LogQL: Prometheus-inspired Log Query Language

David Kaltschmidt, Goutham Veeramachaneni, Tom Wilkie June 2018
Status: Draft

Motivation

The ultimate goal for the log query language is to feel intuitive and quick to write the main use cases. Intuition derives from the mental models that users have already formed around log interaction. Here is customer feedback:

- "Just give me distributed grep"
- "I use `kubectl logs` and then a chain of greps"
- "I add negative filters as much as possible and then manually scan the rest"
- "None of the current UI tools can give me context like `grep -C n` can" [Tweet] and separate Issue

Axioms

- We want to be *the* distributed grep, hence it should be easy to translate the following to a query: <streams> | grep foo | egrep 'x42|x13' | grep -v baz
- We want to keep as much Prometheus syntax and functions as possible; it should feel native and behave as expected to a Prometheus user
- We are not trying to solve all use cases and build a ElasticSearch competitor; simplicity is our main aim here.
- In particular, needle-in-a-haystack style problem, such as "Show me the N users with the highest latency", are out of scope.
- Should be easy to build an autocomplete UI around

Design

Querying can be broken down into parts: Select, Filter, Transform. The result of each step becomes the input to the next.

Select

- Selecting streams is "pre-grep", hence we can rely on non-grep paradigms
- Prometheus has a well established selector logic with labels
- Using Prometheus selectors has the added benefit of selector-reuse when switching from Prometheus to logs
- No filtering (other than on labels) or transformation happens in this phase

Filter

- Filtering should be the grep phase and we can and should use grep paradigms
- Core grep: include/exclude filter chain (`grep`, `grep -v`)

- Useful grep-ism: context (`grep -A`, `grep -B`, `grep -C`)
- Toggle string vs regular expression matching (`grep -F`)
- Toggle case-sensitivity (`grep -i`)
- Filter chain items are commutative if there is no context
- Possible extension: context-aware filtering that does not filter on context lines, only on matched lines, thereby passing context lines through as long as filtered lines passed

Transform
- Transformation are “post-grep”, take lines as input and return a result
- Transformations can be chainable
- Lines to lines: dedup based on a strategy, filtering on extracted values (e.g., status code)
- Lines to timeseries: extract numbers from lines, rate of patterns seen in interval
- Lines to number: count of lines/pattern
- If lines contain context lines, those should be disregarded

Syntax types

For the following sections it helps to establish a type system for inputs and outputs functions:

`filter_expression`	A string or regular expression to filter by.
`selector`	A set of labels used to select log streams.
`vector_expression`	An expression that returns a set of log streams
`range_vector`	A vector whose elements contain the line count for the given range

Filtering

Log line filtering have three essential filter modes: include a match and exclude a match, as well as include either or. While other query languages use AND and NOT, we use operators for brevity. Goals for filter operators:
- Quick to type, easy to remember
- Differentiate between operators and operands
- Consider long expression operands
- Operators inside of operands should be allowed

Matching is done via string or regular expressions matching. Regular expression matching covers OR logic to include either/or matches, e.g.: `foo|bar` matches on foo or bar. To model AND and NOT we add the following functions:
- `match(selector, filter_expression)` filters lines in each stream for lines that match the regular expression.
- `match_not(selector, filter_expression)` filters lines in each stream for lines that do not match the regular expression
- `match_exact(selector, filter_expression)` filters lines in each stream for lines that match the exact substring
- Combine `match(match_not())` to achieve AND/NOT logic.

- All regular expressions have to be in double-quotes. All texts for exact matches have to be in single-quotes.
- Quotation marks inside match expressions have to be escaped (escape symbol: `\`)
  `match(match({}, ”foo|baz\””), ‘bar \\’)`

Typing more than one matcher is pretty cumbersome, hence we should add the following filter operators:

- Pipe between regular expression models AND:
  `… | “foo” | “bar”`
  matches only lines that have foo and bar
- Exclamation point between regular expressions models NOT:
  `…| “foo” ! “bar”`
  matches only lines that have foo and not bar
- Exact string matching may be required for accidental regular expression, or to speed up matching, e.g., match on the literal ‘foo|bar’:
  `… | ‘foo|bar’`

Aggregation operators

Aggregation operators in PromQL work on vectors, i.e., a set of values where each value pertains to a timeseries. The logging equivalent to a time series would be a log stream, but it must be defined what a vector value means given a log stream: a vector element is the number of log lines in a given range. By default, the range is determined by the query’s step parameter (see below), e.g., if the step is 1, then the line count is bucketed for every second. The following operators take vectors as inputs and reduce them in dimensionality:

- `sum(vector_expression)` calculates the sum of all lines in the range across all streams
- `min(vector_expression)/max(vector_expression)/avg(vector_expression)` calculates the minimum/maximum/average number of lines of all streams in that range
- `count(vector_expression)` calculates the number of streams for the given range
- `topk(scalar,vector_expression)/bottomk(scalar,vector_expression)` largest/smallest streams by number of lines in the range

Just like in PromQL, the aggregations can preserve dimensions (stream labels) by including `WITHOUT` or `BY.`

Range vectors

The range operator `[..]`, allows for sets of lines in a given range to be considered for transformation. E.g., `{job=”app”}[1m]` returns the 1-minute windowed line counts for each second which are then available for functional transformations like `rate()`.

Functions

Functions take log lines and transform them. Here is an overview of available functions (detailed documentation below):

- `match/match_not/match_exact/match_not_exact(selector, filter_expression)` Filter log lines on each stream
- `rate(range_vector)` returns the per-second average rate over range vectors
- `extract_number(selector, filter_expression)` returns a timeseries per stream where each log line represents a datapoint of the line's timestamp and the extracted value

Step

The step parameter governs how log lines are converted to scalar values for further computation. By default, the step is 1, treating log lines as buckets with length of 1 second.

Context

Context are the lines that immediately precede or follow a match. The number of lines to return can be specified as a request parameter to accompany the query. The number can have +/- prefix to indicate directionality. Context should not be considered when transforming lines to timeseries or numbers.
Context parameters and their grep equivalents:

- `context=3` => grep -C 3
- `context=+2` => grep -A 2
- `context=-1` => grep -B 1

Note: A previous version included the context as a query keyword, e.g.,
`{...} foo CONTEXT 3`

Sampling

When requesting a limited set of rows, sampling can helpful increase the time range covered by the row set. Sampling should be sent as a request parameter. Sampling should only apply on the matched rows, e.g., a parameter `sample=10` will return only 1/10 of matches, i.e., drop 9 out of 10 matches.

Queries by example

Each query needs to start with a stream selector. This forms source of filtering and transformations. Queries return a set of streams, each having a unique set of labels.

1. Select: Use Prometheus selectors to select the log streams that match certain labels.
   ```
   {job=”app”}
   {job=”app”,namespace=~”prod|dev”}
   ```

2. Filter: Default filtering via regular expressions, example with OR logic and implicit/explicit text match operator:
   ```
   {job=”app”} foo|baz
   {job=”app”} | “foo|baz”
   match({job=”app”}, “foo|baz”)
   ```

3. Filter: AND/NOT filters with custom operators (grep equivalent: `grep foo | grep bar | grep -v baz`)
   ```
   {job=”app”} “foo” | “bar” ! “baz”
   {job=”app”} | “foo” | “bar” ! “baz”
   match_not(match(match({job=”app”}, “foo”), “bar”), “baz”)
   ```

4. Filter: Exact string matching:
   ```
   {job=”app”} | ‘foo(‘
   match_exact({job=”app”}, “foo(“)
   ```

5. Transformation: Rate of entries per second in request log lines
   ```
   rate({job=”app”}[1m])
   ```

6. Transformation: Rate of log lines with filters
   ```
   rate(({job=”app”} | “/foo” ! “/foo/bar”)[1m])
   rate({job=”app”}[1m]) | “/foo” ! “/foo/bar”
   rate(match_not(match({job=”app”}, “/foo”), “/foo/bar”)[1m])
   ```

7. Transformation: Aggregations implicitly turn log lines into bucketed counts, effectively a gauge, with bucket size determined by the query step (default: 1s):
   ```
   sum({job=”app”})
   sum({job=”app”}) by (instance)
   ```

8. Transformation: Extract numbers from log lines
   ```
   extract_int({job=”app”}, “code=(\d+)”)
   ```

```
sum(extract_int({job=”app”}, “code=(\d+)”))
rate(extract_int({job=”app”}, “code=(\d+)”)[10m])
```

Operator and function specification

Base types

```
type labels = [](string, string)
type entry = (time, string)
type stream = (labels, []entry)
type query = (string, string) -> []stream
type sample = (time, float64)
type timeseries = (labels, []sample)
type range_literal = string
```

Operators

[range]   (range vector)

Purpose:

Count lines for each stream in the given range

Syntax:

```
[range_literal]
```

Signature:

```
([string]) : []stream -> [](labels, [][]sample)
```

| string   (AND matcher)

Purpose:

Filter log lines by the given string (include/positive).

Syntax:

```
| string
```

Desugaring:

```
expression | double_quoted_string => match(expression, string)
expression | single_quoted_string => match_exact(expression,
string)
```

! string   (NOT matcher)

Purpose:

Filter log lines that do not match the given string (exclude/negative).

Syntax:

```
! string
```

Desugaring:

```
    expression ! double_quoted_string => match_not(expression,
string)
    expression ! single_quoted_string => match_not_exact(expression,
string)
```

Functions

extract_int()

Purpose:

Extract an integer from each log line and turns it into a timeseries. The filter expression needs to have a matching group to extract the value from, e.g.: `code=(\d+)`

Syntax:

```
    extract_int(selector, filter_expression)
```

Signature:

```
    extract_int : []stream, string -> []timeseries
```

match(), match_not(), match_exact(), match_not_exact()

Purpose:

Filter log lines on each stream

Syntax:

```
    match(selector, filter_expression)
    match_not(selector, filter_expression)
    match_exact(selector, filter_expression)
    match_not_exact(selector, filter_expression)
```

Signature:

```
    match : []stream, string -> []stream
    match_not : []stream, string -> []stream
    match_exact : []stream, string -> []stream
    match_not_exact : []stream, string -> []stream
```

rate()

Purpose:

Calculate the per-second average rate of increase of a range.

Syntax:

```
    rate(range_vector)
```

Signature:

```
    rate : [](labels, [][]sample) -> []timeseries
```

Outdated draft sections below

Status Quo

Currently you can select log streams by selector, and in a separate field in the RPC specify a regular expression to be applied to the entries.

```
type labels = [](string, string)
type entry = (time, string)
type stream = (labels, []entry)
type query = (string, string) -> []stream

type sample = (time, float64)
type timeseries = (label, []sample)
```

Time ranges, limits on number of entries returned, and “direction” to apply those limits are also separate fields in the RPC.

Axioms

- We want a “Prometheus-style” query language; it should feel native and behave as expect to a Prometheus user
- We are not trying to solve all use cases and build a ElasticSearch competitor; simplicity is our main aim here.
- In particular, needle-in-a-haystack style problem, such as “Show me the N users with the highest latency”, are out of scope.
- Should be easy to build an autocomplete UI around

Challenge

The challenge is to extend the query language to do more interesting things. This probably involved moving regular expression matching into the query. Some examples:

- Find all log line that match a regular expression
- For request-per-log style logs, extract the rate of requests which match a certain regular expression
    - And be able to combine this with Prometheus timeseries
- For request-per-log style logs, extract a portion of the log line as a label
- Arithmetic on the labels
    - Find long messages, where(length) > 1000
    - Find all log lines when some filed (event duration) is > 10s

Counter examples that we do not intent to be able to solve:

- A query to find event count of source ip greater than 10000 in 30mins
- Top 10 longest requests
- Top 10 IP Addresses by Timeslice
- Identify the top 10 source IP addresses by Bandwidth Usage

Potential Solutions

Examples (1)

“Find all log line that match a regular expression.”

As a function:

```
match({job=”default/nginx”}, “DEBUG”)

match : []stream -> string -> []stream
```

As a magic label:

```
{job=”default/nginx”, __line__ =~ “DEBUG”}
```

As a binary operator:

```
{job=”default/nginx”} =~ “DEBUG”

(=~) : []stream -> string -> []stream  [but infix]
```

Examples (2)

“For request-per-log style logs, extract the rate of requests...”

Introduce range vector selectors:

```
{job=”default/nginx”}[1m]

([1m]) : []stream -> [](labels, [][]entry)

Similar for Prometheus:

([1m]) : []timeseries -> [](labels, [][]sample)
```

And a rate function, which gives you the rate of entries per second:

```
      type rate = [](labels, [][]entry) -> []timeseries

      rate({job="default/nginx"}[1m])
```

"...which match a certain regular expression"

```
      1) rate(match({job="default/nginx"}, "DEBUG")[1m])

      2) rate({job="default/nginx", __line__=~"DEBUG"}[1m])

      3) rate(({job="default/nginx"} =~ "DEBUG")[1m])

      4) rate(({job="default/nginx"}[1m] =~ "DEBUG"))

      ---

      1) Is explicitly disallow in Prometheus ('range vector of a
function result')

      2) We don't like this syntax

      3) Similar problem to 1, also don't like extra brackets

      4) rate then has "two" types - one for instance vector and one
for range; semantics not clear around ordering of operators.
```

Examples (3)

"For request-per-log style logs, extract a portion of the log line as a label"

```
      extract({job="default/nginx"}, "userid=(\d+)", "userid", "$1",
...)

      extract({job="default/nginx"}, "userid=(\d+),foo=(\d+)",
"userid", "$1", "foo", "$2")
```

Example (4)

Arithmetic on the labels

- Find long messages, where(length) > 1000
- Find all log lines when some filed (event duration) is > 10s

Should we make labels typed (not just strings), add a lets

```
let foo =
      extract_int({job=”default/nginx”}, “code=(\d+)”, “code”, “$1”)
In
      foo{code >= 500}
```

Or add a pipe operator to apply extra filters:

```
extract_int({job=”default/nginx”}, “code=(\d+)”, “code”, “$1”) |>
{code >= 500}
```

Or make it nesting:

```
filter_gt_int(extract({job=”default/nginx”}, “code=(\d+)”, “code”,
“$1”), “code”, 500)
```

Or make the matchers/selectors chainable “implicitly”:

```
{job=”default/nginx”}{code >= 500}

extract({job=”default/nginx”}, “code=(\d+)”, “code”, “$1”} {code >=
500, ...}

({...}) : []stream -> (string,op,string) -> []stream
```

Crazy extension:

```
sum(requests_total{job=”foo”}{status=~”2..”}) /
sum(requests_total{job=”foo”})

->

requests_total{job=”foo”}(
      sum({status=~”2..”})
      /
      sum({})
)
```

Left-Field Ideas

Why not use (SQL, IFQL, …)?

- Our belief more to be gained by aligning with Prometheus and PromQL
- SQL is potentially not a bad fit, but:
    - Full language is beyond our capabilities, so we'd end up using something like Postgres as a frontend
    - Partial SQL-like solutions tend to frustrate(see Influx)
- IFQL (Flux) is also a good fit, but:
    - Flux is unproven and un finished right now (although more finished than our non-existent language!)
    - Flux is verbose and doesn't lend itself to adhoc querying very well

That being said, we should be open to alternative query languages and engines in the future.