LogQL in Grafana Loki

Sean Bradley

Published in Grafana Tutorials
4 min read · Sep 1, 2020

Now that we have a **Loki** data source we can query it with the **LogQL** query language.

In this video, we will try out many LogQL queries on the Loki data source we've setup.

There are two types of LogQL queries:

- Log queries returning the contents of log lines as streams.
- Metric queries that convert logs into value matrixes.

A LogQL query consists of,

- The log stream selector
- Filter expression

We can use operations on both the log stream selectors and filter expressions to refine them.

Log Stream Selectors

Operators

- = : equals
- != : not equals
- =~ : regex matches
- !~ : regex does not match

EXAMPLES

Return all log lines for the job `varlog`

```
{job="varlogs"}
```

Return all log lines for the filename `/var/log/syslog`

```
{filename="/var/log/syslog"}
```

Return all log lines for the job `varlogs` and the filename `/var/log/auth.log`

```
{filename="/var/log/auth.log",job="varlogs"}
```

Show all log lines for 2 jobs with different names

```
{filename=~"/var/log/auth.log|/var/log/syslog"}
```

Show everything you have using regex

```
{filename=~".+"}
```

Show data from all filenames, except for syslog

```
{filename=~".+",filename!="/var/log/syslog"}
```

Show

Filter Expressions

Operators

Used for testing text within log line streams.

- |= : equals
- != : not equals

- |~ : regex matches
- !~ : regex does not match

Examples

Return lines including the text “error”

```
{job="varlogs"} |= "error"
```

Return lines **not** including the text “error”

```
{job="varlogs"} != "error"
```

Return lines including the text “error” or “info” using regex

```
{job="varlogs"} |~ "error|info"
```

Return lines **not** including the text “error” or “info” using regex

```
{job="varlogs"} !~ "error|info"
```

Return lines including the text “error” but **not** including “info”

```
{job="varlogs"} |= "error" != "info"
```

Return lines including the text “Invalid user” and including (“bob” or “redis”) using regex

```
{job="varlogs"} |~ "Invalid user (bob|redis)"
```

Return lines including the text “status 403” or “status 503” using regex

```
{job="varlogs"} |~ "status [45]03"
```

Scalar Vectors and Series of Scalar Vectors

The data so far is returned as streams of log lines. We can graph these in visualizations if we convert them to scalar vectors or even multiple series of scalar vectors.

We can aggregate the lines into numeric values, such as counts, which then become known as a scalar vectors. The functions below will auto create new scalar vectors based on the labels present in a log stream.

- count_over_time : Shows the total count of log lines for time range
- rate : Similar as *count_over_time* but converted to number of entries per second
- bytes_over_time : Number of bytes in each log stream in the range
- bytes_rate : Similar to *bytes_over_time* but converted to number of bytes per second

Examples

The count of jobs at 1 minutes time intervals

```
count_over_time({job="varlogs"}[1m])
```

The rate of logs per minute. Rate is similar to `count_over_time` but shows the entries per second.

```
rate({job="varlogs"}[1m])
```

The count of `errors` at 1h time intervals

```
count_over_time({job="varlogs"} |= "error" [1h])
```

Aggregate Functions

An aggregate function converts multiple series of vectors into a single vector.

- sum : Calculate the total of all vectors in the range at time
- min : Show the minimum value from all vectors in the range at time
- max : Show the maximum value from all vectors in the range at time
- avg : Calculate the average of the values from all vectors in the range at time
- stddev : Calculate the standard deviation of the values from all vectors in the range at time
- stdvar : Calculate the standard variance of the values from all vectors in the range at time
- count : Count the number of elements all all vectors in the range at time
- bottomk : Select lowest k values in all the vectors in the range at time
- topk : Select highest k values in all the vectors in the range at time

Examples

Calculate the total of all vectors in the range at time

```
sum(count_over_time({job="varlogs"}[1m]))
```

Show the minimum value from all vectors in the range at time

```
min(count_over_time({job="varlogs"}[1m]))
```

Show the maximum value from all vectors in the range at time

```
max(count_over_time({job="varlogs"}[1m]))
```

Show only the top 2 values from all vectors in the range at time

```
topk(2, count_over_time({job="varlogs"}[1h]))
```

Aggregate Group

Convert a scalar vector into a series of vectors grouped by filename

Examples

Group a single log stream by filename

```
sum(count_over_time({job="varlogs"}[1m])) by (filename)
```

Group multiple log streams by host

```
sum(count_over_time({job=~"varlogs"}[1m])) by (host)
```

Group multiple log streams by filename and host

```
sum(count_over_time({job=~"varlogs"}[1m])) by (filename,host)
```

Comparison Operators

Comparison Operators. Used for testing numeric values present in scalars and vectors.

- == (equality)
- != (inequality)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

Examples

Returns values greater than 4

```
sum(count_over_time({job="varlogs"}[1m])) > 4
```

Returns values less than or equal to 1

```
sum(count_over_time({job="varlogs"}[1m])) <= 1
```

Logical Operators

These can be applied to both vectors and series of vectors

- and : Bother sides must be true
- or : Either side must be true
- unless : Return values unless value

Examples

Returns values greater than 4 or values less then or equal to 1

```
sum(count_over_time({job="varlogs"}[1m])) > 4 or
sum(count_over_time({job="varlogs"}[1m])) <= 1
```

Return values between 100 and 200

```
sum(count_over_time({job="varlogs"}[1m])) > 100 and
sum(count_over_time({job="varlogs"}[1m])) < 200
```

Arithmetic Operators

- + : Add
- - : Subtract
- * : Multiply
- / : Divide
- % : Modulus
- ^ : Power/Exponentiation

Examples

```
sum(count_over_time({job="varlogs"}[1m])) * 10
sum(count_over_time({job="varlogs"}[1m])) % 2
```

Operator order

Many Operators can be used at a time. The order follows the PEMDAS construct. PEMDAS is an acronym for the words parenthesis, exponents, multiplication, division, addition, subtraction.

Examples

A nonsensical example

```
sum(count_over_time({job="varlogs"}[1m])) % 4 * 2 ^ 2 + 2
# is the same as
((sum(count_over_time({job="varlogs"}[1m])) % 4 * (2 ^ 2)) + 2)
```

Proving that count_over_time / 60 / range(m) = rate

```
rate({job="varlogs"}[2m]) == count_over_time({job="varlogs"}[2m]) /
60 / 2
# is the same as
rate({job="varlogs"}[2m]) == ((count_over_time({job="varlogs"}[2m])
/ 60) / 2)
```

Originally published at https://sbcode.net/grafana/logql/

Thanks for reading my article, always remember to Clap, Comment and Share and I will write more.

Sean

Logql | Loki | Grafana | Grafana Engineer