULATION	percent	9.307605

Extended example

The `untabular` command does exactly what the name says, it converts tabular information into individual rows of results. Suppose you have this search:

```
... | table _time EventCode Message
```

The search produces these results:

_time	EventCode	Message
date-time1	4136	Too late now
date_time2	1234	I dont know
date_time3	3456	Too busy, ask again later
date_time4	1256	Everything is happening at once
date_time4	1257	And right now, as well

Notice that this set of events has duplicate values in the `_time` field for `date_time4`. We will come back to that in a moment.

Use the `untabular` command to remove the tabular format.

```
... | untabular _time FieldName FieldValue
```

Here are the results from the `untabular` command:

_time	FieldName	FieldValue
date-time1	EventCode	4136
date-time1	Message	Too late now
date_time2	EventCode	1234
date-time2	Message	I dont know
date_time3	EventCode	3456
date-time3	Message	Too busy, ask again later
date_time4	EventCode	1256
date-time4	Message	Everything is happening at once
date_time4	EventCode	1257
date_time4	Message	And right now, as well

Events with duplicate timestamps

Remember that the original set of events in this example had duplicates for `date_time4`. If you want to process the events in some way and then put the events back together, you can avoid eliminating the duplicate events by using the `streamstats` command.

Use the `streamstats` command to give each event a unique record number and use that unique number as the key field for the `untablae` and `xseries` commands.

For example, you can add the `streamstats` command to your original search.

```
... | table _time EventCode Message | streamstats count as recno
```

The search produces these results:

_time	EventCode	Message	recno
date-time1	4136	Too late now	1
date_time2	1234	I dont know	2
date_time3	3456	Too busy, ask again later	3
date_time4	1256	Everything is happening at once	4
date_time4	1257	And right now, as well	5

You can then add the `untablae` command to your search, using `recno` as the <x-field>:

```
... | table _time EventCode Message | streamstats count as recno | untablae recno FieldName FieldValue
```

The search produces these results:

recno	FieldName	FieldValue
1	EventCode	4136
1	Message	Too late now
2	EventCode	1234
2	Message	I dont know
3	EventCode	3456
3	Message	Too busy, ask again later
4	EventCode	1256
4	Message	Everything is happening at once
4	EventCode	1257
4	Message	And right now, as well

These events can be put back together by using the `xseries` command, again using the `recno` field as the <x-field>. For example:

```
... | xseries recno FieldName FieldValue
```

The search produces these results:

recno	EventCode	Message
-------	-----------	---------

recno	EventCode	Message
1	4136	Too late now
2	1234	I dont know
3	3456	Too busy, ask again later
4	1256	Everything is happening at once
5	1257	And right now, as well

Restoring the timestamps

In addition to using the `streamstats` command to generate a record number, you can use the `rename` command to restore the timestamp information after the `xseries` command. For example:

```
... | table _time EventCode Message | streamstats count as recno | rename _time as time | untable recno
FieldName FieldValue | xseries recno FieldName FieldValue | rename time as _time
```

(Thanks to Splunk users DalJeanis and BigCosta for their help with this example.)

See also

[xseries](#)

walklex

Description

Generates a list of terms or indexed fields from each bucket of event indexes.



Watch this Splunk How-To video, Using the Walklex Command, to see a demonstration about how to use this command.

Due to the variable nature of merged_lexer.lex and .tsidx files, the walklex command does not always return consistent results. The walklex command doesn't work on hot buckets. This command only works on warm or cold buckets, after the buckets have a merged lexer file or single time-series index (tsidx) file. If neither of these files exist, a message is returned, as expected. This message doesn't indicate that there is a problem with the health of your environment.

Syntax

The required syntax is in **bold**.

```
| walklex
[ type=<walklex-type> ]
[ prefix=<string> | pattern=<wc-string> ]
<index-list>
[ splunk_server=<wc-string> ]
[ splunk_server_group=<wc-string> ]...
```

Required arguments

<index-list>

Syntax: index=<index-name> index=<index-name> ...

Description: Limits the search to one or more indexes. For example, index=_internal.

Optional arguments

prefix | pattern

Syntax: prefix=<string> | pattern=<wc-string>

Description: Limits results to terms that match a specific pattern or prefix. Either prefix or pattern can be specified but not both. Includes only buckets with a merged_lexicon file or a single tsidx file. This means that hot buckets are generally not included.

Default: pattern=*

splunk_server

Syntax: splunk_server=<wc-string>

Description: Specifies the distributed search peers from which to return results.

- ◊ If you are using Splunk Cloud Platform, omit this parameter.
- ◊ If you are using Splunk Enterprise, you can specify only one `splunk_server` argument. However, you can use a wildcard when you specify the server name to indicate multiple servers. For example, you can specify `splunk_server=peer01` or `splunk_server=peer*`. Use `local` to refer to the search head.

Default: All configured search peers return information

splunk_server_group

Syntax: splunk_server_group=<wc-string>

Description: Limits the results to one or more server groups. You can specify a wildcard character in the string to indicate multiple server groups with similar names.

- ◊ If you are using Splunk Cloud Platform, omit this parameter.

Default: None

type

Syntax: type = (all | field | fieldvalue | term)

Description: Specifies which type of terms to return in the lexicon. See [Usage](#) for more information about using the `type` argument options.

- ◊ Use `field` to return only the unique field names in each index bucket.
- ◊ Use `fieldvalue` to include only indexed field terms.
- ◊ Use `term` to exclude all indexed field terms of the form `<field>::<value>`.

Default: all

Usage

The `walklex` command is a **generating command**, which use a leading pipe character. The `walklex` command must be the first command in a search. See [Command types](#).

When the Splunk software indexes event data, it segments each event into raw tokens using rules specified in `segmenters.conf` file. You might end up with raw tokens that are actually key-value pairs separated by an arbitrary delimiter such as an equal (=) symbol.

The following search uses the `walklex` and `where` commands to find the raw tokens in your index. It uses the `stats` command to count the raw tokens.

```
| walklex index=<target-index> | where NOT like(term, "%::%") | stats sum(count) by term
```

Return only indexed field names

Specify the `type=field` argument to have `walklex` return only the field names from indexed fields.

The indexed fields returned by `walklex` can include default fields such as `host`, `source`, `sourcetype`, the `date_*` fields, `punct`, and so on. It can also include additional indexed fields configured as such in `props.conf` and `transforms.conf` and created with the `INDEXED_EXTRactions` setting or other `WRITE_META` methods. The discovery of this last set of additional indexed fields is likely to help you with accelerating your searches.

Return the set of terms that are indexed fields with indexed values

Specify `type=fieldvalue` argument to have `walklex` return the set of terms from the index that are indexed fields with indexed values.

The `type=fieldvalue` argument returns the list terms from the index that are indexed fields with indexed values. Unlike the `type=field` argument, where the values returned are only the field names themselves, the `type=fieldvalue` argument returns indexed field names that have any field value.

For example, if the indexed field term is `runtime::0.04`, the value returned by the `type=fieldvalue` argument is `runtime::0.04`. The value returned by the `type=field` argument is `runtime`.

Return all TSIDX keywords that are not part of an indexed field structure

Specify `type=term` to have `walklex` return the keywords from the TSIDX files that are not part of any indexed field structure. In other words, it excludes all indexed field terms of the form `<field>::<value>`.

Return terms of all three types

When you do not specify a type, or when you specify `type=all`, `walklex` uses the default `type=all` argument. This causes `walklex` to return the terms in the index of all three types: `field`, `fieldvalue`, and `term`.

When you use `type=all`, the indexed fields are not called out as explicitly as the fields are with the `type=field` argument. You need to split the term field on `::` to obtain the field values from the indexed term.

Support for hot buckets

Because the `walklex` command doesn't work on hot buckets, recently loaded data displays in search results only after buckets have rolled over from hot to warm. You can either wait for buckets of an index to roll over from hot to warm on their own, or you can restart Splunk platform or manually roll the buckets over to warm. See Rolling buckets manually from hot to warm.

Restrictions

The `walklex` command applies only to event indexes. It cannot be used with metrics indexes.

People who have **search filters** applied to one or more of their **roles** cannot use `walklex` unless they also have a role with either the `run_walklex` capability or the `admin_all_objects` capability. For more information about role-based search filters, see Create and manage roles with Splunk Web in *Securing the Splunk Platform*. For more information about role-based capabilities, see Define roles on the Splunk platform with capabilities, in *Securing the Splunk Platform*.

Basic examples

1. Return the total count for each term in a specific bucket

The following example returns all of the terms in each bucket of the `_internal` index and finds the total count for each term.

```
| walklex index=_internal | stats sum(count) BY term
```

2. Specifying multiple indexes

The following example returns all of the terms that start with `foo` in each bucket of the `_internal` and `_audit` indexes.

```
| walklex prefix=foo index=_internal index=_audit
```

3. Use a pattern to locate indexed field terms

The following example returns all of the indexed field terms for each bucket that end with `bar` in the `_internal` index.

```
| walklex pattern=*bar type=fieldvalue index=_internal
```

4. Return all field names of indexed fields

The following example returns all of the field names of indexed fields in each bucket of the `_audit` index.

```
| walklex type=field index=_audit
```

See also

Commands

[metadata](#)
[tstats](#)

where

Description

The `where` command uses eval-expressions to filter search results. These eval-expressions must be Boolean expressions, where the expression returns either true or false. The `where` command returns only the results for which the eval expression returns true.

Syntax

```
where <eval-expression>
```

Required arguments

`eval-expression`

Syntax: `<eval-mathematical-expression> | <eval-concatenate-expression> | <eval-comparison-expression> | <eval-boolean-expression> | <eval-function-call>`

Description: A combination of values, variables, operators, and functions that represent the value of your destination field. See [Usage](#).

The <eval-expression> is case-sensitive. The syntax of the eval expression is checked before running the search, and an exception is thrown for an invalid expression.

The following table describes characteristics of eval expressions that require special handling.

Expression characteristics	Description	Example
Field names starting with numeric characters	If the expression references a field name that starts with a numeric character, the field name must be surrounded by single quotation marks.	'5minutes' = "late" This expression is a field name equal to a string value. Because the field starts with a numeric it must be enclosed in single quotations. Because the value is a string, it must be enclosed in double quotations.
Field names with non-alphanumeric characters	If the expression references a field name that contains non-alphanumeric characters, the field name must be surrounded by single quotation marks.	new=count+'server-1' This expression could be interpreted as a mathematical equation, where the dash is interpreted as a minus sign. To avoid this, you must enclose the field name <code>server-1</code> in single quotation marks.
Literal strings	If the expression references a literal string, the literal string must be surrounded by double quotation marks.	new="server- "+count There are two issues with this example. First, <code>server-</code> could be interpreted as a field name or as part of a mathematical equation, that uses a minus sign and a plus sign. To ensure that <code>server-</code> is interpreted as a literal string, enclose the string in double quotation marks.

Usage

The `where` command is a distributable streaming command. See [Command types](#).

The <eval-expression> is case-sensitive.

The `where` command uses the same expression syntax as the `eval` command. Also, both commands interpret quoted strings as literals. If the string is not quoted, it is treated as a field name. Because of this, you can use the `where` command to compare two different fields, which you cannot use the `search` command to do.

Command	Example	Description
Where	... where ipaddress=clientip	This search looks for events where the field <code>ipaddress</code> is equal to the field <code>clientip</code> .
Search	search host=www2	This search looks for events where the field <code>host</code> contains the string value <code>www2</code> .
Where	... where host="www2"	

Command	Example	Description
		This search looks for events where the value in the field <code>host</code> is the string value <code>www2</code> .

Boolean expressions

The order in which Boolean expressions are evaluated with the `where` command is:

1. Expressions within parentheses
2. NOT clauses
3. AND clauses
4. OR clauses

This evaluation order is different than the order used with the `search` command. The `search` command evaluates OR clauses before AND clauses.

Using a wildcard with the where command

You can only specify a wildcard by using the `like` function with the `where` command. The percent (%) symbol is the wildcard that you use with the `like` function. See the [like\(\)](#) evaluation function.

Supported functions

You can use a wide range of evaluation functions with the `where` command. For general information about using functions, see [Evaluation functions](#).

- For a list of functions by category, see [Function list by category](#).
- For an alphabetical list of functions, see [Alphabetical list of functions](#).

Examples

1. Specify a wildcard with the where command

You can only specify a wildcard with the `where` command by using the `like` function. The percent (%) symbol is the wildcard you must use with the `like` function. The `where` command returns `like=TRUE` if the `ipaddress` field starts with the value `198..`

```
... | where like(ipaddress, "198.%")
```

2. Match IP addresses or a subnet using the where command

Return "CheckPoint" events that match the IP or is in the specified subnet.

```
host="CheckPoint" | where like(src, "10.9.165.%") OR cidrmatch("10.9.165.0/25", dst)
```

3. Specify a calculation in the where command expression

Return "physicsjobs" events with a speed is greater than 100.

```
sourcetype=physicsjobs | where distance/time > 100
```

See also

[eval](#), [search](#), [regex](#)

x11

Description

The `x11` command removes the seasonal pattern in your time-based data series so that you can see the real trend in your data. This command has a similar purpose to the [trendline command](#), but it uses the more sophisticated and industry popular X11 method.

The seasonal component of your time series data can be either additive or multiplicative, defined as the two types of seasonality that you can calculate with `x11`: `add()` for additive and `mult()` for multiplicative. See [About time-series forecasting](#) in the *Search Manual*.

Syntax

`x11 [<type>] [<period>] (<fieldname>) [AS <newfield>]`

Required arguments

<fieldname>

Syntax: <field>

Description: The name of the field to calculate the seasonal trend.

Optional arguments

<type>

Syntax: `add()` | `mult()`

Description: Specify the type of `x11` to compute, additive or multiplicative.

Default: `mult()`

<period>

Syntax: <int>

Description: The period of the data relative to the number of data points, expressed as an integer between 5 and 1000. If the period is 7, the command expects the data to be periodic every 7 data points. If you omit this parameter, Splunk software calculates the period automatically. The algorithm does not work if the period is less than 5 and will be too slow if the period is greater than 1000.

<newfield>

Syntax: <string>

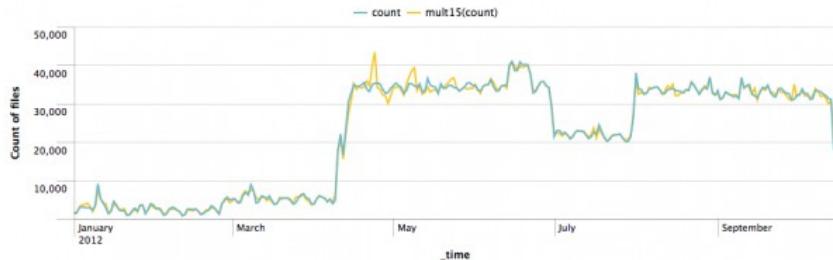
Description: Specify a field name for the output of the `x11` command.

Default: None

Examples

Example 1: In this example, the type is the default `mult` and the period is 15. The field name specified is `count`.

```
index=download | timechart span=1d count(file) as count | x11 mult15(count)
```



Because `span=1d`, every data point accounts for 1 day. As a result, the period in this example is 15 days. You can change the syntax in this example to `... | x11 15(count)` because the `mult` type is the default type.

Example 2: In this example, the type is `add` and the period is 20. The field name specified is `count`.

```
index=download | timechart span=1d count(file) as count | x11 add20(count)
```



See also

[predict](#), [trendline](#)

xmlkv

Description

The `xmlkv` command automatically extracts key-value pairs from XML-formatted data.

For JSON-formatted data, use the `spath` command.

Syntax

The required syntax is in **bold**.

```
xmlkv
[<field>]
maxinputs=<int>
```

Required arguments

None.

Optional arguments

field

Syntax: <field>

Description: The field from which to extract the key and value pairs.

Default: The `_raw` field.

maxinputs

Syntax: maxinputs=<int>

Description: Sets the maximum number of events or search results that can be passed as inputs into the `xmlkv` command per invocation of the command. The `xmlkv` command is invoked repeatedly in increments according to the `maxinputs` argument until the search is complete and all of the results have been displayed. Do not change the value of `maxinputs` unless you know what you are doing.

Default: 50000

Usage

The `xmlkv` command is a distributable streaming command. See [Command types](#).

Keys and values in XML elements

From the following XML, `name` is the key and `Settlers of Catan` is the value in the first element.

```
<game>
  <name>Settlers of Catan</name>
  <category>competitive</category>
</game>
<game>
  <name>Ticket to Ride</name>
  <category>competitive</category>
</game>
```

Examples

1. Automatically extract key-value pairs

Extract key-value pairs from XML tags in the `_raw` field. Processes a maximum of 50000 events.

```
... | xmlkv
```

2. Extract key-value pairs in a specific number of increments

Extract the key-value pairs from events or search results in increments of 10,000 per invocation of the `xmlkv` command until the search has finished and all of the results are displayed.

```
... | xmlkv maxinputs=10000
```

See also

Commands

[extract](#)
[kvform](#)
[multikv](#)
[rex](#)
[spath](#)
[xpath](#)

xmlunescape

Description

Un-escapes `xml` characters, including entity references such as `&`, `<`, and `>`, so that they return to their corresponding characters. For example, `&` becomes `&`.

The `xmlunescape` command is a streaming command. It is distributable streaming by default, but centralized streaming if the `local` setting specified for the command in the `commands.conf` file is set to true. See [Command types](#).

Syntax

```
xmlunescape maxinputs=<int>
```

Optional arguments

`maxinputs`

Syntax: `maxinputs=<int>`

Description: The maximum number of inputs per invocation of the command. The `xmlunescape` command is invoked repeatedly in increments according to the `maxinputs` argument until the search is complete and all of the results have been displayed. Do not change the value of `maxinputs` unless you know what you are doing.

Default: 50000

Examples

Example 1: Un-escape all XML characters.

```
... | xmlunescape
```

xpath

Description

Extracts the `xpath` value from `field` and sets the `outfield` attribute.

Syntax

`xpath [outfield=<field>] <xpath-string> [field=<field>] [default=<string>]`

Required arguments

`xpath-string`

Syntax: `<string>`

Description: Specifies the XPath reference.

Optional arguments

`field`

Syntax: `field=<field>`

Description: The field to find and extract the referenced `xpath` value from.

Default: `_raw`

`outfield`

Syntax: `outfield=<field>`

Description: The field to write, or output, the `xpath` value to.

Default: `xpath`

`default`

Syntax: `default=<string>`

Description: If the attribute referenced in `xpath` doesn't exist, this specifies what to write to the `outfield`. If this isn't defined, there is no default value.

Usage

The `xpath` command is a distributable streaming command. See [Command types](#).

The `xpath` command supports the syntax described in the Python Standard Library 19.7.2.2. Supported XPath syntax.

Examples

1. Extract values from a single element in `_raw` XML events

You want to extract values from a single element in `_raw` XML events and write those values to a specific field.

The `_raw` XML events look like this:

```
<foo>
    <bar nickname="spock">
    </bar>
</foo>
<foo>
    <bar nickname="scotty">
    </bar>
</foo>
<foo>
    <bar nickname="bones">
    </bar>
```

```
</foo>
```

Extract the nickname values from _raw XML events. Output those values to the name field.

```
sourcetype="xml" | xpath outfield=name "//bar/@nickname"
```

2. Extract multiple values from _raw XML events

Extract multiple values from _raw XML events

The _raw XML events look like this:

```
<DataSet xmlns="">
    <identity_id>3017669</identity_id>
    <instrument_id>912383KM1</instrument_id>
    <transaction_code>SEL</transaction_code>
    <sname>BARC</sname>
    <currency_code>USA</currency_code>
</DataSet>

<DataSet xmlns="">
    <identity_id>1037669</identity_id>
    <instrument_id>219383KM1</instrument_id>
    <transaction_code>SEL</transaction_code>
    <sname>TARC</sname>
    <currency_code>USA</currency_code>
</DataSet>
```

Extract the values from the identity_id element from the _raw XML events:

```
... | xpath outfield=identity_id "//DataSet/identity_id"
```

This search returns two results: identity_id=3017669 and identity_id=1037669.

To extract a combination of two elements, sname with a specific value and instrument_id, use this search:

```
... | xpath outfield=instrument_id "//DataSet[sname='BARC']/instrument_id"
```

Because you specify sname='BARC', this search returns one result: instrument_id=912383KM1.

3. Testing extractions from XML events

You can use the makeresults command to test xpath extractions.

You must add field=xml to the end of your search. For example:

```
| makeresults
| eval xml=<DataSet xmlns="">
    <identity_id>1037669</identity_id>
    <instrument_id>219383KM1</instrument_id>
    <transaction_code>SEL</transaction_code>
    <sname>TARC</sname>
    <currency_code>USA</currency_code>
</DataSet>"
```

```
| xpath outfield=identity_id "//DataSet/identity_id" field=xml
```

See also

[extract](#), [kvform](#), [multikv](#), [rex](#), [spath](#), [xmlkv](#)

xyseries

This topic walks through how to use the `xyseries` command.

Description

Converts results into a tabular format that is suitable for graphing. This command is the inverse of the [untable](#) command.

Syntax

```
xyseries [grouped=<bool>] <x-field> <y-name-field> <y-data-field>... [sep=<string>] [format=<string>]
```

Required arguments

<x-field>

Syntax: <field>

Description: The name of the field to use for the x-axis label. The values of this field appear as labels for the data series plotted on the x-axis.

<y-name-field>

Syntax: <field>

Description: The field that contains the values to use as labels for the data series.

<y-data-field>

Syntax: <field> [,<field>] ...

Description: One or more fields that contain the data to chart. When specifying multiple fields, separate the field names with commas.

Optional arguments

format

Syntax: format=<string>

Description: Used to construct output field names when multiple data series are used in conjunction with a split-by-field and separate the <y-name-field> and the <y-data-field>. `format` takes precedence over `sep` and lets you specify a parameterized expression with the stats aggregator and function (\$AGG\$) and the value of the split-by-field (\$VALUE\$).

grouped

Syntax: grouped= true | false

Description: If true, indicates that the input is sorted by the value of the <x-field> and multifile input is allowed.

Default: false

sep

Syntax: sep=<string>

Description: Used to construct output field names when multiple data series are used in conjunctions with a split-by field. This is equivalent to setting `format` to `AGG<sep>$VALUE$`.

Usage

The `xseries` command is a **distributable streaming command**, unless `grouped=true` is specified and then the `xseries` command is a **transforming command**. See [Command types](#).

Alias

The alias for the `xseries` command is `maketable`.

Results with duplicate field values

When you use the `xseries` command to converts results into a tabular format, results that contain duplicate values are removed.

You can use the `streamstats` command create unique record numbers and use those numbers to retain all results. For an example, see the [Extended example](#) for the [untabular command](#).

Example

Let's walk through an example to learn how to reformat search results with the `xseries` command.

Write a search

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 **All time** when you run the search.

Run this search in the search and reporting app:

```
sourcetype=access_* status=200 action=purchase | top categoryId
```

The `top` command automatically adds the count and percent fields to the results. For each categoryId, there are two values, the count and the percent.

The search results look like this:

categoryId	count	percent
STRATEGY	806	30.495649
ARCADE	493	18.653046
TEE	367	13.885736
ACCESSORIES	348	13.166856
SIMULATION	246	9.307605
SHOOTER	245	9.269769
SPORTS	138	5.221339

Identify your fields in the xyseries command syntax

In this example:

- <x-field> = categoryId
- <y-name-field> = count
- <y-data-field> = percent

Reformat search results with xyseries

When you apply the `xyseries` command, the `categoryId` serves as the <x-field> in your search results. The results of the calculation `count` become the columns, <y-name-field>, in your search results. The <y-data-field>, `percent`, corresponds to the values in your search results.

Run this search in the search and reporting app:

```
sourcetype=access_* status=200 action=purchase | top categoryId | xyseries categoryId count percent
```

The search results look like this:

categoryId	138	245	246	348	367	493	806
SPORTS	5.221339						
ACCESSORIES				13.166856			
ARCADE					18.653046		
SHOOTER		9.269769					
SIMULATION			9.307605				
STRATEGY						30.495649	
TEE					13.885736		

Extended example

Let's walk through an example to learn how to add optional arguments to the `xyseries` command.

Write a search

To add the optional arguments of the `xyseries` command, you need to write a search that includes a split-by-field command for multiple aggregates. Use the `sep` and `format` arguments to modify the output field names in your search results.

Run this search in the search and reporting app:

```
sourcetype=access_combined_wcookie | stats count(host) count(productId) by clientip, referer_domain
```

This search sorts referer domain, `count(host)` and `count(productId)` by `clientIp`.

clientip	referer_domain	count(host)	count(productId)
107.3.146.207	http://www.bing.com	3	
107.3.146.207	http://www.buttercupgames.com	358	
107.3.146.207	http://www.google.com	10	
107.3.146.207	http://www.yahoo.com	13	
108.65.113.83	http://www.buttercupgames.com	227	
108.65.113.83	http://www.google.com	13	

Run this search in the search and reporting app:

```
sourcetype=access_combined_wcookie | stats count(host) count(productId) by clientip, referer_domain | xseries clientip referer_domain count(host), count(productId)
```

In this example:

- <x-field> = clientip
- <y-name-field> = referrer domain
- <y-data-field> = host, productId

The xseries command needs two aggregates, in this example they are: count(host) count(productId). The first few search results look like this:

clientip	count(host): http://www.bing.com	count(host): http://www.buttercupgames.com	count(host): http://www.google.com	count(host): http://www.yahoo.com	count(productId): http://www.bing.com
107.3.146.207	3	358	10	13	1
108.65.113.83		227	13	3	
109.169.32.135	2	400	13	7	1
110.138.30.229	1	159	10	5	1
110.159.208.78	2	241	9	4	0
111.161.27.20	2	192	9	4	2

Add optional argument: sep

Add a string to the sep argument to change the default character that separates the <y-name-field> host, and the <y-data-field> productId. The format argument adds the <y-name-field> and separates the field name and field value by the default ":".

Run this search in the search and reporting app:

```
sourcetype=access_combined_wcookie | stats count(host) count(productId) by clientip, referer_domain | xseries clientip referer_domain count(host), count(productId) sep="-"
```

The first few search results look like this:

clientip ↴ ↵	count(host)- ✓ http://www.bing.com TEST ↴ ↵	count(host)- ✓ http://www.buttercupgames.com TEST ↴ ↵	count(host)- ✓ http://www.google.com TEST ↴ ↵	count(host)- ✓ http://www.yahoo.com TEST ↴ ↵	count(productId)- ✓ http://www.bing.com TEST ↴ ↵
107.3.146.207	3	358	10	13	1
108.65.113.83		227	13	3	
109.169.32.135	2	400	13	7	1
110.138.30.229	1	159	10	5	1
110.159.208.78	2	241	9	4	0
111.161.27.20	2	192	9	4	2

Add optional argument: format

The format argument adds the <y-name-field> and separates the field name and field value by the default ":" For example, the default for this example looks like count(host):referrer_domain

When you specify a string to separate the <y-name-field> and <y-data-field> with the format argument, it overrides any assignment from the sep argument. In the following example, the sep argument assigns the "-" character to separate the <y-name-field> and <y-data-field> fields. The format argument assigns a "+" and this assignment takes precedence over sep. In this case \$VAL\$ and \$AGG\$ represent both the <y-name-field> and <y-data-field>. As seen in the search results, the <y-name-field>, host, and <y-data-field>, productId can correspond to either \$VAL\$ or \$AGG\$.

Run this search in the search and reporting app:

```
sourcetype=access_combined_wcookie | stats count(host) count(productId) by clientip, referer_domain | xyseries clientip referer_domain count(host), count(productId) sep="-" format="$AGG$ + $VAL$ TEST"
```

The first few search results look like this:

clientip ↴ ↵	count(host) + ✓ http://www.bing.com TEST ↴ ↵	count(host) + ✓ http://www.buttercupgames.com TEST ↴ ↵	count(host) + ✓ http://www.google.com TEST ↴ ↵	count(host) + ✓ http://www.yahoo.com TEST ↴ ↵	count(productId) + ✓ http://www.bing.com TEST ↴ ↵
107.3.146.207	3	358	10	13	1
108.65.113.83		227	13	3	
109.169.32.135	2	400	13	7	1
110.138.30.229	1	159	10	5	1
110.159.208.78	2	241	9	4	0
111.161.27.20	2	192	9	4	2

Add optional argument: grouped

The grouped argument determines whether the `xseries` command runs as a **distributable streaming command**, or a **transforming command**. The default state grouped=FALSE for the `xseries` command runs as a streaming command.

See also

Commands

[untable](#)

3rd party custom commands

Welcome to the Search Reference. See the left navigation panel for links to the built-in search commands. If you don't find a command in the list, that command might be part of a third-party app or add-on. For information about commands contributed by apps and add-ons, see the documentation on Splunkbase.

Internal Commands

About internal commands

Internal search commands refer to a set of commands that are designed to be used in specific situations, typically at the direction and with guidance from Splunk Support. These commands might be removed, or updated and reimplemented differently, in future versions.

Consult your Splunk Administrator or Splunk Support before using any of these commands.

- [collapse](#)
- [dump](#)
- [findkeywords](#)
- [makejson](#)
- [mcatalog](#)
- [noop](#)
- [pjob](#)
- [redistribute](#)
- [runshellscrip](#)

collapse

The collapse command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

The collapse command condenses multifile results into as few files as the `chunksize` option allows. This command runs automatically when you use `outputlookup` and `outputcsv` commands.

Syntax

`... | collapse [chunksize=<num>] [force=<bool>]`

Optional arguments

`chunksize`

Syntax: `chunksize=<num>`

Description: Limits the number of resulting files.

Default: 50000

`force`

Syntax: `force=<bool>`

Description: If `force=true` and the results are entirely in memory, re-divide the results into appropriated chunked files.

Default: `false`

Examples

Example 1: Collapse results.

```
... | collapse
```

dump

The dump command is an internal, unsupported, experimental command. See About internal commands.

Description

For Splunk Enterprise deployments, export search results to a set of chunk files on local disk. For information about other export methods, see Export search results in the *Search Manual*.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See SPL safeguards for risky commands in Securing the Splunk Platform.

Syntax

Required syntax is in **bold**:

```
dump
basefilename=<string>
[fields=<comma-delimited-string>]
[rollsize=<number>]
[compress=<number>]
[format=<string>]
```

Required arguments

basefilename

Syntax: basefilename=<string>

Description: The prefix of the export filename.

Optional arguments

compress

Syntax: compress=<number>

Description: The gzip compression level. Specify a number from 0 to 9, where 0 means no compression and a higher number means more compression and slower writing speed.

Default: 2

fields

Syntax: fields=<comma-delimited-string>

Description: The list of the fields to export. The entire list must be enclosed in quotation marks. Invalid field names are ignored.

format

Syntax: format= raw | csv | tsv | json | xml

Description: The output data format.

Default: raw

rollsize

Syntax: rollsize=<number>

Description: The minimum file size, in MB, at which point no more events are written to the file and it becomes a candidate for HDFS transfer.

Default: 63 MB

Usage

This command exports events to a set of chunk files on local disk at

"\$SPLUNK_HOME/var/run/splunk/dispatch/<sid>/dump". This command recognizes a special field in the input events, _dstpath, which if set is used as a path to be appended to the `dst` directory to compute the final destination path.

The `dump` command preserves the order of events as the events are received by the command.

Capability required

The `dump` command is considered to be a potentially risky command. To use this command, you must have a role with the `run_dump` capability. See Define roles on the Splunk platform with capabilities.

For more information about risky commands, see SPL safeguards for risky commands.

Examples

Example 1: Export all events from index "bigdata" to the location "YYYYmmdd/HH/host" at "\$SPLUNK_HOME/var/run/splunk/dispatch/<sid>/dump/" directory on local disk with "MyExport" as the prefix of export filenames. Partitioning of the export data is achieved by eval preceding the dump command.

```
index=bigdata | eval _dstpath=strftime(_time, "%Y%m%d/%H") + "/" + host | dump basename=MyExport
```

Example 2: Export all events from index "bigdata" to the local disk with "MyExport" as the prefix of export filenames.

```
index=bigdata | dump basename=MyExport
```

findkeywords

The `findkeywords` command is an internal, unsupported, experimental command. See About internal commands.

Description

Given some integer labeling of events into groups, finds searches to generate these groups.

Syntax

findkeywords labelfield=<field>

Required arguments

labelfield

Syntax: labelfield=<field>

Description: A field name.

Usage

Use the `findkeywords` command after the `cluster` command, or a similar command that groups events. The `findkeyword` command takes a set of results with a field (labelfield) that supplies a partition of the results into a set of groups. The command derives a search to generate each of these groups. This search can be saved as an **event type**.

Examples

Return logs for specific log_level values and group the results

Return all logs where the `log_level` is DEBUG, WARN, ERROR, FATAL and group the results by cluster count.

```
index=_internal source=*splunkd.log* log_level!=info | cluster showcount=t | findkeywords  
labelfield=cluster_count
```

The result is a statistics table:

The screenshot shows the Splunk interface with the following details:

- Search & Reporting** tab is selected.
- New Search** button is visible.
- Search Bar:** index=_internal source=*splunkd.log* log_level!=info | cluster showcount=t | findkeywords labelfield=cluster_count
- Results:** 15 events found before 8/19/14 7:32:08.000 PM.
- Statistics Table Headers:** confidence, eventTypeable, excludeKeywords, groupID, includeKeywords, numInInputGroup, numMatched, percentInInputGroup, percentMatched, sampleEvent.
- Table Data:** Four rows of data are shown, corresponding to the four log levels (WARN, ERROR, FATAL, DEBUG).

confidence	eventTypeable	excludeKeywords	groupID	includeKeywords	numInInputGroup	numMatched	percentInInputGroup	percentMatched	sampleEvent
0.690309	1		1		12	15	0.800000	1.000000	08-19-2014 14:22:23.512 +0200 WARN /splunkbeta/etc/system/default/searchb/parse into key-value pair; if applicable.
0.000000	1		10		1	0	0.066667	0.000000	08-19-2014 14:21:28.096 +0200 WARN C time found in the next 1051200 minutes
0.000000	1		3		1	0	0.066667	0.000000	08-19-2014 18:25:53.478 +0200 ERROR I get info for non-existent user=""
0.000000	1		2		1	0	0.066667	0.000000	08-19-2014 14:20:55.947 +0200 WARN D parse timestamp. Defaulting to timestamp Nov 5 00:00:00 2012). Context: source:/etc/apps/splunk_6.2_overview /data/violations_plus.csv host:splunk_ci

The values of `groupID` are the values of `cluster_count` returned from the `cluster` command.

See also

[cluster](#), [findtypes](#)

makejson

The `makejson` command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

Creates a JSON object from the specified set of fields in the search results, and places the JSON object into a new field.

Syntax

`makejson <wc-field-list> output=<string>`

Required arguments

`output`

Syntax: `output=<string>`

Description: The name to use for the output field where the JSON object is placed.

Optional arguments

`wc-field-list`

Syntax: `<field>(<field>) ...`

Description: Comma-delimited list of fields to use to generate a JSON object. You can use a wild card character in the field names.

Default: All fields are included in the JSON object if a list is not specified.

Usage

You cannot use the `table` or `fields` command to specify the field order in the JSON object that gets created.

Examples

1. Create a JSON object using all of the available fields

The following search create a JSON object in a field called "data" taking in values from all available fields.

```
| makeresults count=5 | eval owner="vladimir", error=random()%3 | makejson output=data
```

- The `makeresults` command creates five search results that contain a timestamp.
- The `eval` command creates two fields in each search result. One field is named `owner` and contains the value `vladimir`. The other field is named `error` that takes a random number and uses the modulo mathematical operator (`%`) to divide the random number by 3.

- The `makejson` command creates a JSON object based on the values in the fields in each search result.

The results look something like this:

<code>_time</code>	<code>owner</code>	<code>error</code>	<code>data</code>
2020-03-10 21:45:14	vladimir	1	{"owner": "vladimir", "error": 1, "_time": 1583901914}
2020-03-10 21:45:14	vladimir	0	{"owner": "vladimir", "error": 0, "_time": 1583901914}
2020-03-10 21:45:14	vladimir	0	{"owner": "vladimir", "error": 0, "_time": 1583901914}
2020-03-10 21:45:14	vladimir	2	{"owner": "vladimir", "error": 2, "_time": 1583901914}
2020-03-10 21:45:14	vladimir	1	{"owner": "vladimir", "error": 1, "_time": 1583901914}

2. Create a JSON object from a specific set of fields

Consider the following data:

<code>_time</code>	<code>owner</code>	<code>error_code</code>
2020-03-10 21:45:14	claudia	1
2020-03-10 20:45:17	alex	4
2020-03-10 06:48:11	wei	2
2020-03-09 21:15:35	david	3
2020-03-09 16:22:10	maria	4
2020-03-08 23:32:56	vanya	1
2020-03-07 14:05:14	claudia	2

The `makejson` command is used to create a JSON object in a field called "data" using the values from only the `_time` and `owner` fields. The `error` field is not included in the JSON object.

```
| makeresults count=7 | eval owner="claudia", error=random()%5 | makejson _time, owner output=data
```

The results look something like this:

<code>data</code>
{"owner": "claudia", "_time": 1583876714}
{"owner": "alex", "_time": 1583873117}
{"owner": "wei", "_time": 1583822891}
{"owner": "david", "_time": 1583788535}
{"owner": "maria", "_time": 1583770930}
{"owner": "vanya", "_time": 1583710376}
{"owner": "claudia", "_time": 1583589914}

data

3. Create a JSON object using a wildcard list of fields

Create a JSON object in a field called "json-object" using the values from the `_time` field and fields that end in `_owner`.

```
| makeresults count=5 | eval product_owner="wei", system_owner="vanya", error=random()%5 | makejson _time, *_owner output="json-object"
```

The results look something like this:

<code>_time</code>	<code>product_owner</code>	<code>system_owner</code>	<code>error</code>	<code>json-object</code>
2020-03-10 22:23:24	wei	vanya	3	{"product_owner": "wei", "system_owner": "vanya", "_time": 1583904204}
2020-03-10 22:23:24	wei	vanya	2	{"product_owner": "wei", "system_owner": "vanya", "_time": 1583904204}
2020-03-10 22:23:24	wei	vanya	1	{"product_owner": "wei", "system_owner": "vanya", "_time": 1583904204}
2020-03-10 22:23:24	wei	vanya	3	{"product_owner": "wei", "system_owner": "vanya", "_time": 1583904204}
2020-03-10 22:23:24	wei	vanya	2	{"product_owner": "wei", "system_owner": "vanya", "_time": 1583904204}

4. Use with schema-bound lookups

You can use the `makejson` command with schema-bound lookups to store a JSON object in the `description` field for later processing.

Suppose that a Splunk application comes with a KVStore collection called `example_ioc_indicators`, with the fields `key` and `description`. For long term supportability purposes you do not want to modify the collection, but simply want to utilize a custom lookup within a framework, such as Splunk Enterprise Security (ES) Threat Framework.

Let's start with the first part of the search:

```
| makeresults count=1 | eval threat="maliciousdomain.example", threat_expiry="2020-01-01 21:13:37 UTC", threat_name="Sample threat", threat_campaign="Sample threat", threat_confidence="100" | makejson threat_expiry, threat_name, threat_campaign, threat_confidence output=description | table threat, description
```

This search produces a result that looks something like this:

<code>threat</code>	<code>description</code>
maliciousdomain.example	{"threat_name": "Sample threat", "threat_confidence": 100, "threat_expiry": "2020-01-01 21:13:37 UTC", "threat_campaign": "Sample threat"}

You would then add the `outputlookup` command to send the search results to the lookup:

```
... | outputlookup append=t example_ioc_indicators
```

To use this custom lookup within a framework, you would specify this in a search:

```
... | lookup example_ioc_indicators OUTPUT description AS match_context | spath input=match_context
```

See also

Related commands
[spath](#)

mcatalog

The `mcatalog` command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

The `mcatalog` command performs aggregations on the values in the `metric_name` and `dimension` fields in the metric indexes.

Syntax

```
| mcatalog [prestats=<bool>] [append=<bool>] ( <values("<field>")> [AS <field>] )  
[WHERE <logical-expression>] [ (BY|GROUPBY) <field-list> ]
```

Required arguments

`values (<field>)`

Syntax: `values(<field>) [AS <field>]`

Description: Returns the list of all distinct values of the specified field as a multivalue entry. The order of the values is lexicographical. See [Usage](#).

Optional arguments

`append`

Syntax: `append=<bool>`

Description: Valid only when `prestats=true`. This argument runs the `mcatalog` command and adds the results to an existing set of results instead of generating new results.

Default: false

`<field-list>`

Syntax: `<field>, ...`

Description: Specify one or more fields to group results.

`<logical-expression>`

Syntax: `<time-opts>|<search-modifier>|((NOT)?`

`<logical-expression>)|<index-expression>|<comparison-expression>|(<logical-expression> (OR)?`

`<logical-expression>)`

Description: Includes time and search modifiers, comparison, and index expressions. Does not support CASE or TERM directives. You also cannot use the WHERE clause to search for terms or phrases.

`prestats`

Syntax: `prestats=true | false`

Description: Specifies whether to use the prestats format. The prestats format is a Splunk internal format that is designed to be consumed by commands that generate aggregate calculations. When using the prestats format you can pipe the data into the `chart`, `stats`, or `timechart` commands, which are designed to accept the prestats format. When `prestats=true`, AS instructions are not relevant. The field names for the aggregates are determined by the command that consumes the prestats format and produces the aggregate output.

Default: false

Logical expression options

<comparison-expression>

Syntax: <field><comparison-operator><value> | <field> IN (<value-list>)

Description: Compare a field to a literal value or provide a list of values that can appear in the field.

<index-expression>

Syntax: "<string>" | <term> | <search-modifier>

Description: Describe the events you want to retrieve from the index using literal strings and search modifiers.

<time-opts>

Syntax: [<timeformats>] (<time-modifier>)*

Description: Describe the format of the starttime and endtime terms of the search

Comparison expression options

<comparison-operator>

Syntax: = | != | < | <= | > | >=

Description: You can use comparison operators when searching field/value pairs. Comparison expressions with the `equal` (=) or `not equal` (!=) operator compare string values. For example, "1" does not match "1.0". Comparison expressions with greater than or less than operators < > <= >= numerically compare two numbers and lexicographically compare other values. See [Usage](#).

<field>

Syntax: <string>

Description: The name of a field.

<value>

Syntax: <literal-value>

Description: In comparison-expressions, the literal number or string value of a field.

<value-list>

Syntax: (<literal-value>, <literal-value>, ...)

Description: Used with the IN operator to specify two or more values. For example use `error IN (400, 402, 404, 406)` instead of `error=400 OR error=402 OR error=404 OR error=406`

Index expression options

<string>

Syntax: "<string>"

Description: Specify keywords or quoted phrases to match. When searching for strings and quoted strings (anything that's not a search modifier), Splunk software searches the `_raw` field for the matching events or results.

<search-modifier>

Syntax: <sourcetype-specifier> | <host-specifier> | <source-specifier> | <splunk_server-specifier>

Description: Search for events from specified fields. For example, search for one or a combination of hosts, sources, and source types. See searching with default fields in the *Knowledge Manager manual*.

<sourcetype-specifier>

Syntax: sourcetype=<string>

Description: Search for events from the specified sourcetype field.

<host-specifier>

Syntax: host=<string>

Description: Search for events from the specified host field.

<source-specifier>

Syntax: source=<string>

Description: Search for events from the specified source field.

<splunk_server-specifier>

Syntax: splunk_server=<string>

Description: Search for events from a specific server. Use "local" to refer to the search head.

Time options

For a list of time modifiers, see [Time modifiers for search](#).

<timeformat>

Syntax: timeformat=<string>

Description: Set the time format for starttime and endtime terms.

Default: timeformat=%m/%d/%Y:%H:%M:%S.

<time-modifier>

Syntax: starttime=<string> | endtime=<string> | earliest=<time_modifier> | latest=<time_modifier>

Description: Specify start and end times using relative or absolute time.

Note: You can also use the earliest and latest attributes to specify absolute and relative time ranges for your search. For more about this time modifier syntax, see [About search time ranges](#) in the *Search Manual*.

starttime

Syntax: starttime=<string>

Description: Events must be later or equal to this time. Must match `timeformat`.

endtime

Syntax: endtime=<string>

Description: All events must be earlier or equal to this time.

Usage

You use the `mcatalog` command to search metrics data. The metrics data uses a specific format for the metrics fields. See [Metrics data format](#) in *Metrics*. The `_values` field is not allowed with this command.

The `mcatalog` command is a **generating command** for reports. Generating commands use a leading pipe character. The `mcatalog` command must be the first command in a search pipeline, except when `append=true`.

If your role does not have the `list_metrics_catalog` capability, you cannot use `mcatalog`.

See About defining roles with capabilities in the *Securing Splunk Enterprise* manual.

WHERE

Use the WHERE clause to filter by supported dimensions.

If you do not specify an index name in the WHERE clause, the `mcatalog` command returns results from the default metrics indexes associated with your role. If you do not specify an index name and you have no default metrics indexes associated with your role, `mcatalog` returns no results. To search against all metrics indexes use `WHERE index=*`.

For more information about defining default metrics indexes for a role, see Add and edit roles with Splunk Web in *Securing Splunk Enterprise*.

Group by

You can group by `dimension` and `metric_name` fields.

The `mcatalog` command does not allow grouping by time ranges. The argument is not included in its syntax.

Time dimensions

The `mcatalog` command does not recognize the following time-related dimensions.

Unsupported dimensions		
date_hour	date_wday	timeendpos
date_mday	date_year	timestamp
date_minute	date_zone	timestampstartpos
date_month	metric_timestamp	
date_second	time	

Lexicographical order

Lexicographical order sorts items based on the values used to encode the items in computer memory. In Splunk software, this is almost always UTF-8 encoding, which is a superset of ASCII.

- Numbers are sorted before letters. Numbers are sorted based on the first digit. For example, the numbers 10, 9, 70, 100 are sorted lexicographically as 10, 100, 70, 9.
- Uppercase letters are sorted before lowercase letters.
- Symbols are not standard. Some symbols are sorted before numeric values. Other symbols are sorted before or after letters.

You can specify a custom sort order that overrides the lexicographical order. See the blog Order Up! Custom Sort Orders.

Examples

1. Return all of the metric names in a specific metric index

Return all of the metric names in the `new-metric-idx`.

```
| mcatalog values(metric_name) WHERE index=new-metric-idx
```

2. Return all metric names in the default metric indexes associated with the role of the user

If the user role has no default metric indexes assigned to it, the search returns no events.

```
| mcatalog values(metric_name)
```

3. Return all IP addresses for a specific metric_name among all metric indexes

Return of the IP addresses for the `login.failure` metric name.

```
| mcatalog values(ip) WHERE index=* AND metric_name=login.failure
```

4. Returns a list of all available dimensions in the default metric indexes associated with the role of the user

```
| mcatalog values(_dims)
```

noop

The `noop` command is an internal, unsupported, experimental command. See About internal commands.

Description

The `noop` command is an internal command that you can use to debug your search. It includes several arguments that you can use to troubleshoot search optimization issues.

You cannot use the `noop` command to add comments to a search. If you are looking for a way to add comments to your search, see Add comments to searches in the *Search Manual*.

Syntax

```
noop [<log-level-expression>] [<appender-expression>] [set_ttl = <timespan>] [search_optimization = <boolean>]  
[search_optimization.<optimization_type> = <boolean>] [sample_ratio = <int>] [<remote-log-fetch>] ...
```

Required arguments

None.

Optional arguments

appender-expression

Syntax: `log_appender = "<appender_name>; [<attribute_name> = <attribute_value>, ...]"`

Description Identifies an appender from `log-searchprocess.log` and specifies changed values for one or more attributes that belong to that appender. These value changes apply to the search job for the lifetime of the job. They are not reused after the search finishes. The list of attribute value changes must be enclosed in quotes. See [Appender expression options](#).

log-level-expression

Syntax: `log_<level> = "<channel>; ..."`

Description: Sets or changes the log levels for one or more log channels at search startup. The log channel list must be double-quoted and semicolon-separated. See [Log level expression options](#).

optimization_type

Syntax: `search_optimization.<optimization_type> = <boolean>`

Description: Enables or disables a specific type of search optimization for the search. To disable multiple optimization types, create a comma-separated list of `search_optimization.<optimization_type>` arguments. See [Optimization type arguments](#).

Default: true

remote-log-fetch

Syntax: `remote_log_fetch = [*|<indexer_name>;<indexer_name>...]`

Description: Downloads remote search logs from the specified list of indexers in order to troubleshoot searches. This argument overrides the `fetch_remote_search_log` setting in the `limits.conf` file, which is disabled by default for saved searches.

sample_ratio

Syntax: `sample_ratio = <int>`

Description: Sets a randomly-sampled subset of results to return from a given search. It returns 1 out of every `<sample_ratio>` events. For example, if you supply `| noop sample_ratio=25`, the Splunk software returns a random sample of 1 out of every 25 events from the search result set. The `sample_ratio` argument requires that `search` be the **generating command** of the search to which you are applying `noop`.

The `sample_ratio` does the same thing as the event sampling feature that you can manage through Splunk Web. The difference is that it enables you to apply event sampling to a subsearch, while the Splunk Web version of event sampling is applied only to the main search. See [Event sampling](#) in the *Search Manual*.

Default: 1

search_optimization

Syntax: `search_optimization = <boolean>`

Description: Enables or disables all optimizations for the search.

Default: true

set_ttl

Syntax: `set_ttl = <timespan>`

Description: Specifies the lifetime of the search job using time modifiers like `1d` for one day or `12h` for twelve hours. The search job lifetime is the amount of time that the job exists in the system before it is deleted. The default lifetime of an **ad hoc search** is 10 minutes. You might use this setting to make an ad hoc search job stay in the system for 24 hours or 7 days.

Optimization type arguments

Here are the `search_optimization.<optimization_type>` arguments that you can use with `noop`.

search_optimization argument	Controls
search_optimization.eval_merge	Eval merge optimization
search_optimization.merge_union	Merge union optimization
search_optimization.predicate_merge	Predicate merge optimizations
search_optimization.predicate_push	Predicate pushdown optimizations
search_optimization.predicate_split	Predicate split optimizations
search_optimization.projection_elimination	Projection elimination optimizations
search_optimization.required_field_values	Required field value optimizations
search_optimization.replace_append_with_union	Replace append command with union command optimization
search_optimization.replace_stats_cmds_with_tstats	Replace stats command with tstats command optimization This optimization type is disabled by default.
search_optimization.search_flip_normalization	Predicate flip normalization
search_optimization.search_sort_normalization	Predicate sort normalization

For more information about specific search optimization types, see [Built-in optimizations](#).

Log level expression options

level

Syntax: log_<level>

Description: Valid values are the Splunk platform internal logging levels: `debug`, `info`, `warn`, and `error`, and `fatal`. You can apply different log levels to different sets of channels.

channel

Syntax: <channel>; ...

Description: Specifies one or more log channels to apply the log level to. Use wildcards to catch all channels with a matching string of characters in their name. The list of log channels is semicolon-separated.

For example, `| noop log_debug="FastTyper;SearchParser"` runs `log_debug` on the `FastTyper` and `SearchParser` channels.

Appender expression options

appender_name

Syntax: <string>

Description: The name of an appender from the `log-searchprocess.cfg` file. Use a wildcard `*` to identify all appenders in the `log-searchprocess.cfg` file. The `noop` parser is case-sensitive. It sends an error message if you submit an appender name with incorrect case-formatting.

attribute_name

Syntax: maxFileSize | maxBackupIndex | ConversionPattern | maxMessageSize

Description: Attributes that can be changed for a given appender. The `noop` parser is case-sensitive, so do not change the case-formatting of these attributes. It sends an error message if you submit an appender name with incorrect case-formatting.

Attribute name	Description	Example value
----------------	-------------	---------------

maxFileSize	Sets the maximum size, in bytes, of a <code>search.log</code> file before it rolls. You must provide a value that is higher than the value that is currently set for the selected appender in the <code>log-searchprocess.cfg</code> file.	250000000
maxBackupIndex	Sets the maximum number of rolled <code>search.log</code> files. You must provide a value that is higher than the value that is currently set for the selected appender in the <code>log-searchprocess.cfg</code> file.	5
ConversionPattern	Specifies the log entry format. Possible variables are: %c (category), %d (date, followed by date variables in curly brackets), %m (log message), %n (newline), %p (priority - the log level), %r (relative time, msec), %R (relative time, sec), %t (thread time), and %T (thread ID).	%d{%-m-%d-%Y %H:%M:%S.%l} %-5p %c - %m%n
maxMessageSize	Sets the maximum size, in bytes, of messages sent by the log. Defaults to 16384. You must provide a value that is higher than the value that is currently set for the selected appender in the <code>log-searchprocess.cfg</code> file.	16384

attribute_value

Syntax: <string>

Description: Provides an updated value for the selected appender attribute. The values you provide for the `maxFileSize`, `maxBackupIndex`, and `maxMessageSize` attributes must be higher than the values that are currently set for those attributes in the `log-searchprocess.cfg` file. In other words, if the `maxFileSize` setting for the `searchprocessAppender` is currently set to `10000000`, you can only submit a new `maxFileSize` value that is higher than `10000000`.

Usage

You can use the `noop` command to enable or disable search optimizations when you run a search. Enabling or disabling search optimizations can help you troubleshoot certain kinds of search issues. For example, you might experiment with disabling and enabling search optimizations to determine whether they are causing a search to be slow to complete.

For information about managing search optimization through `limits.conf` for all of the users in your Splunk platform deployment, see Built-in optimization in the *Search Manual*.

Managing all search optimizations with the noop command

The `noop` command can enable or disable all search optimizations for a single run of a search.

If all search optimizations are enabled for your Splunk deployment in `limits.conf`, you can add the following argument to the end of a search string to disable all optimizations when you run that search:

```
.... | noop search_optimization=false
```

If all search optimizations are disabled for your Splunk deployment in `limits.conf`, you can add the following argument to the end of a search string to enable all search optimizations when you run that search:

```
.... | noop search_optimization=true
```

Managing specific search optimizations with the `noop` command

You can use the `optimization_type` argument to selectively disable or enable specific types of search optimization.

Here is an example of a set of `noop` arguments that disable the predicate merge and predicate pushdown optimizations for a search.

```
.... | noop search_optimization.predicate_merge=f, search_optimization.predicate_push=f
```

This example works only if you have enabled all optimizations in `limits.conf`.

When you set `enabled=false` for the `[search_optimization]` stanza in `limits.conf` you disable all search optimizations for your Splunk platform deployment. With this `limits.conf` configuration, your searches can use `noop` to enable all optimizations and selectively disable specific optimization types.

For example, if you have the `[search_optimization]` stanza set to `enabled = false` in `limits.conf`, the following search enables all optimizations except projection elimination.

```
index=_internal | eval c = x * y / z | stats count BY a, b | noop search_optimization=true,  
search_optimization.projection_elimination=false
```

However, When you set `enabled=false` for the `[search_optimization]` stanza in `limits.conf`, your searches cannot enable specific optimization types unless specific conditions are met. See [How `noop` interoperates with `limits.conf` search optimization settings](#).

How the `noop` command interoperates with `limits.conf` search optimization settings

Review how you have configured search optimization for your Splunk platform deployment in `limits.conf` before you use the `noop` command to enable or disable optimization types. The search processor respects `limits.conf` settings for optimization types only when `[search_optimization]` is enabled.

For example, if the `[search_optimization]` stanza is set to `enabled=true` in `limits.conf`, the search processor checks whether individual optimization types are enabled or disabled in `limits.conf`. On the other hand, if the `[search_optimization]` stanza is set to `enabled = false`, the search processor does not check the settings for other optimization types. It assumes all of the optimization types are set to `enabled=false`.

This search processor logic affects the way that the `noop` command works when you use it to enable or disable search optimization for an individual search.

For example, imagine that you have the following configuration in `limits.conf`:

```
[search_optimization]  
enabled=false
```

```
[search_optimization::projection_elimination]  
enabled=false
```

With this configuration, the search processor ignores the disabled projection elimination optimization. Because `[search_optimization]` is disabled, the search processor assumes all optimizations are disabled.

Say you have this configuration, and you run the following search, which uses the `noop` command to enable search optimization:

```
.... | noop search_optimization=true
```

When you do this, you enable search optimization, but the search processor sees that in `limits.conf`, the projection elimination optimization is disabled. It runs the search with all optimization types enabled except projection elimination.

Instead, use the `noop` command in a search to enable search optimization and selectively enable the projection elimination optimization:

```
.... | noop search_optimization=true search_optimization.projection_elimination=true
```

When this search runs, it overrides both `limits.conf` settings: the setting for `[search_optimization]` and the setting for `[search_optimization::projection_elimination]`. The search runs with all optimizations enabled.

Use noop to set debugging channels for a search

The `log_<level>` argument lets you set the debugging channel for a search at a specific log level, such as `debug` or `warn`. You might use this if you need to set the log level for a specific search but do not have CLI access to the Splunk platform implementation.

The Splunk platform changes the log level after it parses the `noop` command. It can do this before the search head parses arguments from other search commands, even if it comes after those commands in the search string. For instance, the following search properly logs some debug messages from the `makeresults` command despite the fact that it precedes the `noop` command:

```
| makeresults count=1 | noop log_debug="MakeResultsProcessor"
```

However, the `log_<level>` argument cannot set the log level for search process components that are ahead of SPL argument processing in the order of operations. For example, `LicenseMgr` is one of those components. When you run this search, it still logs at the default level of `info` for `LicenseMgr` even though you specify `debug` in the SPL.

```
index=_internal | head 1 | noop log_debug="LicenseMgr"
```

If you have command-line access and you need to debug an issue with that component or ones like it, you can modify `$SPLUNK_HOME/etc/log-searchprocess.cfg` directly to set the logging level before the search is dispatched and produce more verbose output in `search.log`.

The `noop` command must be part of the streaming pipeline. Because the Splunk software performs argument parsing on the search head and then pushes the search to the indexers, make sure that the `noop` command is part of the streaming pipeline. Place the `noop` command before the first non-streaming command in the search string. An easy way to do this is to put it after the first command in the search string, which is usually `search`.

The `log_<level>` argument supports wildcard matching. You can also set different log levels for different debugging channels in the same search.

```
.... | noop log_debug=Cache* log_info="SearchOperator:kv;SearchOperator:multikv"
```

For more information about logs and setting log levels for debugging channels, see What Splunk logs about itself in the *Troubleshooting Manual*.

Use `noop` to apply `log-searchprocess.cfg` appender attribute changes to a search job

For debugging purposes, you can use `noop` to apply changed attributes for `log-searchprocess.cfg` appenders to individual runs of a search. Appenders are blocks of configurations for specific sub-groups of log components. Example appenders include `searchprocessAppender`, `watchdog_appender`, and `searchTelemetryAppender`. You can use the `*` wildcard to select all appenders.

For example, the following search changes the maximum size of the `search.log` file to 50 MB and sets the maximum number of rolled `search.log` files to 99.

```
.... | noop log_appender="searchprocessAppender;maxFileSize=50000000;maxBackupIndex=99"
```

These changes are applied for the lifetime of that particular search. They are not saved or applied to other searches.

You can only change values for the following appender attributes: `maxFileSize`, `maxBackupIndex`, `ConversionPattern`, and `maxMessageSize`. Values you supply for `maxFileSize`, `maxBackupIndex`, and `maxMessageSize` must be higher than the current values for those appender attributes in `log-searchprocess.cfg`.

For more information about changing appender attributes for log debugging purposes, see [Enable log debugging](#) in the [Troubleshooting Manual](#).

prjob

The `prjob` command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

Use the `prjob` command for parallel reduce search processing of an SPL search in a distributed search environment. The `prjob` command analyzes the specified SPL search and attempts to reduce the search runtime by automatically placing a `redistribute` command in front of the first non-streaming SPL command like `stats` or `transaction` in the search. It provides the same functionality as the `redistribute` command, but with a simpler syntax. Similar to the `redistribute` command, use the `prjob` command to automatically speed up high cardinality searches that aggregate a large number of search results.

Syntax

```
prjob [<subsearch>]  
or  
prjob [num_ofReducers=<int>] [<subsearch>]
```

Required arguments

`subsearch`

Syntax: [<subsearch>]

Description: Specifies the search string that the `prjob` command attempts to process in parallel.

Optional arguments

`num_ofReducers`

Syntax: [num_ofReducers=<int>]

Description: Specifies the number of eligible indexers from the indexer pool that may function as intermediate reducers. For example: When a search is run on 10 indexers and the configuration is set to use 60% of the indexer pool (with a maximum value of 5), it implies that only five indexers may be used as intermediate reducers. If the value of `num_ofReducers` is set to greater than 5, only five reducers are available due to the limit. If the value of `num_ofReducers` is set to less than 5, the number of reducers used shrinks from the maximum limit of 5.

The value for `num_ofReducers` is controlled by two groups of settings:

- `reducers`:
- `maxReducersPerPhase + winningRate`

The number of intermediate reducers is determined by the value set for `reducers`. If no value is set for `reducers`, the search uses the values set for `maxReducersPerPhase` and `winningRate` to determine the number of intermediate reducers.

For example: In a scenario where Splunk is configured so that the value of `num_ofReducers` is set to 50 percent of the indexer pool and the `maxReducersPerPhase` value is set to four indexers, a parallel reduce search that runs on six search peers will be assigned to run on three intermediate reducers. Similarly, a parallel reduce search that runs on four search peers, will be assigned to run on two intermediate reducers. However, searches that runs on ten search peers would be limited to the maximum of four intermediate reducers.

Usage

Use the `prjob` command instead of the `redistribute` command when you want to run a parallel reduce job without determining where to insert the `redistribute` command or managing the `by-clause` field.

The `prjob` command may be used only as the first command of a search. Additionally, you must include the entire search within the `prjob` command.

To use the `prjob` command, set the `phased_execution_mode` to `multithreaded` or `auto` and set `enabled` to `true` in the `[search_optimization::pr_job_extractor]` stanza of the `limits.conf` configuration file.

The `prjob` command does not support real time or verbose mode searches. Real time or verbose mode searches with the `prjob` command may run, but the `redistribute` operation will be ignored. Also, you may not use the `prjob` and the `redistribute` command within the same search.

The `prjob` command supports the same commands as the `redistribute` command. For more information, see [redistribute](#). The `prjob` command only reduces the search runtime of an SPL search that contains at least one of the following non-streaming commands: ..."

- `stats`
- `tstats`
- `streamstats`
- `eventstats`
- `sistats`
- `sichart`
- `sitimechart`

- transaction (only on a single field)

Examples

Example 1: Using the `prjob` command in a search automatically places the `redistribute` command before the first non-streaming SPL command in the search. This speeds up a `stats` search that aggregates a large number of results. The `stats count by host` portion of the search is processed on the intermediate reducers and the search head aggregates the results.

Therefore, the following search:

```
| prjob [search index=myindex | stats count by host]
```

is transformed to:

```
search index=myindex | redistribute | stats count by host
```

Example 2: Speeds up a search that includes `eventstats` and uses `sitimechart` to perform the statistical calculations for a `timechart` operation. The intermediate reducers process `eventstats`, `where`, and `sitimechart` operations. The search head runs the `timechart` command to turn the reduced `sitimechart` statistics into sorted, visualization-ready results.

```
| prjob [search index=myindex | eventstats count by user, source | where count>10 | sitimechart max(count) by source | timechart max(count) by source]
```

Example 3: Speeds up a search that uses `tstats` to generate events. The `tstats` command must be placed at the start of the subsearch, and uses `prestats=t` to work with the `timechart` command. The `sitimechart` command is processed on the intermediate reducers and the `timechart` command is processed on the search head.

```
| prjob [search index=myindex | tstats prestats=t count by _time span=1d | sitimechart span=1d count | timechart span=1d count]
```

Example 4: The `eventstats` and `where` commands are processed in parallel on the reducers, while the `sort` command and any other following commands are processed on the search head. This happens because the `sort` command is a non-streaming command that is not supported by the `prjob` command.

The `prjob` command does not have an impact on this search.

```
| prjob [ search index=myindex | eventstats count by user, source | where count >10 | sort 0 -num(count) | ... ]
```

redistribute

The `redistribute` command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

The `redistribute` command implements **parallel reduce** search processing to shorten the search runtime of a set of supported SPL commands. Apply the `redistribute` command to high-cardinality dataset searches that aggregate large numbers of search results.

The `redistribute` command requires a **distributed search** environment where indexers have been configured to operate as **intermediate reducers**.

You can use the `redistribute` command only once in a search.

See Overview of parallel reduce search processing in Splunk Enterprise *Distributed Search*.

Syntax

```
redistribute [num_ofReducers=<int>] [<by-clause>]
```

Required arguments

None.

Optional arguments

`num_ofReducers`

Syntax: `num_ofReducers=<int>`

Description: Specifies the number of indexers in the indexer pool that are repurposed as intermediate reducers.

Default: The default value for `num_ofReducers` is controlled by three settings in the `limits.conf` file: `maxReducersPerPhase`, `winningRate`, and `reducers`. If these settings are not changed, by default the Splunk software sets `num_ofReducers` to 50 percent of your indexer pool, with a maximum of 20 indexers. See [Usage](#) for more information.

`by-clause`

Syntax: `BY <field-list>`

Description: The name of one or more fields to group by. You cannot use a wildcard character to specify multiple fields with similar names. You must specify each field separately. See [Using the by-clause](#) for more information.

Usage

In Splunk deployments that have distributed search, a two-phase map-reduce process is typically used to determine the final result set for the search. Search results are mapped at the indexer layer and then reduced at the search head.

The `redistribute` command inserts an intermediary reduce phase to the map-reduce process, making it a three-phase map-reduce-reduce process. This three-phase process is parallel reduce search processing.

In the intermediary reduce phase, a subset of the indexers become intermediate reducers. The intermediate reducers perform reduce operations for the search commands and then pass the results on to the search head, where the final result reduction and aggregation operations are performed. This parallelization of reduction work that otherwise would be done entirely by the search head can result in faster completion times for high-cardinality searches that aggregate large numbers of search results.

For information about managing parallel reduce processing at the indexer level, including configuring indexers to operate as intermediate reducers, see Overview of parallel reduce search processing, in the *Distributed Search* manual.

If you use Splunk Cloud Platform, use `redistribute` only when your indexers are operating with a low to medium average load. You do not need to perform any configuration tasks to use the `redistribute` command.

Supported commands

The `redistribute` command supports only **streaming commands** and the following nonstreaming commands:

- `stats`
- `tstats`
- `streamstats`
- `eventstats`
- `sichart`
- `sitimechart`

The `redistribute` command also supports the `transaction` command, when the `transaction` command is operating on only one field. For example, the `redistribute` command cannot support the `transaction` command when the following conditions are true:

- The `redistribute` command has multiple fields in its `<by-clause>` argument.
- The `transaction` command has multiple fields in its `<field-list>` argument.
- You use the `transaction` command in a mode where no field is specified.

For best performance, place `redistribute` immediately before the first supported nonstreaming command that has high-cardinality input.

When search processing moves to the search head

The `redistribute` command moves the processing of a search string from the intermediate reducers to the search head in the following circumstances:

- It encounters a nonstreaming command that it does not support.
- It encounters a command that it supports but that does not include a split-by field.
- It encounters a command that it supports and that includes split-by fields, but the split-by fields are not a superset of the fields that are specified in the `by-clause` argument of the `redistribute` command.
- It detects that a command modifies values of the fields specified in the `by-clause` of the `redistribute` command.

Using the by-clause to determine how results are partitioned on the reducers

At the start of the intermediate reduce phase, the `redistribute` command takes the mapped search results and redistributes them into partitions on the intermediate reducers according to the fields specified by the `by-clause` argument. If you do not specify any `by-clause` fields, the search processor uses the field or fields that work best with the commands that follow the `redistribute` command in the search string.

Command type

The `redistribute` command is an **orchestrating command**, which means that it controls how a search runs. It does not focus on the events processed by the search. The `redistribute` command instructs the distributed search query planner to convert centralized streaming data into distributed streaming data by distributing it across the intermediate reducers.

For more information about command types, see Types of commands in the *Search Manual*.

Setting the default number of intermediate reducers

The default value for the `num_ofReducers` argument is controlled by three settings in the `limits.conf` file: `maxReducersPerPhase`, `winningRate`, and `reducers`.

Setting name	Definition	Default value
maxReducersPerPhase	The maximum number of indexers that can be used as intermediate reducers in the intermediate reduce phase.	20
winningRate	The percentage of indexers that can be selected from the total pool of indexers and used as intermediate reducers in a parallel reduce search process. This setting applies only when the <code>reducers</code> setting is not configured.	50
reducers	A list of valid indexers that are to be used as dedicated intermediate reducers for parallel reduce search processing. When you run a search with the <code>redistribute</code> command, the valid indexers in the <code>reducers</code> list are the only indexers that are used for parallel reduce operations. If the number of valid indexers in the <code>reducers</code> list exceeds the <code>maxReducersPerPhase</code> value, the Splunk platform randomly selects a set of indexers from the <code>reducers</code> list that meets the <code>maxReducersPerPhase</code> limit.	" " (empty list)

If you provide a value for the `num_of_reducers` argument that exceeds the limit set by the `maxReducersPerPhase` setting, the Splunk platform sets the number of reducers to the `maxReducersPerPhase` value.

The redistribute command and search head data

Searches that use the `redistribute` command ignore all data on the search head. If you plan to use the `redistribute` command, the best practice is to forward all search head data to the indexer layer. See Best Practice: Forward search head data to the indexer layer in the *Distributed Search* manual.

Using the redistribute command in chart and timechart searches

If you want to add the `redistribute` command to a search that uses the `chart` or `timechart` commands to produce statistical results that can be used for chart visualizations, include either the `sichart` command or the `sitimechart` command in the search as well. The `redistribute` command uses these `si-` commands to perform the statistical calculations for the reporting commands on the intermediate reducers. When the `redistribute` command moves the results to the search head, the `chart` or `timechart` command transforms the results into a format that can be used for chart visualizations.

A best practice is to use the same syntax and values for both commands. For example, if you want to have `| timechart count by referrer_domain` in your `redistribute` search, insert `| sitimechart count by referrer_domain` into the search string:

```
index=main | redistribute | transaction referer_domain | search eventcount>500 | sitimechart count by referer_domain | search referer_domain=*.net | timechart count by referer_domain
```

If an order-sensitive command is present in the search

Certain commands that the `redistribute` command supports explicitly return results in a sorted order. As a result of the partitioning that takes place when the `redistribute` command is run, the Splunk platform loses the sorting order. If the Splunk platform detects that an order-sensitive command, such as `streamstats`, is used in a `redistribute` search, it automatically inserts `sort` into the search as it processes it.

For example, the following search includes the `streamstats` command, which is order-sensitive:

```
... | redistribute by host | stats count by host | streamstats count by host, source
```

The Splunk platform adds a `sort` segment before the `streamstats` segment when it processes the search. You can see the `sort` segment in the search string if you inspect the search job after you run it.

```
... | redistribute by host | stats count by host | sort 0 str(host) | streamstats count by host, source
```

The `stats` and `streamstats` segments are processed on the intermediate reducers because they both split by the `host` field, the same field that the `redistribute` command is distributing on. The work of the `sort` segment is split between the indexers during the map phase of the search and the search head during the final reduce phase of the search.

If you require sorted results from a redistribute search

If you require the results of a `redistribute` search to be sorted in that exact order, use `sort` to perform the sorting at the search head. There is an additional performance cost to event sorting after the `redistribute` command partitions events on the intermediate reducers.

The following search provides ordered results:

```
search * | stats count by foo
```

If you want to get that same event ordering while also adding `redistribute` to the search to speed it up, add `sort` to the search:

```
search * | redistribute | stats count by foo | sort 0 str(foo)
```

The `stats` segment of this search is processed on the intermediate reducers. The work of the `sort` segment is split between the indexers during the map phase of the search and the search head during the final reduce phase of the search.

Redistribute and virtual indexes

The `redistribute` command does not support searches of virtual indexes. The `redistribute` command also does not support unified searches if their time ranges are long enough that they run across virtual archive indexes. For more information, see the following *Splunk Analytics for Hadoop* topics:

- About virtual indexes
- Configure and run unified search

Override the number of intermediate reducers for a specific search

When you run a parallel reduce search with the `redistribute` command, you can use the `num_of_reducers` argument to override the number of reducers that are determined by the parallel reduce search settings in the `limits.conf` file.

For example, say your `limits.conf` settings determine that seven intermediate reducers are used by default in all parallel reduce searches. You can design a parallel reduce search where `num_ofReducers = 5`. Every time that search runs, only five intermediate reducers are used in its intermediate reduce phase.

If you provide a value for the `num_of_reducers` setting that exceeds the limit set by the `maxReducersPerPhase` setting in the `limits.conf` file, the Splunk platform sets the number of reducers to the `maxReducersPerPhase` value.

Examples

1. Speed up a search on a large high-cardinality dataset

In this example, the `redistribute` command is applied to a `stats` search that is running over an extremely large high-cardinality dataset. The `redistribute` command reduces the completion time for the search.

```
... | redistribute by ip | stats count by ip
```

The intermediate reducers process the `| stats count by ip` portion of the search in parallel, lowering the completion time for the search. The search head aggregates the results.

2. Speed up a timechart search without declaring a by-clause field to redistribute on

This example uses a search over an extremely large high-cardinality dataset. The search string includes the `eventstats` command, and it uses the `sitimechart` command to perform the statistical calculations for a `timechart` operation. The search uses the `redistribute` command to reduce the completion time for the search. A `by-clause` field is not specified, so the search processor selects one.

```
... | redistribute | eventstats count by user, source | where count>10 | sitimechart max(count) by source | timechart max(count) by source
```

When this search runs, the intermediate reducers process the `eventstats` and `sitimechart` segments of the search in parallel, reducing the overall completion time of the search. On the search head, the `timechart` command takes the reduced `sitimechart` calculations and transforms them into a format that can be used for charts and visualizations.

Because a `by-clause` field is not identified in the search string, the intermediate reducers redistribute and partition events on the `source` field.

3. Speed up a search that uses tstats to generate events

This example uses a search over an extremely large high-cardinality dataset. This search uses the `tstats` command in conjunction with the `sitimechart` and `timechart` commands. The `redistribute` command reduces the completion time for the search.

```
| tstats prestats=t count BY _time span=1d | redistribute by _time | sitimechart span=1d count | timechart span=1d count
```

You have to place the `tstats` command at the start of the search string with a leading pipe character. When you use the `redistribute` command in conjunction with `tstats`, you must place the `redistribute` command after the `tstats` segment of the search.

In this example, the `tstats` command uses the `prestats=t` argument to work with the `sitimechart` and `timechart` commands.

The `redistribute` command causes the intermediate reducers to process the `sitimechart` segment of the search in parallel, reducing the overall completion time for the search. The reducers then push the results to the search head, where the `timechart` command processes them into a format that you can use for charts and visualizations.

4. Speed up a search that includes a mix of supported and unsupported commands

This example uses a search over an extremely large high-cardinality dataset. The search uses the `redistribute` command to reduce the search completion time. The search includes commands that are both supported and unsupported by the `redistribute` command. It uses the `sort` command to sort of the results after the rest of the search has been processed. You need the `sort` command for event sorting because the `redistribute` process undoes the

sorting naturally provided by commands in the `stats` command family.

```
... | redistribute | eventstats count by user, source | where count >10 | sort 0 -num(count)
```

In this example, the intermediate reducers process the `eventstats` and `where` segments in parallel. Those portions of the search complete faster than they would when the `redistribute` command is not used.

The Splunk platform divides the work of processing the `sort` portion of the search between the indexer and the search head.

5. Speed up a search where a supported command splits by fields that are not in the redistribute command by-clause argument

In this example, the `redistribute` command redistributes events across the intermediate reducers by the `source` field. The search includes two commands that are supported by the `redistribute` command but only one of them is processed on the intermediate reducers.

```
... | redistribute by source | eventstats count by source, host | where count > 10 | stats count by userid, host
```

In this case, the `eventstats` segment of the search is processed in parallel by the intermediate reducers because it includes `source` as a split-by field. The `where` segment is also processed on the intermediate reducers.

The `stats` portion of the search, however, is processed on the search head because its split-by fields are not a superset of the set of fields that the events have been redistributed by. In other words, the `stats` split-by fields do not include `source`.

runshellscript

The `runshellscript` command is an internal, unsupported, experimental command. See [About internal commands](#).

Description

For Splunk Enterprise deployments, executes scripted alerts. This command is not supported as a search command.

This command is considered risky because, if used incorrectly, it can pose a security risk or potentially lose data when it runs. As a result, this command triggers SPL safeguards. See [SPL safeguards for risky commands](#) in [Securing the Splunk Platform](#).

Syntax

```
runshellscript <script-filename> <result-count> <search-terms> <search-string> <savedsearch-name> <description> <results-url> <deprecated-arg> <results_file> <search-ID> <results-file-path-deprecated-arg>
```

Usage

The script file needs to be located in either `$SPLUNK_HOME/etc/system/bin/scripts` OR `$SPLUNK_HOME/etc/apps/<app-name>/bin/scripts`. The following table describes the arguments passed to the script.

Argument	Description
\$0	The filename of the script.
\$1	The result count, or number of events returned.
\$2	The search terms.
\$3	The fully qualified search string.
\$4	The name of the saved search.
\$5	The description or trigger reason. For example, "The number of events was greater than 1."
\$6	The link to saved search results.
\$7	DEPRECATED - empty string argument.
\$8	The search ID.

The `runshellscript` command validates the \$8 search ID argument on

- Whether the provided search ID exists.
- Whether you have permission to access the provided search ID.

See also

[script](#)

Search in the CLI

About searches in the CLI

If you use Splunk Enterprise, you can issue search commands from the command line using the Splunk CLI. This topic discusses how to search from the CLI. If you're looking for how to access the CLI and find help for it, refer to "About the CLI" in the *Splunk Enterprise Admin Manual*.

CLI help for search

You can run historical searches using the `search` command, and real-time searches using the `rtsearch` command. The following is a table of useful search-related CLI help objects. To see the full help information for each object, type into the CLI:

```
./splunk help <object>
```

Object	Description
rtsearch	Returns the parameters and syntax for real-time searches.
search	Returns the parameters and syntax for historical searches.
search-commands	Returns a list of search commands that you can use from the CLI.
search-fields	Returns a list of default fields.
search-modifiers	Returns a list of search and time-based modifiers that you can use to narrow your search.

Search in the CLI

Historical and real-time searches in the CLI work the same way as searches in Splunk Web, except that there is no timeline rendered with the search results and there is no default time range. Instead, the results are displayed as a raw events list or a table, depending on the type of search.

- For more information, read "Type of searches" in the Search Overview chapter of the Search Manual.

The syntax for CLI searches is similar to the syntax for Splunk Web searches, except that you can pass parameters outside of the query to specify the time limit of the search, where to run the search, and how results are displayed.

- For more information about the CLI search options, see the next topic in this chapter, "[CLI search syntax](#)".
- For more information about how to search remote Splunk servers from your local server, see "Access and use the CLI on a remote server" in the *Splunk Enterprise Admin Manual*.

Syntax for searches in the CLI

If you use Splunk Enterprise, you can issue search commands from the command line using the Splunk CLI. This is a quick discussion of the syntax and options available for using the `search` and `rtsearch` commands in the CLI.

The syntax for CLI searches is similar to the syntax for searches you run from Splunk Web except that you can pass parameters outside of the search object to control the time limit of the search, specify the server where the search is to be

run, and specify how results are displayed.

```
search | rtsearch [object] [-parameter <value>]
```

Search defaults

By default when you run a search from the CLI, the search uses All Time as the time range. You can specify time ranges using one of the CLI search parameters, such as `earliest_time`, `index_earliest`, or `latest_time`.

The first 100 events are returned when you run a historical search using the CLI. Use the `maxout` search parameter to specify the number of events to return.

Search objects

Search objects are enclosed in single quotes (' ') and can be keywords, expressions, or a series of search commands. On Windows OS use double quotes (" ") to enclose your search object.

- For more information about searching, see Start searching in the *Search Tutorial*.
- For a brief description of every search command, see the [Command quick reference](#) in the *Search Reference*.
- For a quick reference for Splunk concepts, features, search commands, and functions, see the [Quick Reference Guide](#) in the *Search Reference*.

Search objects can include not only keywords and search commands but also fields and modifiers to specify the events you want to retrieve and the results you want to generate.

- For more information about fields, see Use fields to search in the *Search Tutorial*.
- For more information about default fields and how to use them, see Use default and internal fields in the *Knowledge Manager Manual*.
- For more information about time modifiers, see [Time modifiers for search](#) in the *Search Reference*.

Search parameters

Search parameters are options that control the way the search is run or the way the search results are displayed. All of these parameters are optional. Parameters that take Boolean values support `0`, `false`, `f`, `no` as negatives and `1`, `true`, `t`, `yes` as positives.

Specify these search parameters at the end of your search, after you have specified all of the commands and command arguments. See [Example 4](#).

Parameter	Values	Defaults	Description
app	<app_name>	search	Specify the name of the app in which to run your search.
batch	<bool>	F	Indicates how to handle updates in preview mode.
detach	<bool>	F	Triggers an asynchronous search and displays the job ID and TTL for the search.
earliest_time	<time-modifier>	–	The relative time modifier for the start time of the search. This is optional for <code>search</code> and required for <code>rtsearch</code> .
header	<bool>	T	

Parameter	Values	Defaults	Description
			Indicates whether to display a header in the table output mode.
index_earliest	<time-modifer>		The start time of the search. This can be expressed as an epoch or relative time modifier and uses the same syntax as the "earliest" and "latest" time modifiers for search language. This is optional for both <code>search</code> and <code>rtsearch</code> .
index_latest	<time-modifer>		The end time of the search. This can be expressed as an epoch or relative time modifier and uses the same syntax as the "earliest" and "latest" time modifiers for search language. This is optional for both <code>search</code> and <code>rtsearch</code> .
latest_time	<time-modifer>	–	The relative time modifier for the end time of search. For <code>search</code> , if this is not specified, it defaults to the end of the time (or the time of the last event in the data), so that any "future" events are also included. For <code>rtsearch</code>, this is a required parameter and the real-time search will not run if it's not specified.
max_time	<number>	0	The length of time in seconds that a search job runs before it is finalized. A value of 0 means that there is no time limit.
maxout	<number>	search, 100 rtsearch, 0	The maximum number of events to return or send to <code>stdout</code> when exporting events. A value of 0 means that it will output an unlimited number of events.
output	rawdata, table, csv, auto	Use <code>rawdata</code> for non-transforming searches. Use <code>table</code> for transforming searches.	Indicates how to display the job.
preview	<bool>	T	Indicates that reporting searches should be previewed (displayed as results are calculated).
timeout	<number>	0	The length of time in seconds that a search job is allowed to live after running. A value of 0 means that the job is canceled immediately after it is run.
uri	[http https]://name_of_server:management_port		<p>Specify the server name and management port. <code>name_of_server</code> can be the fully-resolved domain name or the IP address of the Splunk server.</p> <p>The default <code>uri</code> value is the <code>mgmtHostPort</code> value that you defined in the Splunk server's <code>web.conf</code>.</p> <p>For more information, see Access and use the CLI on a remote Splunk Server in the <i>Admin manual</i>.</p>

Parameter	Values	Defaults	Description
wrap	<bool>	T	Indicates whether to line wrap for individual lines that are longer than the terminal width.

Examples

You can see more examples in the CLI help information.

1. Retrieve events from yesterday that match root sessions

```
./splunk search "session root daysago=1"
```

2. Retrieve events that match web access errors and detach the search

```
./splunk search 'eventtype=webaccess error' -detach true
```

3. Run a windowed real-time search

```
./splunk rtsearch 'index=_internal' -earliest_time 'rt-30s' -latest_time 'rt+30s'
```

See more examples of Real-time searches and reports in the CLI in the *Admin Manual*.

4. Return a list of unique hostnames

There are two recommended ways that you can do this. This first is with the `stats` command:

```
./splunk search 'index=' | stats count by host | fields -count' -preview true
```

Alternatively, since you are only interested in the host field, you can use the `metadata` command:

```
./splunk search '| metadata type=hosts | fields host' -preview true
```

Here, the `-preview` flag is optional and used to view the results as it is returned. In contrast, the `table` command, unlike the `fields` command, generally requires all inputs before it can emit any non-preview output. In this case, you would need to use the `preview` flag to be able to view the results of the search.

5. Return yesterday's internal events

```
./splunk search 'index=_internal' -index_earliest -1d@d -index_latest @d
```