Prometheus Support for Counter "created" Timestamp

…also known as *_created, start* or *reset* timestamp

Date: Jun 13, 2023

Authors:

- Arthur Silva Sens (@ArthurSens)
- Bartłomiej Płotka (@bwplotka)
- Max Amin (@macxamin)
- Daniel Hrabovcak (@TheSpiritXIII)

Contributors:

- Ben Kochie
- Björn Rabenstein

Reviewers:

- Goutham Veeramachaneni

Stage: Accepted

This document was officially approved in https://github.com/prometheus/proposals/pull/29, see the latest state there. Stay here if you want to follow the previous discussions.

Related resources:

- Design doc: Transition to OpenMetrics
- OpenMetrics Spec (text)
- OpenMetrics proto
- Prometheus exposition format (text)
- Deprecated but still used Prometheus exposition format proto
- OTLP metrics spec
- Prometheus GitHub issue

Approvers:

- <add yourself>

Context

Prometheus counters are one of the most useful metric types in Prometheus. Thanks to monotonous characteristics and simple semantics it allows cheap and reliable calculations of rate/increase/delta (called “rates” in this document), even in the event of missed scrapes. Counters always start from zero on the client instrumentation, thus the rate calculations can detect reset situations (e.g. application restarted) by detecting decreasing values over time. Specifically, this matches OpenTelemetry cumulative streams with reset injection semantics.

This worked well for years, but certain edge cases started to impact the community:

- Common problems of uninitialized counters occur. We don't know when the metric with value 0 started, so the counter starts with the incremented value. This can impact visualization and alerting.
- Counter resets can go undetected if a counter is quickly incremented after a counter reset so that the next scrape sees a value that is higher than the value during the last scrape before the counter reset.
- Finding absolute value for longer metrics is expensive as one has to go sample by sample to detect resets. This impacts ingestors as it requires stateful algorithms that discern between new metrics and long-running ones it did not see before, as well as advanced algorithms like read-level deduplication algorithm in Thanos.

Furthermore, the same problems occur for Summaries and Histograms (old and native ones) since all of them are represented by counters.

For the above reasons, OpenMetrics (OM) came, before OpenTelemetry, with the optional "created timestamp" for counters, summaries and histograms. OpenTelemetry metrics specification follows a similar strategy, while strongly correlating created timestamps per sample.

Finally, solving created timestamp support in Prometheus should solve one of the problems encountered when adopting OpenMetrics in Prometheus in practice (see alternative 1 for details).

In this proposal, we explore ways to support **created** timestamps in Prometheus TSDB, so it can improve the accuracy of rate calculations, solve counter-init problems and unblock efficient transformations to different data models like OpenTelemetry or 3rd party vendors.

Assumptions

Created timestamp characteristics:

- Likely changes less often than the sample value.
- Might change less often than the duration of a single chunk.
- It likely changes more often than metadata's type, name, and help.

Exposition Formats

Before we talk about solutions, let's have more context on the ecosystem of exposition formats and the created timestamp support.

1. OpenMetrics text format

OpenMetrics text format allows exposing reset timestamp with optional "artificial" metric with _created suffix for each counter series. This has a nice property of "natural" support of Prometheus for created timestamps as the users caring about this can simply query the _created metric for each counter and have created timestamps over time.

2. OpenMetrics proto

The proto format allows encoding created timestamp in the sample, similar to the OpenTelemetry start timestamp. Generally, the client ecosystem does not support this format yet.

3. Prometheus exposition format proto

There is no created timestamp in this deprecated format. However, it is worth highlighting that this format is the only one that supports native histograms and adding a created timestamp in a similar fashion to OpenMetrics proto (in the sample) is trivial.

Other Inputs

Prometheus receive Remote Write

Users can use remote write to append to Prometheus TSDB. Remote write currently does not support created timestamps explicitly (unless we put them as separate series), but it likely will in the future.

OpenTelemetry OTLP metric ingestion

So far it was only proposed in DevSummit, but future Prometheus versions might support ingesting metrics using OTLP metric. OTLP allows passing created timestamps per sample.

Goals

- Prometheus can collect and store reset timestamps for counters, summaries and histograms on scrape using common exposition formats.
- Prometheus will persist created timestamp across restarts (WAL).
- Prometheus can use created timestamps for rates
- Prometheus can inject created timestamps for stateless remote write use.
    - Stateless means here that every streamed batch of samples for certain counters have its created timestamp known.

Non-goals

- Propose changes to Prometheus text exposition format for now. Advanced functionalities like exemplars, native histograms or created timestamps are not on the short-term roadmap.
- Elaborate on technical details for rate implementation (should be in a different proposal).
- Remote Write extensions (should be in different proposals).
- Support Gauges and Gauge-like metrics (Info, Stateset, Gauge Histograms). None of the exposition formats allows this, thus it's out of scope.

Proposal

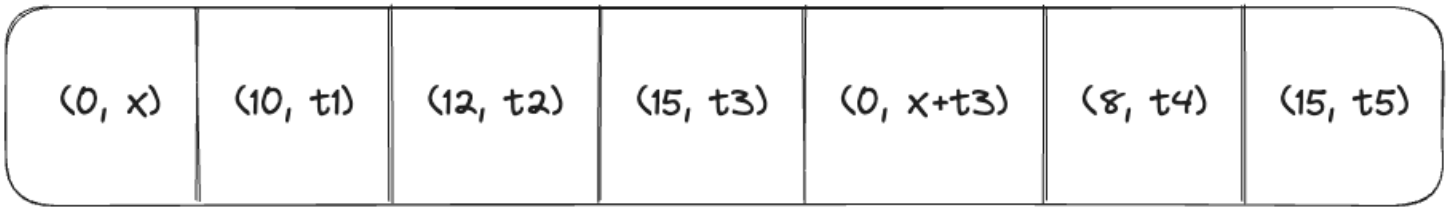
We propose to append a newly appearing created timestamp as an additional 0-value sample to the TSDB. We call this a "**zero injection**".

During a scrape, created values are processed before normal sample values. If a certain metric is not yet present in the TSDB, a "synthetic" sample is appended with value 0 and timestamp equal to the created timestamp found in the scrape. Followed by appending the actual sample with the usual scraped value and timestamp.

```
# HELP http_requests_total Amount of http requests
# TYPE http_requests_total counter
http_requests_total 10
http_requests_created x
```

On following scrapes, created timestamps are not appended until a scrape finds a created timestamp higher than the latest sample's timestamp, which would trigger the insertion of another sample with a 0-value sample.

```
# HELP http_requests_total Amount of http requests
# TYPE http_requests_total counter
http_requests_total 8
http_requests_created x+t3
```



Benefits

- Little changes to TSDB storage and no change to interfaces, the easiest option to implement.
- No change needed for rate's logic to have more accurate behavior
- In theory, if we don't want to support stateless RW, we don't need to change Remote Write to preserve (some) created timestamp information

Downsides

There are a couple of issues or tricky cases we have to acknowledge or mitigate:

There are cases where we will have to lose created timestamps (CT) information (e.g. out-of-order CT or too old CT).

There are cases where we will have to ignore CT, by dropping it, because ingesting them as synthetic zero would be misleading for rates. However, the same information might be still relevant for other systems (e.g. third party).

Let's go through those cases.

In the proposed flow, given TSDB append semantics following scenarios are accepted. Those are valid cases, when ingesting sample **S** with defined created timestamp **CT,** in Prometheus head **H:**

a. **CT** == **prev CT** (no-op)

Where **prev CT** means the last S.timestamp where S.value == 0 value or S.timestamp of the sample before "reset" situation (where the last sample is bigger than the next one).

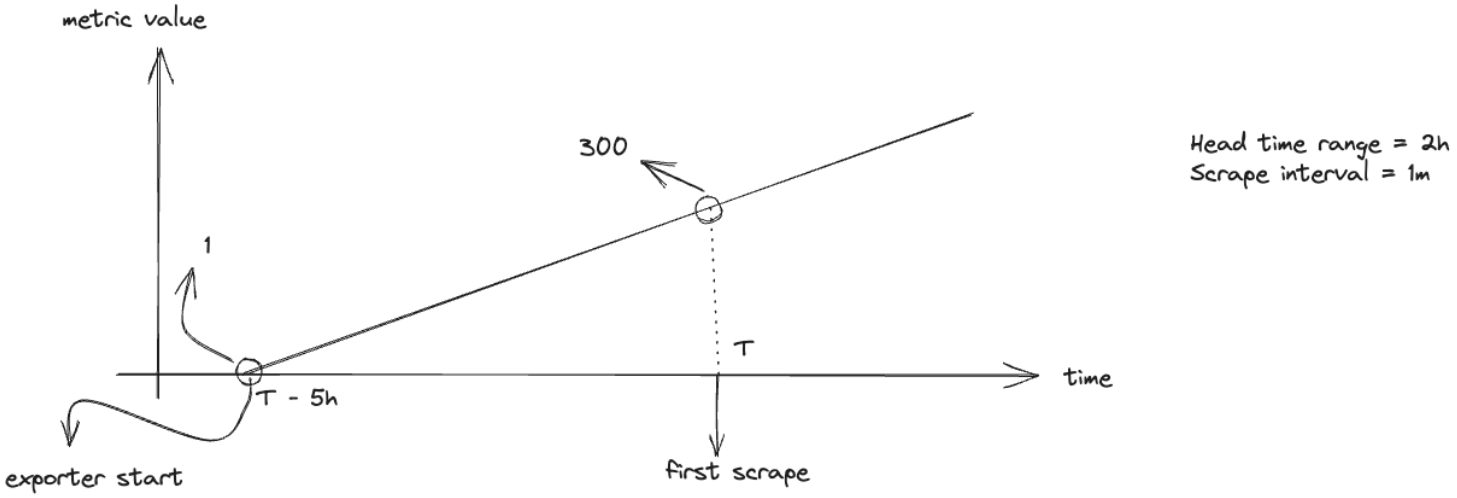
b. **CT** > **prev S**.timestamp AND **CT** < **S**.timestamp AND **S**.value > 0 AND **CT** > **H**.startTimestamp
c. **CT** == **S**.timestamp AND **S**.value == 0 (no-op)
d. **CT** is not specified (no-op)
e. **CT** != **S**.timestamp AND **S**.value == 0

Generally, specs are quite fuzzy about what created timestamp edge cases we can observe. They define created timestamp as "Info that can help ingestors discern between new metrics and long-running ones it did not see before." (OM) or "In an unbroken sequence of observations, the StartTimeUnixNano always matches either the TimeUnixNano or the StartTimeUnixNano of other points in the same sequence" (Otel). Let's outline **a few**

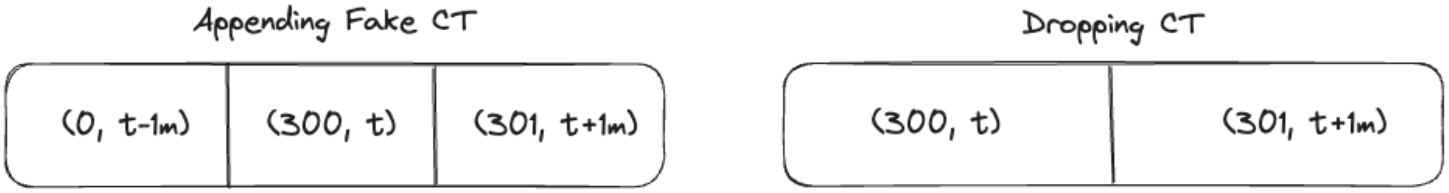
failure modes we can observe when ingesting a sample **S** with defined created timestamp **CT** in Prometheus head **H:**

A. **CT < H.**startTimestamp

Situations when **CT** is older than Prometheus' head (block was already cut for the given head period) can be quite common. E.g. the process might be running 5h and Prometheus wasn't scraping it, after 5h we finally start scraping it so **CT** == **T-5H** and **S**.timestamp = **T**. We could "fake" **CT** by appending **CT**.value = 0 with **CT**.timestamp = **S**.timestamp - scrape interval to always have at least 1 **CT** per series but we'd provide incorrect values to rates.



In a situation like the image above, where the metric is increasing on a rate of 1 per minute and the **CT** is below the **H**.startTime. If we append **CT** as **S**.timestamp - scrape interval, we would see incorrect behavior during rate calculation.



For that reason, we'd prefer to not ingest **CT** at all if it **CT** < **H**.startTimestamp and focus on delivering correct values for rates.

B. **CT** > **prev CT** AND **CT** < prev **S**.timestamp

This can happen as a client error, if the client starts to export **CT** when it previously wasn't or if there is clock skew between Prometheus and the exporter being scraped. We could use out-of-order ingestion to preserve the information, but OOO is expensive, specially for only one sample to be injected in OOO. Also, we plan to solve this problem in our future work. For those reasons, the **CT** will be dropped in this scenario.

C. **CT** > **S.**timestamp
D. **CT** == **S**.timestamp AND **S.**value != 0

C,D and cases are semantically incorrect, TSDB ignores **CT** assuming client error.

Other alternatives (e.g. metadata) could allow us to preserve CT in those tricky cases, without appending limitations.

Precision issue around scrape intervals

As mentioned in Created Timestamp semantics, **CT** SHOULD be set as the time when a time series first came into existence. If Prometheus scrapes a target close(think of miliseconds) to the time when a series was created, short rates (lower than interval between 2 scrapes) may provide a highly exaggerated result.

beorn@grafana.com: One solution, for a series that is seen for the first time–do not use the actual created timestamp, but use t1 - scrape_interval as the timestamp (simulating a scrape of the counter before it was first incremented).

What to do: TBD

There will be additional changes required to support stateless Remote Write with created timestamp support or OTLP

Some systems strictly require timestamp information for each metric. Users of such systems have to ensure all samples have associated created timestamps.

Especially for streamed protocols like Remote Write it's impractical (e.g. hard to scale) to only rely on occasional synthetic zeros in a batch of samples sent in each remote write message. One issue is that you have to iterate over past samples to tell the CT of the given sample. Second and more tricky issue is that, when a batch starts from a non-zero sample the CT is either unknown or relies on stateful caching of last seen CT in the previous batches.

This is similar for potential OTLP or federation with CT.

To mitigate this a separate proposal will be made to propose creating a timestamp to Remote Write spec e.g. per sample or series. For example remote write request could add created timestamp information for given time series sample batch ONLY if the first sample is non zero.

With this we have to acknowledge two disadvantages regarding proposed solution (adding synthetic 0s):

- A slight difference in semantics compared to TSDB. This means potentially duplicated information in Remote Write. With improved Remote Write, it's likely we will send occasional synthetic zeros AND e.g. explicit created timestamp if the first sample in batch is non zero (or some form of this).
- To implement such improved Remote Write, TSDB would need to cache CT information in certain cases, where it's impossible or inefficient to traverse samples in order to check synthetic zeros.

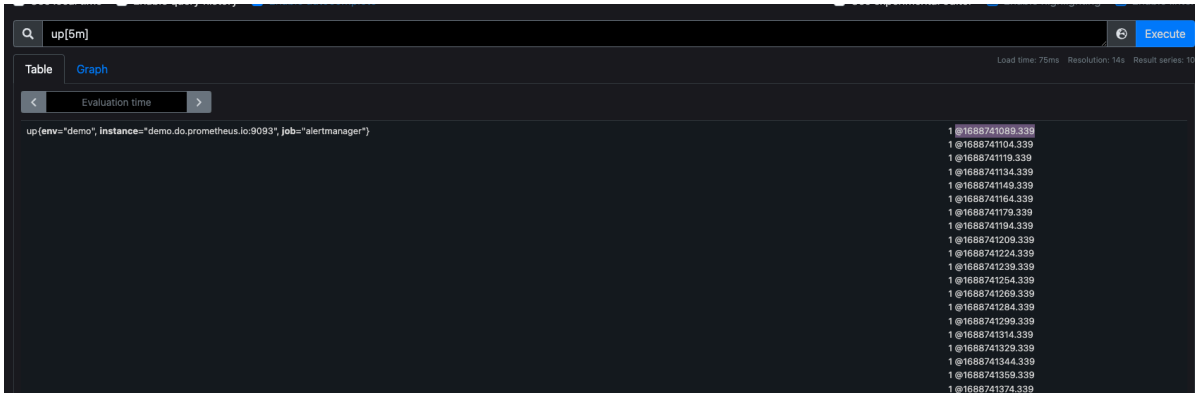
With those drawbacks acknowledged, we still think synthetic zero is the best solution.

Other issues

Less important issues to acknowledge:

- Does not help with metadata improvements we need (see an alternative)
- For users, the synthetic zero-sample looks like a normal sample from the scrape

Users can't rely on sample timestamps to tell when exactly scrape happened. Users often check scrape exact timestamp via Table view:



For synthetic zero you will observe a timestamp which is CT, not scrape timestamp. This might be misleading if users are debugging e.g. why the scrape interval is uneven.

However, users should not rely on this anyway, because some clients might expose “explicit” timestamps.

- No efficiency improvement for operations that convert counters to absolute count for given range (rates, Thanos read deduplication)

As beorn@grafana.com described it: With correct exposition and ingestion of created timestamps, you can detect a counter reset solely by a change in the created timestamp. If you stored the created timestamp as per-series metadata, you could simply look up that small piece of metadata and know already where the counter resets are. If you then know that the range in question doesn't have a counter reset, you can completely skip counter reset detection. Otherwise, you can still identify the chunk with the reset and only go through this chunk rather than the complete range.

For Prometheus, rate inefficiency was not a biggest issue, thus we propose to acknowledge this and move on. In case when rate inefficiency becomes significant or for Thanos cases, additional data structures can be added on top of Prometheus model to “cache” changes in created timestamp.

- Won’t work easily on gauge (?). Not really a big deal since supporting gauges is a non-goal.

Collection

We propose created timestamps to be supported when Prometheus is used with Prometheus protobuf format, OpenMetrics proto and text formats. Let’s explore each case:

Prometheus proto exposition format

Given recent discussion around the OM deprecation and the fact Prometheus proto is the only protocol that supports native histograms, we propose to add created timestamp to Prometheus proto exposition format.

Similar to OM proto, we propose to add created timestamp to each metric type that should support it, so:

```
message Counter {
  optional double   value    = 1;
  optional Exemplar exemplar = 2;
  optional int64    created_timestamp_ms = 3;
}

message Summary {
  optional uint64   sample_count = 1;
  optional double   sample_sum   = 2;
  repeated Quantile quantile     = 3;
```

```
  optional int64    created_timestamp_ms = 4;
}

message Histogram {
  // ...
  repeated double positive_count    = 14; // Absolute count of each bucket.
  optional int64  created_timestamp_ms = 15;
}
```

Once that is done, it's trivial to parse CT and use it as proposed.

Open Metrics proto

While not adopted protocol, it's similarly trivial to integrate OpenMetrics proto per sample CT with the proposed solution.

Open Metrics text

For OpenMetrics text the situation is more complex. For OM text format, the _created timestamps series would be normally ingested as a normal series. However, for this solution we propose to capture the timestamp from the series and treat it as the synthetic zero. Then we propose to **remove** the artificial series to avoid further metric collisions, semantic differences (series vs created timestamp metadata) and consistency with other exposition formats.

The removal of a special metric is **a breaking change.** However, we still propose doing it to stop the spread of this confusing pattern in the community (with adequate warning). The current usage of created metrics is low and users are actually not adopting OM for that reason. Thus allowing users to opt-out (users can set if they actually want to preserve those metrics) seems like a good way forward.

Note that, the conversion of _created series to created timestamp append will not be a trivial, or efficient code. We have to check various metrics, and buffer appends of potentially related metrics. Ideally, this will go away from the next version of OpenMetrics text (outside of the scope of this proposal).

Created Timestamp semantics

It is not clear to the community if the created series of a time series should be set as the time when the process exposing metrics starts, or when the time series springs into existence for the first time.

Following OpenMetrics example projects, e.g. Client Python, the created timestamp is tied with the time series initialization, and not the process. And for good reason, we also say that client libraries SHOULD include created timestamps for a time series as the time when it first came into existence.

If it was included as the process start time, we would not solve one of the most pressing problems with rate extrapolation accuracy, where proving an accurate timestamp that represents the series initialization is important.

For a better understanding of rate extrapolation, see the excellent video made by Julius Volz.

Future Work

The synthetic 0 approach is our main proposal because it brings value for most cases while having an easier implementation path. An easier implementation path is important since the main developer (Arthur Silva Sens)

behind the proposal is a GSoC student and is still getting used to the whole codebase. It is important to say, however, that this is not the final solution.

The Synthetic 0 proposal solves the usual happy path, but there are some downsides that we can still solve:

- Cannot ingest out-of-order created timestamps.
- Cannot ingest created timestamps outside of HEAD range.
- Requires RW ingestors to implement stateful API to identify created timestamps.
- Is not aligned with the work being done around time series metadata.

We propose that the Synthetic 0 proposal is released behind a feature flag, e.g. "created-timestamp-ingestion", and we would change the implementation, following Alternative 1, before committing to full stability guarantees without a feature flag.

If needed, a new design doc will be made to further explain Alternative 1.

Alternatives

This section described variations to the different parts of the above proposal:

1. Instead of zeros, inject sentinel values

TBD (proposed by Michael).

2. Store created timestamp as metadata

From discussions that happened in Prometheus issues[1], OpenMetrics issues[2] and Google docs[3], created timestamps act closely to what we would call "time series metadata". A created timestamp is captured once during the lifecycle of a running exporter and the timestamp will be constant unless the exporter restarts or client libraries provide reset mechanisms. Such behavior is quite similar to current existing metric metadata, HELP, and TYPE.

Today, Prometheus correlates Metadata with a tuple of metric names and their corresponding scrape target. To store created timestamps as metadata, we'd change this behavior to **correlate Metadata with Timeseries**. To correctly represent the lifecycle of a monitored application that restarts over time, we'd store multiple values of Created timestamps for the same time series. At the same time, we need to acknowledge that other metadata information might also change overtime and lack of support for such scenario is already a problem today. E.g. Grafana struggles to present correct tooltips for metrics that changed their Help text or correctly populate graphs from metrics that changed their unit over time.

Essentially, the metadata structure would look like this:

```
// TimeseriesMetadata is a piece of metadata for a time series.
type TimeseriesMetadata struct {
      timeseries label.Labels
      Type    []textparse.MetricType
      Help    []string
      Unit    []string
      Created []time.Time
}
```

Pros:

- Metadata already is present in WAL (safe during Prometheus restarts)
- Metadata transfer via Prometheus Remote-Write is already a work in progress
- Recently metadata in WAL improved, so it's stored per series and over time. The Remote Write spec is in progress.
- The same strategy can be used for other metadata-like information, e.g. scrape interval.

Cons:

- Requires re-design of metadata APIs[5][6] (they are promoted as 'experimental').
- OpenMetrics and other formats (e.g. OTLP) also don't support per series metadata.

3. Append samples in a new Chunk type.

We could propose that **created timestamps** are stored alongside samples, as a new chunk type e.g. "XORct". Created will be optional during append.

Here we're semantically treating created timestamps as sample metadata.

The XORct Chunk format is heavily inspired by the XOR Chunk, with the difference that an extra <varint> is added per sample. Subsequent writes to the same series will append the delta from the previous created timestamp, similarly to what is done with samples.

The XORct Chunk format:

```
| #samples <uint16> | ts_0 <varint> | v_0 <float64> | c_0 <varint> | ts_1_delta <uvarint> | v_1_xor <varbit_xor> | c_1_delta <uvarint> | ... | padding <x bits> |
```

Write Interface

The Append() method appender interface can be modified, including an extra parameter to also store **created** values:

```
type AppenderWithCreatedTS interface {
      // "created" represents the timestamp where the counter/histogram/summary was
      //  created. If created == -1, it means there was no Created value for the time
      // series.
      AppendWithCreatedTS(ref SeriesRef, l labels.Labels, t int64, v float64, created int64) (SeriesRef, error)
      //...
}
```

If the appender does not have a value for created timestamp, an invalid number will be stored instead.

Read Interface

The Chunk interator interface would change to also return Created timestamps as a third argument.

```
// Iterator is a simple iterator that can only get the next value.
// Iterator iterates over the samples of a time series, in timestamp-increasing order.
type Iterator interface {
      // <Code redacted for readability>

      // At returns the current timestamp/value/created trio if the value is a float.
      // Before the iterator has advanced, the behaviour is unspecified.
      At() (int64, float64, int64)
```

```
        // AtT returns the current timestamp.
        // Before the iterator has advanced, the behaviour is unspecified.
        AtT() int64
        // AtC returns the current created timestamp.
        // Before the iterator has advanced, the behaviour is unspecified.
        AtC() int64
}
```

When reading XORct Chunks, users of Chunk interator need to expect that invalid numbers might be returned.
Pros:

- No changes to WAL or block format.
- Allows accurate created timestamp for each sample.
- Follows OpenTelemetry semantics of correlating created timestamps with samples.

Cons:

- Requires new methods for chunk reading to pass created timestamps along.
- Requires parsing the whole chunk to get a created timestamp.
- Does not help with similar metadata-like data like scrape interval used.

4. Follow OM text and use make _created time series a common pattern in Prometheus

Following OpenMetrics Text format convention, another approach would be to establish that the created timestamp is always as separate metrics. This would also mean that other exposition formats and inputs (those which stores per sample) would create/append samples to artificially created series on scrape. Technically this would allow little to no modifications or Prometheus TSDB and remote write with OpenMetrics text format.

A lot has been said (see Design doc: Transition to OpenMetrics, [2], [3]) about the disadvantages of treating created timestamps as a separate series (see Cons), so this approach has been rejected.

Pros:

- Already implemented for OpenMetrics text.
- Already supported in Remote-Write.

Cons:

- A source of friction for adopting OpenMetrics text format for Prometheus users.
- Inconsistent with OpenTelemetry and OpenMetrics proto format.
- Confusing and surprising semantics by adding special series that have to be used especially. The link to related series is also poor and prone to errors.
- Metric name collisions - there might be user's metrics with _created suffix that means something else.
- Insufficient storage, leading up to 50% memory increase if the instance ingests mostly counters. This is because each series adds indexing overhead. The created timestamp semantics allows for significantly more efficient storage if we treat them not as a series but tailor its APIs. Additionally, increases network utilization as more data is being scraped and transferred over the wire.
- Inefficient and error-prone usage. rate(), increase() and delta() functions would need to perform extra queries to TSDB to find restarts and staleness. Metrics with clashing names might provide unexpected results.

5. New TSDB Entry

Similar to Exemplars and NativeHistograms, the TSDB can have an extended interface that allows appending created timestamps to series:

```
type Appender interface {
      // ...
      ExemplarAppender
      HistogramAppender
      MetadataUpdater
      CreatedTimestampAppender
}

// CreatedTimestampAppender provides an interface for adding created timestamps to DB.
type CreatedTimestampAppender interface {
      AppendCreatedTimestamp(ref SeriesRef, l labels.Labels, c int64) (SeriesRef, error)
}
```

The created timestamp could be stored in the TSDB as a new TSDB entry, similar to exemplars.

Pros:

- Different append implementations give flexibility for different collection strategies (exposition formats) without requiring changes to current exposition formats or client libraries.
- A stateful appender could avoid confusion or metrics collision with special _created series (OM text format).
- Allows clear and flexible semantics: created timestamp can be attached to any series and can change over time.
- Allows efficient storage (e.g optimizing to only store changes) and usage.

Cons:

- Non-trivial implementation. Requires TSDB block and WAL changes.
- Additional maintenance of different entry that looks kind of like metadata or sample.