Staleness and Isolation in Prometheus 2.0

Brian Brazil
Founder

Robust Perception

Who am I?

- One of the core developers of Prometheus
- Founder of Robust Perception
- Primary author of Reliable Insights blog
- Contributor to many open source projects
- Ex-Googler, after 7 years in the Dublin office

You may have heard of me :)

What's this talk about?

The new Staleness semantics are one of the two big changes in Prometheus 2.0, the other being the new v3 storage backend.

It's a change that was long awaited, I filed the original bug in July 2014.

I'm going to look at what previous Prometheus versions did, the problems with that, desired semantics, how this was implemented and the ultimate results.

Staleness Before 2.0

If you evaluate a range vector like `my_counter_total[10m]` it always returns all data points between the evaluation time and 10 minutes before that.

If you evaluate an instant vector like `my_gauge` it will return the latest value before the evaluation time, but won't look back more than 5 minutes.

That is, a time series goes "stale" when it has no samples in the last 5 minutes.

This 5 minutes is controlled by the `-query.staleness-delta` flag. Changing it is rarely a good idea.

Old Staleness Problems: Down alerts

If you had an alert on `up == 0` the alert will continue to fire 5 minutes after the target is no longer returned from service discovery.

This causes alert spam for users with hair-trigger alerting thresholds when a target goes down and fails at least one scrape, and then is rescheduled elsewhere.

You can handle this by increasing your FOR clause by 5m, but it's a common issue users run into it.

Robust Perception

Old Staleness Problems: Sometimes series

Let's say you ignored the guidelines telling you not to, and on some scrapes a target exposed `metric{label="foo"}` with the value 1 and in other scrapes it instead exposed `metric{label="bar"}` with the value 1.

If you evaluate `metric` you could see both, and have no idea which came from the most recent scrape.

There are some advanced use cases with recording rules where this could be useful, it doesn't affect just the above anti-pattern.

Old Staleness Problems: Double counting

Related to the previous two issues, say you were evaluating `count(up)` for a job and targets came and went.

You'd count how many targets were scraped in total over the past 5 minutes, not the amount of currently scraped targets as you'd expect.

Put another way, if a target went away and came back under a new name you'd be double counting it with such an expression.

Old Staleness Problems: Longer intervals

If you have a `scrape_interval` or `eval_interval` over 5 minutes then there will be gaps when you try to graph it or otherwise use the data as instance vectors.

In practice, the limit is around 2 minutes due to having to allow for a failed scrape.

You could bump `-query.staleness-delta`, but that has performance implications and usually users wishing to do so are trying to do event logging rather than metrics.

Old Staleness Problems: Pushgateway timestamps

Data from the Pushgateway is always exposed without timestamps, rather than the time at which the push occurred.

This means the same data is ingested with lots of different timestamps as each Prometheus scrape happens.

A simple `sum_over_time` can't sum across batch jobs.

What do we want?

- When a target goes away, its time series are considered stale
- When a target no longer returns a time series, it is considered stale
- Expose timestamps back in time, such as for the Pushgateway
- Support longer eval/scrape intervals

Staleness Implementation

When a target or time series goes away, we need some way to signal that the time series is stale.

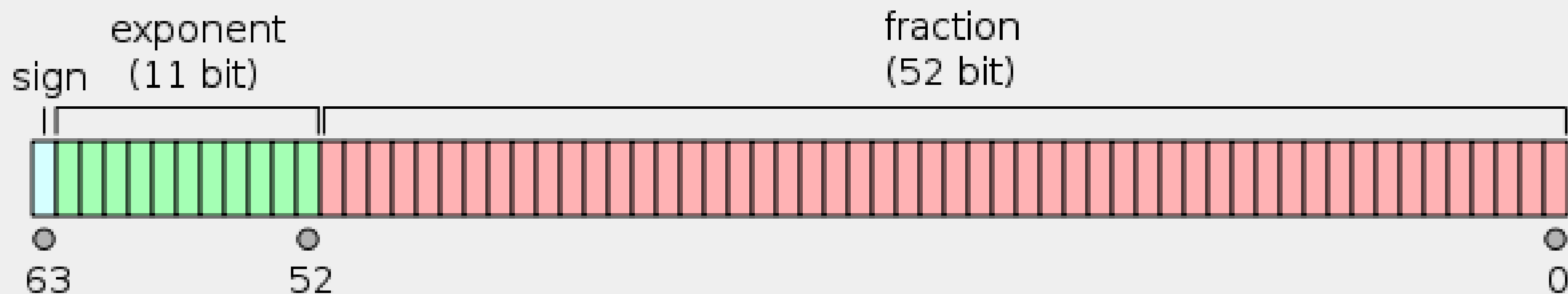
We can have a special value for this, which is ingested as normal.

When evaluating an instant vector, if this stale marker is the most recent sample then we ignore that time series.

When evaluating a range vector, we filter out these stale markers from the rest of the samples.

A Word on IEEE754

We need a special value that users can't use, and we have that in NaNs.



+Inf/-Inf have the exponent as all 1s and fraction as 0.

NaN has the exponent as all 1s and the fraction as non-zero - so there's 2^52-1 possible values. We choose one for a real NaN, and another as a stale marker.

Image Source: Wikipedia

Working with NaN

NaN is special in floating point, in that NaN != NaN.

So all comparisons need to be done on the bit representation, as Prometheus supports NaN values.

Can use `math.Float64bits(float64)` to convert to a uint64, and compare those - which the `IsStaleNaN(float64)` utility function does.

Some small changes were required the tsdb code to always compare bitwise rather than using floating point. All other changes for staleness live in Prometheus itself.

Time series no longer exposed

The easy case is time series that are there in one scrape, but not the next.

We remember the samples we ingested in the previous scrape.

If there was a time series in the previous scrape that is not in the current scrape, we ingest a stale marker into the tsdb in its stead.

Scrape fails

If a scrape fails, we want to mark all the series in the previous scrape as stale.

Otherwise down targets would have their values persist longer than they should.

This works the same was as for one missing series and using the same data structure - the time series in the previous scrape.

In subsequent failed scrapes, nothing is ingested (other than up&friends) as the previous scrape was empty.

Target goes away - Part 1

When service discovery says it no longer exists, we just ingest stale markers for the time series in the previous scrape and for up&friends. Right?

Not quite so simple.

Firstly what timestamp would we use for these samples?

Scrapes have the timestamp of when the scrape starts, so we'd need to use the timestamp of when the next scrape was going to happen.

But what happens if the next scrape DOES happen?

Target goes away - Part 2

Service discovery could indicate that a target no longer exists, and then change its mind again in between.

We wouldn't want stale markers written in this case. That'd break the next scrape due to the tsdb being append only, as that timestamp would already have been written.

We could try to detect this and keep continuity across the scrapes, but this would be tricky and a target could move across scrape_configs.

Target goes away - Part 3

So we do the simplest and stupidest thing that works.

When a target is stopped due to service discovery removing it, we stop scraping but don't completely delete the target.

Instead we sleep until after the next scrape would have happened and its data would be ingested. Plus a safety buffer.

Then we ingest stale markers with the timestamp of when that next scrape would have been. If the scrape actually happened, the append only nature of the tsdb will reject them. If it didn't we have the stale markers we need.

Target goes away - Part 4

Let's take an example. We have a target scraping at t=0, t=10, t=20 etc.

At t=25 the target is removed. The next scrape would be at t=30 and it could take up to t=40 for the data to be ingested if to went right up to the edge of the scrape timeout.

We add 10% slack on the interval, so at t=41 ingest stale markers and then delete the target.

So about one scrape interval after the next scrape would have happened, the target is stale. Much better than waiting 5 minutes!

Timestamps back in time

There's two challenges: what should the semantics be, and how can we implement those efficiently?

For range vectors, it's pretty obvious: ingest the samples as normal.

For instant vectors we'd like to return the value for as long as the Pushgateway exposes them.

Timestamps back in time, instant vectors

In a worst case at t=5, the Pushgateway exposes t=1.

Instant vectors only look back N minutes, so we'd need to tell PromQL to look back further - doable with a sample that has the timestamp of the scrape and a special value pointing to the timestamp of the actual sample.

Then at t=6 it exposes t=2.

This is a problem, as we've already written a data point for t=5, and Prometheus storage is append only for (really important) performance reasons.

So that's off the table.

Staleness and timestamps

So timestamps with the pushgateway weren't workable.

So what do we do when we try to ingest a sample with a timestamp?

The answer is we ignore it, and don't include it in the time series seen in that scrape. No stale markers will be written, and none of the new staleness logic applies.

So really, don't try to do push with timestamps for normal monitoring :)

Timestamps back in time, client libraries

Adding timestamp support to client libraries was blocked on staleness being improved to be able to reasonably handle them. But it turns out that we can't fix that. So what do we do?

Main issue we saw was that even though no client library supported it, users were pushing timestamps to the Pushgateway - and learning the hard way why that's a bad idea.

As as that was the main abuse and it turns out there are going to be no valid use cases for timestamps with the Pushgateway, those are now rejected.

Also added `push_time_seconds` metric.

Backup logic - Part 1

So if time series that don't get the new stale markers exist, how are they handled?

And what happens if Prometheus crashes before stale markers are written?

We could stick with the old 5m logic, but we did a little better.

We look at the last two samples and presume that's the interval. We then consider it stale if the last sample is more than 4 intervals (plus safety buffer) ago.

Backup logic - Part 2

Why 4 intervals? That seems like a lot.

The answer is federation.

Presuming 10s intervals a worst case is that an evaluation/scrape could start at t=10, and only get ingested at t=20. This could be scraped for federation at t=30, and only get ingested at t=40.

So at t=50 it is possible that a value at t=10 is the newest value, and not stale.

So we allow 4.1 intervals.

Implications for federation

There's no special logic been added for federation, which is a instant vector query.

Given federation exposes timestamps, this means the new staleness logic doesn't apply.

Federation is intended for job-level aggregated metrics that remain present. So it's immune to targets appearing and disappearing, so staleness doesn't matter.

Per-instance alerts would be on the federated Prometheus where new staleness works. So this is all fine in practice, if you're using federation correctly.

If.

Longer intervals

Prometheus 2.0 storage only supports pulling data from a given time range

For longer intervals we need to ask for more than 5m of data - which could get expensive.

Could ingest "live markers" every 2 minutes that point back at where the last real data point is, but if the whole goal of a longer interval was to reduce data volume this doesn't help (and would mess up the compression).

This is something for potential future work.

Implications for remote read/write

As there's no staleness related logic in the tsdb, no special handling is required for remote storage either.

However we are depending on a special NaN value, so remote storage implementations must preserve that correctly throughout their stack.

Non-storage remote writer endpoints/adapters may wish to filter out the stale NaNs.

How it worked out

- When a target goes away, its time series are considered stale
  - A little tricky, but this works now
- When a target no longer returns a time series, it is considered stale
  - This works now
- Expose timestamps back in time, such as for the Pushgateway
  - This didn't work out :(
- Support longer eval/scrape intervals
  - No changes

Staleness: What You Need To Do

If you have hair-trigger alerts based on `absent()`, extend the FOR clause.

If you're federating instance-level metrics, switch to only aggregated metrics.

Otherwise, enjoy the improved semantics :)

Staleness Summary

Special NaN stale markers are used to indicate a time series has gone stale.

Inserts when a target or time series goes away.

Doesn't apply to scraped samples with explicit timestamps.

Fallback of 4.1 intervals. 5m lookback still applies.

But wait there's more!

There's another big change I've developed for Prometheus 2.0: Isolation.

This is a problem a (tiny) number of users have actually noticed, but which is quite a big problem semantically that affects everyone.

What's Isolation?

Isolation is the I in ACID. Prometheus 2.0 has the rest (A is slightly debatable).

Isolation means that a query won't see half of a scrape.

This causes problems primarily with histograms, which depend on ingesting the whole histogram at once. But this problem is not specific to histograms, happens anytime you need to see two time series from the same scape/eval.

How not to solve it

Let's track the oldest scrape in-progress, and ignore all data after that!

I'm sure everyone will be happy with minute old data. And that wouldn't support explicit timestamps.

How about a watermark per target/rule? Would be the way to do it with 1.0, but not 2.0 as the tsdb doesn't know about targets. Also wouldn't support explicit timestamps.

How to solve it

With 2.0, scrapes/rules are committed as a batch - which is just what we need.

So each `Appender` (batch) gets a write id from a 64 bit monotonic counter.

In addition to the each sample's value and timestamp, we also note the write id.

For reading we track which `Appenders` have yet to Commit/Rollback, and what the highest write id is.

When doing a read, snapshot the open Appenders and the highest write id. We ignore those write ids, and any writes started after the snapshot.

You make it sound so easy

Isolation is a non-trivial feature in databases, and v3 storage wasn't designed with it in mind.

For example how do we efficiently store the write ids? Prometheus storage engines have only ever done time and value.

We can cheat a bit and keep it only in memory. If there's a crash, the atomicity of the Write Ahead Log (WAL) ensures we won't see partial writes.

Only keep write ids around that we need. If the lowest non-committed write id is X and no current reads need it, we can purge all write ids before that.

First get it working

To start with, I implemented a very simple and inefficient solution.

For example, the write ids were in a list that grew forever.

Also wrote unittests and stress tests to confirm that the logic worked.

Then onto optimisation through better data structures.

Data structures - Appenders

To track open `Appenders` we use a map.

When an `Appender` is created, its id is added to the map

When an `Appender` is committed or is rolled back, it is removed from the map.

Data structures - Sample write ids

To store the write ids for the samples, a ring buffer is used. This is constant time to add/remove an element.

If we need a bigger buffer, such as if a query is slow, we double the size of the buffer and copy over the values.

The data in the ring buffer represents the write ids of the most recent samples for a time series.

We cleanup old write ids at the next Append to that series. A background thread would take too long to go through potentially millions of time series.

Data structures - Queriers

To track in progress queries, we use a doubly linked list for `IsolationStates`.

When running a query we snapshot the list of open `Appenders` into an `IsolationState`, and also note down the one with the oldest write id.

We always put new `IsolationStates` at the end of the list, and when they complete they can remove themselves in constant time by updating two pointers.

When doing cleanup, the `IsolationState` at the start of the list is always the oldest so we can remove all write ids before the oldest id mentioned in the `IsolationState`.

Performance

So after all that work, what did Prombench say?

An increase of 5% CPU and 11% RAM.

This was pretty much exactly what I was expecting, which shows the importance of choosing the right data structures.

This is still far better than Prometheus 1.0.

No micro optimisations done yet, but probably already pretty close to optimal.

Further Work

This is not committed yet, https://github.com/prometheus/tsdb/pull/105

Right now isolation is for a single `Querier`.

Will need to be able to work across `Queriers` so all selectors in a PromQL expression have the same view of the data. Thus isolation needs to be exposed in the tsdb API.

Currently scrape data and up&friends are written via separate Appenders. They should use a single Appender so data is properly isolated.

Isolation Summary

Don't want to see partial scrapes when querying.

Track which scrape/eval recent writes came from.

When querying, ignore samples from writes which are still in progress.

Using the right data structures results in good performance.

Coming soon to a Prometheus near you!

Questions?

Robust Perception Blog: www.robustperception.io/blog

Training: training.robustperception.io

Queries: prometheus@robustperception.io