This Document Is Shared Publicly

Propagate metric metadata via remote write

Author: Josh Abreu
Date: February 2020
Status: Draft
Visibility: Document is public

Problem Statement

Currently, Prometheus implements API endpoints to expose metric metadata (`HELP, TYPE, METRIC NAME, UNIT`). Specifically the **`/api/v1/metadata`** and **`/api/v1/targets/metadata`** endpoints.These endpoints allow correlation between metrics and metadata for both operators and integrations:

- Operators and users can take a holistic view of the metadata exposed by the Prometheus server by inspecting the JSON output of the API endpoint.
- Visualization integrations like Grafana take advantage of this API endpoint to provide users a better experience whilst using the Prometheus data e.g. Display of the HELP/TYPE for the metric under inspection or inform query hints.

At the moment of writing, metric metadata is never sent via remote write. Existing remote write integrations are not able to implement these endpoints (and therefore missout on features) using strictly the information that is sent via the protocol.

Challenges

Propagating this information is non-trivial and poses many design and implementation challenges:

- The protocol can't rely on the current implementation for samples as **metadata is never stored on disk** and samples for remote write are read from the WAL.
- Metadata is scraped and kept in-memory on a per-target basis. This implies there will be **duplication across metadata scraped by targets** and brings bandwidth concerns to the table.
- The **remote write protocol has no notion of "Targets"**. Within Prometheus, metadata is closely stored and related to a Target.
- Scraping is a mission-critical operation of Prometheus. **Any sort of blocking operation or slow-down while scraping due to propagation of metadata is not accepted**.

Goals and non-Goals

Goals

- Make the feature as frictionless as possible and allow integrations to adopt it without (hopefully) any sort of configuration changes on Prometheus. This means, it is enabled by default and does not break current implementations.
- Do the simplest thing now that allows remote integrations to replicate `api/v1/metadata` but ensure it is usable later. There are many challenges attached to this feature, I want to make sure we do the minimum that's viable but leaving room for improvement in the future.

Non-Goals

- Propagate information that supports implementation of **`/api/v1/targets/metadata.`** Specifically, target information.
- Metadata is currently a "best-effort" approach within Prometheus, it is never permanently stored and it's always kept in memory. I'm not looking to provide any guarantees that Prometheus itself does not.
- Not store and/or read metadata to/from the WAL/TSDB. Instead, we want to find what has relatively low complexity and start building a foundation for that in the future.

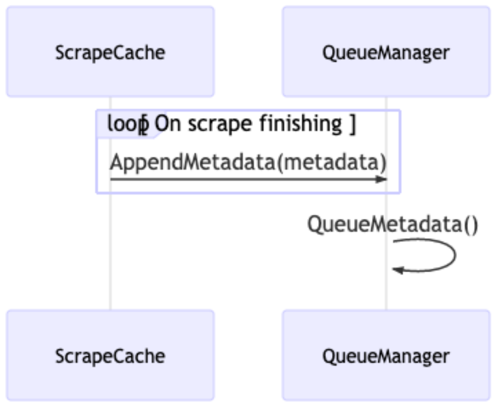
This document presents various options on how we can propagate metric metadata under the remote write protocol taking the previous into consideration.

Potential Solutions

Reading metadata

Append metadata to a remote write queue as it is scraped (push)

As a target is scraped, metadata is being set incrementally: first HELP, then the TYPE, then the UNIT, etc. Then, at the end of the scrape process for that target, the cache is verified to purge metadata entries we have not seen in the past 10 scrapes.



As a lock is already being grabbed during the process, one option to minimise locking is reading the metadata currently in the cache at this time. The idea is to send the metadata as it is being read from the cache to a remote write queue for buffering before propagation.

The challenge with this approach is that we would block until metadata finishes enqueuing, causing potential slowdowns if the remote write queue can not cope with the rate at which metadata is being pushed. Furthermore, this causes coupling between remote write and scraping.

Append metadata on a regular interval (pull)

Another option is to flip the impedance from the previous approach and instead of "pushing" metadata to the queue, we can "pull" it by reading the cache on a regular period. This approach is very similar to how the `wal.Watcher` works today for reading samples.

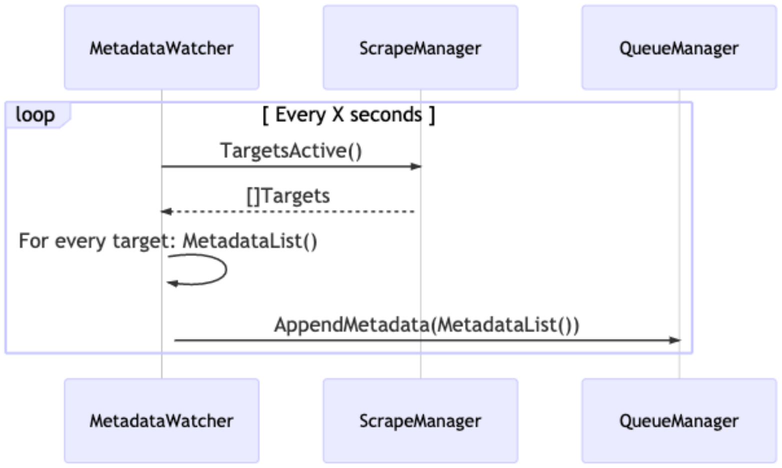
This approach avoids any sort of complete block of the scraping process as this would run on a separate goroutine. However, some locks would still be contested from time to time as `TargetsActive()` the function where we can obtain the targets to get the metadata from uses the several locks, including the one from the metadata cache as we execute `ListMetadata()`. Additionally, _some_ metadata will be missed if the targets churns as `TargetsActive()` will only return active targets, and on restart.

Sending metadata

Sending metadata poses two main questions:

- **How do we send it?**
- **How often?**

Within this section, we'll evaluate several options for sending and leave the "how often" discussion for the conclusion. Keep in mind that our main goal for the feature is to make it as



frictionless as possible and allow our users to adopt it without (hopefully) any sort of configuration changes.

Send metadata on a different Protobuf message

We create a new protobuf definition called `MetadataRequest` (or something similar) that would send metadata of metrics. Then, we have integrations handle (or not) this new message. With this approach, we create an opt-in approach where integrations can choose to implement it when they're ready.

E.g.:

```
message MetadadataRequest {
  repeated MetricMetadata = 2 [(gogoproto.nullable) = false];
}
```

Introducing the feature as a new message makes it harder to increase adoption as integrations would need to define how to handle the new message, where to send it, and eventually enable the feature within Prometheus.

A clear example of this would be Cortex, as it doesn't know how to handle any other messages but WriteRequest on the push endpoint.

Send metadata for the samples in a `WriteRequest`

Another option would be to embed the metadata for the samples being sent as part of a `WriteRequest`. That is, metadata that is specific to the current samples within the message.

E.g.:

```
message TimeSeries {
  repeated Label labels   = 1 [(gogoproto.nullable) = false];
  repeated Sample samples = 2 [(gogoproto.nullable) = false];
  MetricMetadata metadata = 3;
}

message MetricMetadata {
  enum MetricType {
    Counter        = 0;
    // … more to add
  }
  MetricType type = 1;
  string metricName = 2;
  string help = 4;
  string unit = 5;
}
```

The challenge here is the duplicity of the metadata as per the protobuf protocol, fields not recognised by the consumer will simply be dropped to ensure backwards-compatibility.

Metadata for a given metric rarely differs as targets often have high overlap in metadata. For example, at Grafana Labs the `/api/v1/metadata` endpoint for one of Prometheus servers (with 782k active series) returns 1803 metadata entries, out of those only 10 have more than one entry of metadata.

Sending metadata associated with samples results in plenty of duplicated data being sent on a regular interval. Furthermore, increasing the byte size of requests that includes the samples will result in a slower ingestion rate for integrations. As a side effect, this might affect shard calculation and thus significantly increase the memory footprint while remote write is on.

Send metadata at a different interval but within a `WriteRequest`

A final option is to reuse the same `WriteRequest` we currently use for samples. Similarly to the previous approach, adding a new field to the existing message helps us leverage the backwards-compatibility properties of the protobuf protocol.

E.g.:
```
message WriteRequest {
  repeated prometheus.TimeSeries timeseries = 1 [(gogoproto.nullable)
= false];
  repeated prometheus.MetricMetadata metadata = 2;
}
```

The important aspect of this approach is that there is no relation between the samples sent and metadata. In fact, the approach proposed encourages not sending samples when we send metadata (and vice-versa).

We'd have separate shard(s) for sending metadata, completely independent from the sending of samples. This allows us to leverage the current delivery mechanism of the message (as we'd use the same endpoint) but control the rate and what metadata gets sent independently from the samples.

This approach would effectively be changing the protocol slightly as at the moment of writing we never send `WriteRequests` without data. We'd need to make sure remote write integrations can support this case.

Conclusion

Reading metadata

I conclude that the preferable option is: Appending metadata on a regular interval. Given the sensitivity of the scraping operation within Prometheus, this approach poses a lesser amount of risk for slowing or blocking it. Operationally, it is somewhat similar to the `wal.Watcher` which makes the pattern recognizable within the codebase.

The tl;dr; is that we pull data from the scrape cache at regular intervals.

Sending metadata

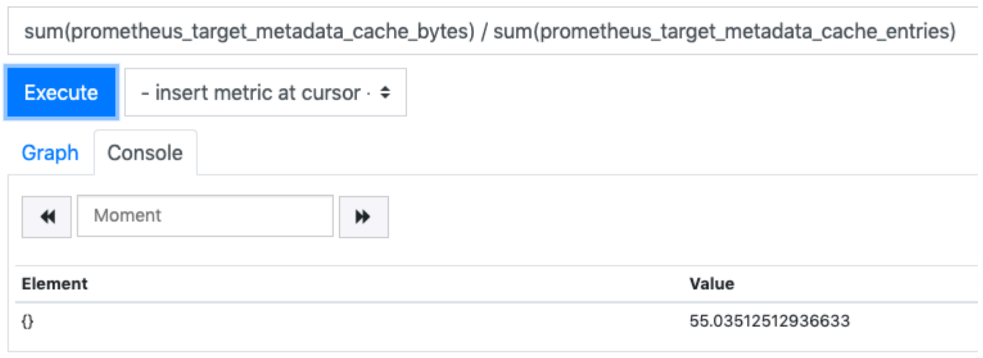
The preferable option is: Send metadata at a different interval but within a `WriteRequest`. Given the overarching goal of ease of opt-in, this option introduces the minimum amount of friction for adoption while minimising the risk of slow-down of the protocol.

Bandwidth

Since Prometheus does not impose a limit on metadata (HELP, TYPE, UNIT or Metric Name), a valid concern is how much bandwidth we end up consuming.

As a reference, we can use Grafana Labs numbers from one of our Prometheis. Keep in mind that within Prometheus, metadata is not kept per series but rather per metric name in the scrape cache (and also we keep a scrape cache per target). With that in mind, let's look at some numbers for the uncompressed data.

With ~2.6M active series, we can o[bserve](#) a total of ~10.65MB for the metadata cache size across ~194k metadata entries. **That gives us an average metadata entry size of 55 bytes**.

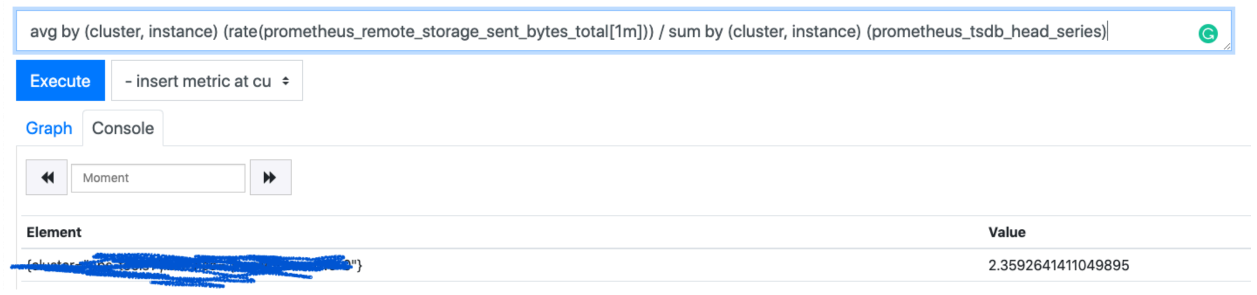


```
sum(prometheus_target_metadata_cache_bytes) / sum(prometheus_target_metadata_cache_entries)
```

Element	Value
{}	55.03512512936633

However, out of those we only have at most ~1800 unique entries (per `/api/v1/metadata`).

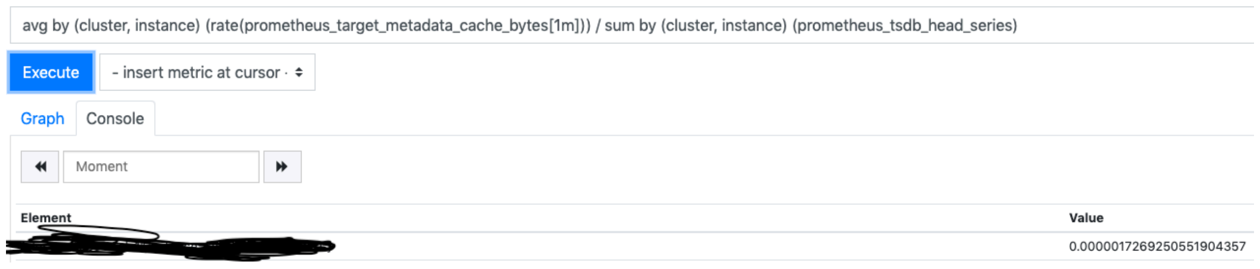
For comparison, let’s see how much bandwidth does sample data consumes for an example cluster. First, let’s look at the ratio of series and remote write bandwidth. The query below indicates that the ratio is about ~2.3 bytes per series which implies that with 1M active series is about ~ 2.1MB/s of sample data.



```
avg by (cluster, instance) (rate(prometheus_remote_storage_sent_bytes_total[1m])) / sum by (cluster, instance) (prometheus_tsdb_head_series)
```

Element	Value
[redacted]"}	2.3592641411049895

Compared this with metadata and it is observed that the ratio is about 0.000001 bytes per series and less than > 1 byte/s of metadata data with 1M active series.



```
avg by (cluster, instance) (rate(prometheus_target_metadata_cache_bytes[1m])) / sum by (cluster, instance) (prometheus_tsdb_head_series)
```

Element	Value
[redacted]	0.0000017269250551904357

Taking it one step further, the proposal is to only send this metadata every 15 seconds. Because of this, we believe it's feasible to propagate metadata via remote writing at this interval.

Frequency

Within this work, there are two key intervals. One is the interval at which metadata is read from the scrape cache and the other is the interval at which is sent to the remote write endpoint. Safe to say that the discussion revolves around "what are sensible defaults?" as both of these should configurable by an operator.

We propose the combination of the two and use a default of 15 seconds (similar to the scrape interval). This means, we send at the moment we pull the data from the scrape cache (and block the single goroutine while doing so).

Considering the fact that metadata is purged from the cache at the end of every scrape and we have a default scrape interval of 15 seconds. I think this interval gives us enough time to purge *some* metadata we've not seen in a while.

Open Questions

References

Diagrams
[1]

```
sequenceDiagram
  Loop AppendMetadata
          ScrapeCache->>+QueueManager:
      end
      QueueManager->>+QueueManager: QueueMetadata
      QueueManager->>+RemoteWriteEndpoint: Send
```

[2]

```
sequenceDiagram
  Loop Every X seconds
        MetadataWatcher->>+ScrapeManager: TargetsActive()
            ScrapeManager-->>+MetadataWatcher: []Targets
            MetadataWatcher->>+MetadataWatcher: For every target: MetadataList()
            MetadataWatcher->>+QueueManager: AppendMetadata(MetadataList())
      end
```