Splunk Search 201

Lab Guide

Overview

The purpose of this workshop is to enable your teams to search, investigate, analyze, and report on data in the Splunk platform in order to increase your teams' productivity and efficiency in identifying and resolving issues.

Prerequisites

In order to complete these exercises, you will need your own Splunk instance. Splunk's hands-on workshops are delivered via the Splunk Show portal and you will need a Splunk.com account in order to access this.

If you don't already have a Splunk.com account, please create one here before proceeding with the rest of the workshop.

⚠️ Troubleshooting Connectivity

If you experience connectivity issues with accessing either your workshop environment or the event page, please try the following troubleshooting steps. If you still experience issues please reach out to the team running your workshop.

- **Use Google Chrome** (if you're not already)
- If the event page (i.e. *https://show.splunk.com/event/<eventID>*) didn't load when you clicked on the link, try **refreshing the page**
- **Disconnect from VPN** (if you're using one)
- **Clear your browser cache and restart your browser** (if using Google Chrome, go to: Settings > Privacy and security > Clear browsing data)
- **Try using private browsing mode** (e.g. Incognito in Google Chrome) to rule out any cache issues
- **Try using another computer** such as your personal computer - all you need is a web browser! Cloud platforms like AWS can often be blocked on corporate laptops.

Table of Contents

Lab 1 – Building transactions with the `stats` `list()/values()` functions

Goal: Write an SPL query to group web log events into transactions based on their session ID. Exclude the address and category_id fields from the displayed results, and only show the 10 most recent transactions, sorted in descending order based on their duration.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar.
2. Start by exploring the available data in your environment! Under the “How to search” section, click on **Data Summary:**
   a. You will see that there are only 4 source types available in our lab environment.
3. **Write an SPL search** to show the Web log events, excluding the address and category_id fields from the results displayed:
   a. Looking at the available source types in the previous step, we can easily identify the relevant one to use in this query:
      i. For Web log events, we will look at `sourcetype=access_combined`
   b. To exclude the indicated fields, use the `fields` command:

   ```
   | fields - address, category_id
   ```
4. Now, **use the `stats` command** with the appropriate functions **to group the events into transactions:**
   a. To calculate the duration of a single transaction, we will use the `stats` function `range()`:
      i. This function will calculate the time between the first and last events for a specific value of the field specified in parenthesis:

      ```
      | stats range(_time) as duration
      ```
   b. Next, we will use the `values()` and `list()` functions to display the distinct values of the action field seen in a single transaction, as well as a list of all the raw events themselves:

   ```
   | stats range(_time) as duration, values(action) as actions, list(_raw)
   as _raw
   ```
   c. Lastly, we will use the “group by” clause to determine the events that should be part of the same transaction, which in this case should be the session ID:

   ```
   | stats range(_time) as duration, values(action) as actions, list(_raw)
   as _raw by JSESSIONID
   ```

5. Now that we have the events grouped into transactions, we sort the results in decreasing order based on the duration field and only show the 10 most recent transactions:

   a. To sort the events in descending order, we use the `sort` command:

   ```
   | sort -duration
   ```

   b. To show only the 10 most recent transactions, we use the `head` command:

   ```
   | head 10
   ```

Solution

```
sourcetype=access_combined
| fields - address, category_id
| stats range(_time) as duration
  values(action) as actions
  list(_raw) as _raw by JSESSIONID
| sort -duration
| head 10
```

Lab 2 – Building transactions with the `transaction` command

Goal: Write an SPL query to group web log events into transactions based on their session ID and username. Only show the 10 most recent transactions, sorted in descending order based on their duration. Display the results in a table, and include *at least* the following fields: session ID, duration, username, and uri path.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.

2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`

3. To group events into transactions by their session ID and username, we will use the transaction command:

   ```
   | transaction JSESSIONID username
   ```

4. To sort the events in descending order, we use the `sort` command:

   ```
   | sort -duration
   ```

5. To show only the 10 most recent transactions, we use the `head` command:

   ```
   | head 10
   ```

6. Lastly, to show our results in a table and include the fields specified, we can use the `table` command:

   ```
   | table JSESSIONID duration username uri_path
   ```

Solution

```
sourcetype=access_combined
| transaction JSESSIONID username
| sort -duration
| head 10
| table JSESSIONID duration username uri_path
```

Lab 3 – The chart command

Goal: Write an SPL query to determine how the duration of web user sessions varies, and for each duration interval, show how many distinct users were seen.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.
2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`
3. The first thing we want to do is group events into transactions based on the session ID and the username in each event:
   a. We can use the `stats` command or the `transaction` command to do this.
   b. Since `stats` is more efficient, we will use this one:

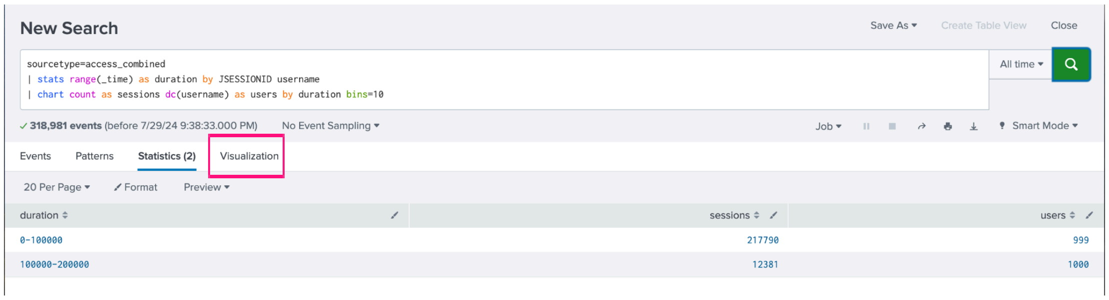
   ```
   | stats range(_time) as duration by JSESSIONID username
   ```
4. Now, we use the chart command to organize the results as needed:
   a. We will group the user session transactions by their duration, using a maximum of 10 intervals (`bin` option):

   ```
   | chart <functions to be used here> by duration bins=10
   ```
   b. For each duration interval, we calculate the number of user sessions using the `count()` function:

   ```
   | chart count as sessions by duration bins=10
   ```
   c. To display the distinct count of usernames for each duration interval, we will use the `dc()` function:

   ```
   | chart count as sessions dc(username) as users by duration bins=10
   ```

5. Once we get the results we are looking for, we can visualize them in a **Column Chart**:
   a. In the results window, switch from the **Statistics** tab to the **Visualization** tab:



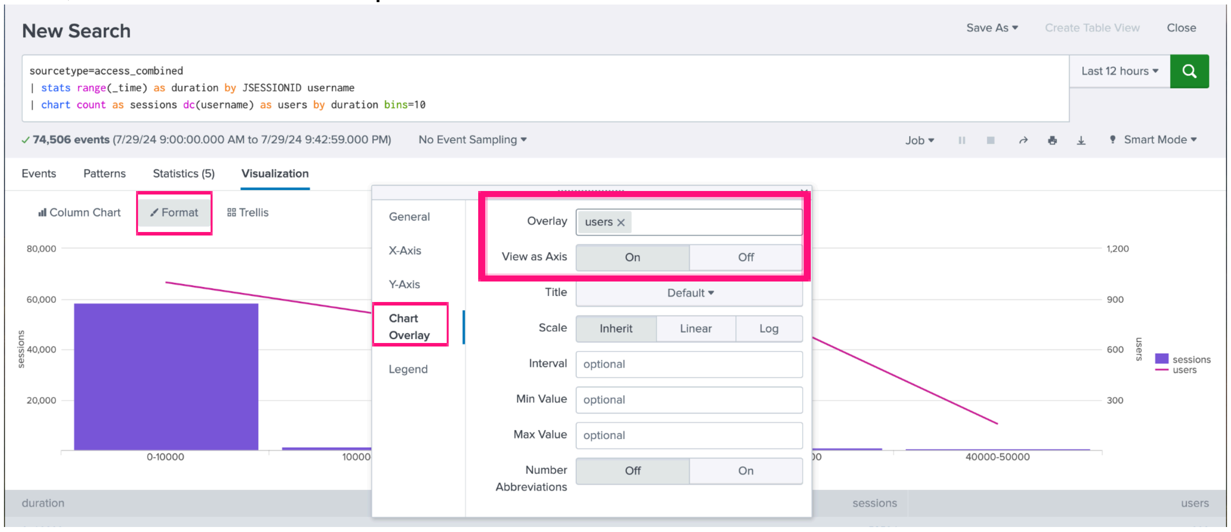
New Search

```
sourcetype=access_combined
| stats range(_time) as duration by JSESSIONID username
| chart count as sessions dc(username) as users by duration bins=10
```

✓ 318,981 events (before 7/29/24 9:38:33.000 PM) No Event Sampling

duration	sessions	users
0-100000	217790	999
100000-200000	12381	1000

   b. Select **Column Chart** from the visualization options.
   c. Click on **Format** and navigate to the **Chart Overlay** settings. Select **users** as the overlay field, and make sure the option to **View as Axis** is set to **On**:



New Search

```
sourcetype=access_combined
| stats range(_time) as duration by JSESSIONID username
| chart count as sessions dc(username) as users by duration bins=10
```

✓ 74,506 events (7/29/24 9:00:00.000 AM to 7/29/24 9:42:59.000 PM) No Event Sampling

Overlay: users
View as Axis: [✓ On] Off
Title: Default
Scale: [✓ Inherit] Linear Log
Interval: optional
Min Value: optional
Max Value: optional
Number Abbreviations: [✓ Off] On

Solution

```
sourcetype=access_combined
| stats range(_time) as duration by JSESSIONID username
| chart count as sessions dc(username) as users by duration bins=10
```

Lab 4 – The `stats` `stdev()` function

Goal: Write an SPL query to track the average duration of user sessions, using averages and medians, as well as the typical fluctuation of these values, using standard deviations and percentiles.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.
2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`
3. The first thing we want to do is group events into transactions based on the session ID and the username in each event:
    a. We can use the `stats` command or the `transaction` command to do this.
    b. Since `stats` is more efficient, we will use this one:
    ```
    | stats earliest(_time) as _time range(_time) as duration by JSESSIONID
    username
    ```
4. Now, to track the average duration and typical fluctuation over time, we can use the `timechart` command with the relevant functions:
    a. To calculate the average duration of user sessions, using both averages and medians:
    ```
    | timechart avg(duration) median(duration)
    ```
    b. To calculate the typical fluctuation of the above values, using standard deviations and percentiles:
    ```
    | timechart stdev(duration) perc25(duration) perc75(duration)
    ```
    c. Now, if we combine the two commands above and reorganize them, we get:
    ```
    | timechart avg(duration) stdev(duration) perc25(duration)
    median(duration) perc75(duration)
    ```

Solution
```
sourcetype=access_combined
| stats earliest(_time) as _time range(_time) as duration by JSESSIONID username
| timechart avg(duration) stdev(duration) perc25(duration) median(duration)
perc75(duration)
```

Lab 5 – The `eventstats` command

Goal: Write an SPL query to identify transactions with unusually slow durations, which may potentially indicate service health issues.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.
2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`
3. To find what we are looking for, we first need to group our events into transactions by the username and the service uri path:

   ```
   | stats earliest(_time) as _time range(_time) as duration by JSESSIONID
   uri_path
   ```

4. We then use the `eventstats` command to calculate the average duration and standard deviation per service, and add this information to our transaction events:

   ```
   | eventstats avg(duration) as average_duration stdev(duration) as
   stdev_duration by uri_path
   ```

5. Lastly, we filter our results to only show those transactions where the duration exceeded the average duration by more than 2 standard deviations:

   ```
   | where duration > (average_duration + 2*stdev_duration)
   ```

Solution

```
sourcetype=access_combined
| stats earliest(_time) as _time range(_time) as duration by JSESSIONID uri_path
| eventstats avg(duration) as average_duration stdev(duration) as stdev_duration
by uri_path
| where duration > (average_duration + 2*stdev_duration)
```

Lab 6 – The `streamstats` command

Goal: Write an SPL query to continuously monitor transactions and identify those with unusually slow durations compared to the recent transaction history.

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.
2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`
3. To find what we are looking for, we first need to group our events into transactions by the username and the service uri path:

   ```
   | stats earliest(_time) as _time range(_time) as duration by JSESSIONID
   uri_path
   ```

4. We then use the `streamstats` command to calculate the rolling average duration and rolling standard deviation per service:

   ```
   | streamstats avg(duration) as rolling_avg_duration stdev(duration) as
   rolling_stdev_duration by uri_path
   ```

5. Lastly, we filter our results to only show those transactions where the duration exceeded the (historical) average duration by more than 2 (historical) standard deviations:

   ```
   | where duration > (rolling_avg_duration + 2*rolling_stdev_duration)
   ```

Solution

```
sourcetype=access_combined
| stats earliest(_time) as _time range(_time) as duration by JSESSIONID uri_path
| streamstats avg(duration) as rolling_avg_duration stdev(duration) as
rolling_stdev_duration by uri_path
| where duration > (rolling_avg_duration + 2*rolling_stdev_duration)
```

Lab 7 – Putting everything to practice!

Goal: **Write an SPL query that generates a visualization to help identify potential bots in our web logs.**

1. Navigate to the **Search & Reporting** app and click on the **Search** tab in the menu bar if you're not already there.

2. Since we want to focus on Web log events, we will use: `sourcetype=access_combined`

3. We will now start by grouping events in 1-minute intervals using the bin command:

   ```
   | bin _time span=1m
   ```

4. Next, we can calculate the number of events within a single minute for each username as the clicks per minute:

   ```
   | stats count as clicks_per_minute by _time username
   ```

5. Using eventstats allows us to easily calculate the average number of clicks per minute and its standard deviation for each user and add this data to the events:

   ```
   | eventstats avg(clicks_per_minute) as avg_clicks stdev(clicks_per_minute) as
   dev_clicks by username
   ```

6. We can now easily identify potentially problematic users by filtering the results to only include those where the average number of clicks per minute is greater than 1, and sorting them in ascending order based on their standard deviation:

   a. To filter the results, we can use:

      ```
      | where avg_clicks > 1
      ```

b. **(Optional)** If we want to group the results by the username for easier visualization:

   i. Use the eval command to ensure all timestamps listed for a given username are in human-readable format:

```
| eval _time=strftime(_time, "%Y-%m-%d %H:%M:%S")
```

   ii. Use the chart command to display the relevant data and group by username:

```
| chart list(_time) as _time list(clicks_per_minute) as user_clicks
values(avg_clicks) as avg_clicks values(dev_clicks) as dev_clicks
by username
```

c. To sort the results in ascending order based on the standard deviation:

```
| sort + dev_clicks
```

7. **(Optional)** For those who want to take things a step further, we can create a visualization to easily identify the times when a potential bot was seen in our web log data:

   a. Repeat steps 1-4 above:

```
sourcetype=access_combined
| bin _time span=1m
| stats count as clicks_per_minute by _time username
```

   b. Using eventstats allows us to easily calculate the average number of clicks per minute and its standard deviation for each time interval and add this data to the events:

```
| eventstats avg(clicks_per_minute) as avg_clicks
stdev(clicks_per_minute) as dev_clicks by _time
```

   c. To determine the min and max threshold for the clicks per minute field, we can use eval:

      i. Minimum threshold:

```
| eval lower=avg_clicks-dev_clicks
```

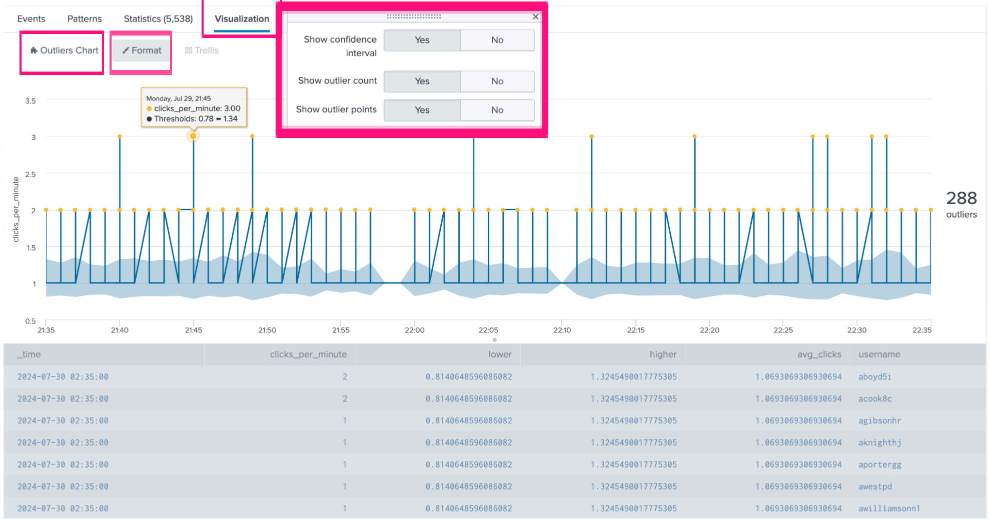
      ii. Maximum threshold:

```
| eval higher=avg_clicks+dev_clicks
```

   d. If we use the table command, we can create an outliers chart that displays the clicks per minute values over time and highlights those that fall outside of the acceptable range (from min and max thresholds calculated above):

```
| table _time, clicks_per_minute, lower, higher, avg_clicks, username
```

e. Once we get the results we are looking for, we can visualize them in an **Outliers Chart**:
   i. In the results window, switch from the **Statistics** tab to the **Visualization** tab.
   ii. Select **Outliers Chart** from the visualization options.
   iii. Click on **Format** and set all options to **Yes:**



Show confidence interval: [✓ Yes] No
Show outlier count: [✓ Yes] No
Show outlier points: [✓ Yes] No

_time	clicks_per_minute	lower	higher	avg_clicks	username
2024-07-30 02:35:00	2	0.8140648596086082	1.3245490017775305	1.0693069306930694	aboyd5i
2024-07-30 02:35:00	2	0.8140648596086082	1.3245490017775305	1.0693069306930694	acook8c
2024-07-30 02:35:00	1	0.8140648596086082	1.3245490017775305	1.0693069306930694	agibsonhr
2024-07-30 02:35:00	1	0.8140648596086082	1.3245490017775305	1.0693069306930694	aknighthj
2024-07-30 02:35:00	1	0.8140648596086082	1.3245490017775305	1.0693069306930694	aportergg
2024-07-30 02:35:00	1	0.8140648596086082	1.3245490017775305	1.0693069306930694	awestpd
2024-07-30 02:35:00	1	0.8140648596086082	1.3245490017775305	1.0693069306930694	awilliamsonn1

Solution

~~~
sourcetype=access_combined
| bin _time span=1m
| stats count as clicks_per_minute by _time username
| eventstats avg(clicks_per_minute) as avg_clicks stdev(clicks_per_minute) as
dev_clicks by username
| where avg_clicks > 1

``` Optional commands to format results (6b)
| eval _time=strftime(_time, "%Y-%m-%d %H:%M:%S")
| chart list(_time) as _time list(clicks_per_minute) as user_clicks
values(avg_clicks) as avg_clicks values(dev_clicks) as dev_clicks by username
```

| sort + dev_clicks
~~~

Alt. Solution

```
sourcetype=access_combined
| bin _time span=1m
| stats count as clicks_per_minute by _time username
| eventstats avg(clicks_per_minute) as avg_clicks stdev(clicks_per_minute) as
dev_clicks by _time
| eval lower=avg_clicks-dev_clicks
| eval lower=avg_clicks+dev_clicks
| table _time, clicks_per_minute, lower, higher, avg_clicks, username
```