LogQL Reference

- Binary operators
  - Arithmetic operators
    - Arithmetic Examples
  - Logical and set operators
    - Binary operators examples
  - Comparison operators
  - Pattern match filter operators
  - Order of operations
  - Keywords on and ignoring
  - Many-to-one and one-to-many vector matches
- Comments
- Pipeline Errors
  - Pipeline Errors in Metric Queries
- Functions
  - label_replace()

Loki accepts a number of operators and functions within a LogQL query.

Binary operators

Arithmetic operators

The following binary arithmetic operators exist in Loki:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulo)

- `^` (power/exponentiation)

Binary arithmetic operators are defined between two literals (scalars), a literal and a vector, and two vectors.

Between two literals, the behavior is obvious: They evaluate to another literal that is the result of the operator applied to both scalar operands (`1 + 1 = 2`).

Between a vector and a literal, the operator is applied to the value of every data sample in the vector, e.g. if a time series vector is multiplied by 2, the result is another vector in which every sample value of the original vector is multiplied by 2.

Between two vectors, a binary arithmetic operator is applied to each entry in the left-hand side vector and its matching element in the right-hand vector. The result is propagated into the result vector with the grouping labels becoming the output label set. Entries for which no matching entry in the right-hand vector can be found are not part of the result.

Pay special attention to operator order when chaining arithmetic operators.

Arithmetic Examples

Implement a health check with a simple query:

```
1 + 1
```

Double the rate of a log stream's entries:

```
sum(rate({app="foo"}[1m])) * 2
```

Get proportion of warning logs to error logs for the `foo` app

```
sum(rate({app="foo", level="warn"}[1m])) / sum(rate({app="foo", level="error"}[1m]))
```

Logical and set operators

These logical/set binary operators are only defined between two vectors:

- `and` (intersection)
- `or` (union)
- `unless` (complement)

`vector1 and vector2` results in a vector consisting of the elements of vector1 for which there are elements in vector2 with exactly matching label sets. Other elements are dropped.

`vector1 or vector2` results in a vector that contains all original elements (label sets + values) of vector1 and additionally all elements of vector2 which do not have matching label sets in vector1.

`vector1 unless vector2` results in a vector consisting of the elements of vector1 for which there are no elements in vector2 with exactly matching label sets. All matching elements in both vectors are dropped.

Binary operators examples

This contrived query will return the intersection of these queries, effectively `rate({app="bar"})`:

```
rate({app=~"foo|bar"}[1m]) and rate({app="bar"}[1m])
```

Comparison operators

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

Comparison operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs. By default they filter. Their behavior can be modified by providing `bool` after the operator, which will return 0 or 1 for the value rather than filtering.

Between two scalars, these operators result in another scalar that is either 0 (false) or 1 (true), depending on the comparison result. The `bool` modifier must **not** be provided.

`1 >= 1` is equivalent to `1`

Between a vector and a scalar, these operators are applied to the value of every data sample in the vector, and vector elements between which the comparison result is false get dropped from the result vector. If the `bool` modifier is provided, vector elements that would be dropped instead have the value 0 and vector elements that would be kept have the value 1.

Filters the streams which logged at least 10 lines in the last minute:

```
count_over_time({foo="bar"}[1m]) > 10
```

Attach the value(s) `0`/`1` to streams that logged less/more than 10 lines:

```
count_over_time({foo="bar"}[1m]) > bool 10
```

Between two vectors, these operators behave as a filter by default, applied to matching entries. Vector elements for which the expression is not true or which do not find a match on the other side of the expression get dropped from the result, while the others are propagated into a result vector. If the `bool` modifier is provided, vector elements that would have been dropped instead have the value 0 and vector elements that would be kept have the value 1, with the grouping labels again becoming the output label set.

Return the streams matching `app=foo` without app labels that have higher counts within the last minute than their counterparts matching `app=bar` without app labels:

```
sum without(app) (count_over_time({app="foo"}[1m])) > sum without(app)
(count_over_time({app="bar"}[1m]))
```

Same as above, but vectors have their values set to `1` if they pass the comparison or `0` if they fail/would otherwise have been filtered out:

```
sum without(app) (count_over_time({app="foo"}[1m])) > bool sum without(app)
(count_over_time({app="bar"}[1m]))
```

Pattern match filter operators

- `|>` (line match pattern)
- `!>` (line match not pattern)

Pattern Filter not only enhances efficiency but also simplifies the process of writing LogQL queries. By eliminating the need for complex regex patterns, users can create queries using a more intuitive syntax, reducing the cognitive load and potential for errors.

Within the pattern syntax the `<_>` serves as a wildcard, representing any arbitrary text. This allows the query to match log lines where the specified pattern occurs, such as log lines containing static content, with variable content in between.

Line match pattern example:

```
{service_name=`distributor`} |> `<_> caller=http.go:194 level=debug <_> msg="POST
/push.v1.PusherService/Push <_>`
```

Line match not pattern example:

```
{service_name=`distributor`} !> `<_> caller=http.go:194 level=debug <_> msg="POST
/push.v1.PusherService/Push <_>`
```

For example, the example queries above will respectively match and not match the following log line from the `distributor` service:

```
ts=2024-04-05T08:40:13.585911094Z caller=http.go:194 level=debug traceID=23e54a271db607cc
orgID=3648 msg="POST /push.v1.PusherService/Push (200) 12.684035ms"
ts=2024-04-05T08:41:06.551403339Z caller=http.go:194 level=debug traceID=54325a1a15b42e2d
orgID=1218 msg="POST /push.v1.PusherService/Push (200) 1.664285ms"
ts=2024-04-05T08:41:06.506524777Z caller=http.go:194 level=debug traceID=69d4271da1595bcb
orgID=1218 msg="POST /push.v1.PusherService/Push (200) 1.783818ms"
ts=2024-04-05T08:41:06.473740396Z caller=http.go:194 level=debug traceID=3b8ec973e6397814
orgID=3648 msg="POST /push.v1.PusherService/Push (200) 1.893987ms"
ts=2024-04-05T08:41:05.88999067Z caller=http.go:194 level=debug traceID=6892d7ef67b4d65c
orgID=3648 msg="POST /push.v1.PusherService/Push (200) 2.314337ms"
ts=2024-04-05T08:41:05.826266414Z caller=http.go:194 level=debug traceID=0bb76e910cfd008d
orgID=3648 msg="POST /push.v1.PusherService/Push (200) 3.625744ms"
```

Order of operations

When chaining or combining operators, you have to consider operator precedence: Generally, you can assume regular mathematical convention with operators on the same precedence level being left-associative.

More details can be found in the Golang language documentation.

`1 + 2 / 3` is equal to `1 + ( 2 / 3 )`.

`2 * 3 % 2` is evaluated as `(2 * 3) % 2`.

Keywords on and ignoring

The `ignoring` keyword causes specified labels to be ignored during matching. The syntax:

```
<vector expr> <bin-op> ignoring(<labels>) <vector expr>
```

This example will return the machines which total count within the last minutes exceed average value for app `foo`.

```
max by(machine) (count_over_time({app="foo"}[1m])) > bool ignoring(machine)
avg(count_over_time({app="foo"}[1m]))
```

The on keyword reduces the set of considered labels to a specified list. The syntax:

```
<vector expr> <bin-op> on(<labels>) <vector expr>
```

This example will return every machine total count within the last minutes ratio in app `foo`:

```
sum by(machine) (count_over_time({app="foo"}[1m])) / on()
sum(count_over_time({app="foo"}[1m]))
```

Many-to-one and one-to-many vector matches

Many-to-one and one-to-many matchings occur when each vector element on the "one"-side can match with multiple elements on the "many"-side. You must explicitly request matching by using the group_left or group_right modifier, where left or right determines which vector has the higher cardinality. The syntax:

```
<vector expr> <bin-op> ignoring(<labels>) group_left(<labels>) <vector expr>
<vector expr> <bin-op> ignoring(<labels>) group_right(<labels>) <vector expr>
<vector expr> <bin-op> on(<labels>) group_left(<labels>) <vector expr>
<vector expr> <bin-op> on(<labels>) group_right(<labels>) <vector expr>
```

The label list provided with the group modifier contains additional labels from the "one"-side that are included in the result metrics. And a label should only appear in one of the lists specified by `on` and `group_x`. Every time series of the result vector must be uniquely identifiable. Grouping modifiers can only be used for comparison and arithmetic. By default, the system matches `and`, `unless`, and `or` operations with all entries in the right vector.

The following example returns the rates requests partitioned by `app` and `status` as a percentage of total requests.

```
sum by (app, status) (
  rate(
    {job="http-server"}
      | json
      [5m]
  )
)
/ on (app) group_left
sum by (app) (
```

```
  rate(
    {job="http-server"}
      | json
      [5m]
  )
)

=>
[
  {app="foo", status="200"} => 0.8
  {app="foo", status="400"} => 0.1
  {app="foo", status="500"} => 0.1
]
```

This version uses `group_left(<labels>)` to include `<labels>` from the right hand side in the result and returns the cost of discarded events per user, organization, and namespace:

```
sum by (user, namespace) (
  rate(
    {job="events"}
      | logfmt
      | discarded="true"
      [5m]
  )
)
* on (user) group_left(organization)
max_over_time(
  {job="cost-calculator"}
    | logfmt
    | unwrap cost
    [5m]
) by (user, organization)

=>
[
  {user="foo", namespace="dev", organization="little-org"} => 10
  {user="foo", namespace="prod", organization="little-org"} => 50
  {user="bar", namespace="dev", organization="big-org"} => 70
  {user="bar", namespace="prod", organization="big-org"} => 200
]
```

Comments

LogQL queries can be commented using the `#` character:

```
{app="foo"} # anything that comes after will not be interpreted in your query
```

With multi-line LogQL queries, the query parser can exclude whole or partial lines using `#`:

```
{app="foo"}
    | json
    # this line will be ignored
    | bar="baz" # this checks if bar = "baz"
```

Pipeline Errors

There are multiple reasons which cause pipeline processing errors, such as:

- A numeric label filter may fail to turn a label value into a number
- A metric conversion for a label may fail.
- A log line is not a valid json document.
- etc...

When those failures happen, Loki won't filter out those log lines. Instead they are passed into the next stage of the pipeline with a new system label named `__error__`. The only way to filter out errors is by using a label filter expressions. The `__error__` label can't be renamed via the language.

For example to remove json errors:

```
  {cluster="ops-tools1",container="ingress-nginx"}
    | json
    | __error__ != "JSONParserErr"
```

Alternatively you can remove all error using a catch all matcher such as `__error__ = ""` or even show only errors using `__error__ != ""`.

Pipeline Errors in Metric Queries

A metric query fails when a sample carries `__error__`, and returns an error with the appropriate status code. This stops an aggregation from silently miscounting the lines that failed. A label filter

on `__error__` says what to do with those lines instead:

- `| __error__=""` drops them, so the aggregation covers only the lines that succeeded.
- `| __error__!=""` keeps only them, which is how you count or alert on parse failures.
- `| __error__=~".*"` keeps every line, errored or not.

The `__error__` filter should be placed after the stage that generated this error. This means if you need to remove errors from an unwrap expression it needs to be placed after the unwrap. For example:

```
quantile_over_time(
  0.99,
  {container="ingress-nginx",service="hosted-grafana"}
  | json
  | unwrap response_latency_seconds
  | __error__=""[1m]
) by (cluster)
```

A range aggregation's `by (...)` (e.g. `count_over_time()`) keeps the error labels next to the labels it names, so an errored sample still reports `__error__`. `without (...)` behaves the same way: it keeps the error labels alongside whatever labels it doesn't exclude. A vector aggregation's `by (...)` (e.g. `count()`) drops them, like any other label it does not name.

In a `__error__` filter joined by `and` or `or`, only a part that asks to keep the errored lines keeps them:

- `| __error__="" or pod="p2"` asks to drop them, so an errored line that the `pod="p2"` part re-admits fails the query rather than being counted
- `| __error__="" and pod="p2"` asks to drop them too, and `and` applies both parts, so the errored lines never reach the aggregation and the query runs

The `__error__` filter applies to every error a pipeline can raise, not only a parser error. A line whose `| unwrap` conversion failed has no number to report, so its sample reports `0` next to the error labels; an outer aggregation reads that `0` like any other value, so check the `__error__` label before aggregating further.

Functions

Loki supports functions to operate on data.

label_replace()

For each time series in `v`,

```
label_replace(v instant-vector,
    dst_label string,
    replacement string,
    src_label string,
    regex string)
```

matches the regular expression `regex` against the label `src_label`. If it matches, then the time series is returned with the label `dst_label` replaced by the expansion of `replacement`.

`$1` is replaced with the first matching subgroup, `$2` with the second etc. If the regular expression doesn't match, then the time series is returned unchanged.

This example will return a vector with each time series having a `foo` label with the value `a` added to it:

```
label_replace(rate({job="api-server",service="a:c"} |= "err" [1m]), "foo", "$1",
  "service", "(.*):.*")
```

□□□ #2
□ □□□ □□□ 2026-10-31 13:48:01 UTC
□ □□□ □□□ 2026-10-31 13:48:02 UTC