Design doc: Prometheus Metadata Store

Author(s)	Arve Knudsen Ganesh Vernekar
Created	2024-06-24
Status	In Discussion

Publicly shared

There is also a public CNCF slack channel to discuss this project #prometheus-metadata-dev

Background

Prometheus currently does not persist any additional metadata with series beyond WAL blocks. There is some basic set of metadata that Prometheus is capable of storing only in the memory (and WAL to help restore the in-memory state and for remote-write). One of the hacks to store the additional metadata beyond these basic ones and to persist them is to add them as labels to series.

We have gathered a whole bunch of uses cases from the community signaling a need for a persistent metadata storage (some detailed analysis is in Prometheus Metadata Use Cases and Workflows):

- Persist the basic metadata that we have in the memory (described above) to give a better query experience for the past data.
- Simplify joins with info metrics (like target_info) - there is work on `info()` going on, but a metadata store could help make it more efficient.
- Store OTel resource attributes (including potentially entities when the Entity Data Model becomes generally available) as metadata and provide richer query experience. Kind of tied to the above `info` function, but being able to query them separately could improve UI experience.
    - A benefit of storing OTel resource attributes in metadata is that it won't be subject to any potential label count limit on `target_info`.
- Store OTel semantic conventions and their version history (could be useful for custom semconvs, so users don't have to provide URLs - h/t Owen Williams)
- Alternative to metric re-labelling - sometimes you want to attach some additional labels/info to the time series that are necessarily not identifiers for the series. The current

way is to do relabelling and attach labels to series, but it can cause a lot of series churn. A metadata store can help here to keep the churn low.
- Store the metric type so that we know what sample type a particular series holds (even historical changes). This can help identify if a series holds classic vs native histogram, or counter vs gauge, etc.
- Better query experience by persisting some of these metadata separately - potentially simplifying PromQL (compared to attaching metadata as series labels) and accessing metadata items only when required.
- Put the "created timestamp" in metadata and track changes over time
- Store the intended scrape interval of the series which can help in improving things like the `rate()` function. We can also do `$__range_interval` more accurately with such metadata.
- Track information about first and last sample of series (first sample linked with created timestamps, last sample is overlapping/complementing/obsoleting(?) staleness markers).
- Migration of metrics from classical histogram type to native histogram
    - krajo Krajcsovits This could be a use case for versioned semantic conventions

Also, you may want to brush up your TSDB index knowledge to understand the proposals deeply (maybe this blog post will help). But it is not strictly necessary to get a general understanding of what we are trying to propose.

Problem

Currently, there is no efficient or easy way to support the use cases mentioned above.

Some of the use cases can be worked around by attaching these metadata labels as the series labels, either at instrumentation or by relabelling the series. Adding these as series labels can bloat the number of labels and cause unnecessary churn. It can also make writing PromQL a little messy. Depending on the usage of custom metadata, it can blow up the number of series as well.

Use cases like simplifying joins with info metrics with info() function, although they can be implemented with existing storage using info metrics, it would be more efficient and native to store these info metrics as metadata and use them for queries.

Also, the basic metadata that is put into TSDB is only stored in the memory and is not persisted in the blocks.

All the above use cases and inefficient workarounds point towards the need for a unified metadata store in Prometheus.

Goals

Design the persistent storage for storing and querying the metadata efficiently such that it helps with all (or most of the “important”) use cases described above. The goal is also to store the changes to metadata over time (in other words, track the active time range of a metadata).

Non-Goals; some of them are briefly touched upon in “Other Notes” at the end of the doc

- To design an interface for ingestion or querying of these metadata. This requires a design doc and considerations of its own for scraping/remote-write/etc.
- To design the in-memory storage for metadata. For the large part, the design decisions for persistent storage will influence the in-memory design. And in-memory storage can be continuously modified without breaking backward compatibility.
- To design the interactions with WAL. This would probably be straightforward like storing series in WAL.

Proposals

We are considering metadata to be a set of label-values that are associated with time series and designing solutions to store these label-values.

Note: Any update in the index will bump the index version. This can be made backward compatible for a few releases by planning ahead (with old versions ignoring this metadata section).

Proposal 0: Do nothing

Stuck with inefficient or no solutions to above mentioned use cases. Making Prometheus a less attractive choice as a metric backend with these use cases growing in popularity.

Proposal 0.5: Agree on Metadata storage API and add initial in-mem implementation

Bartłomiej Płotka: Many discussions here discuss about sophisticated storage changes, while I’d argue we don’t fully understand efficiency characteristics on how that this data will be retrieved. The second challenge we face is the upcoming experimental Parquet storage. It’s promising but it blocks progress of metadata storage and metadata UX/PromQL aspects.

Instead of trying do things perfectly, I propose we iterate, similar to what we did with exemplar storage. Add a simple in-memory storage with the explicit limit on amount of memory or metadata elements in the storage behind a highly experimental flag. This allows us to focus on UX, integration points, but also an exact shape of Metadata API for storing and querying the

data from UX perspective. This will help design the right, persistent storage for later. Keep it simple (I'd not even go into Arrow here).

TBD: Define APIs (I will let arve.knudsen@gmail.com propose something).

For writes we already have [`AppenderV2`] that allows optional metadata attachment on every sample append. How it needs to be extended to support those use cases?

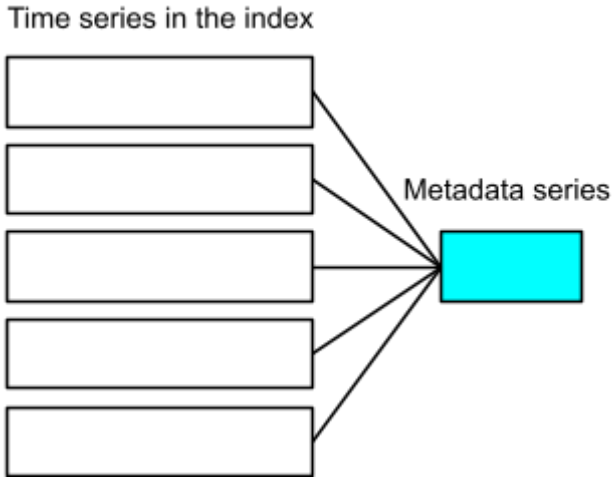
Proposal 1.0: Add "metadata series" in the index and extend the current Series format

This proposal is to add another index for metadata (technically treating metadata as a separate set of series). We can still intern the strings with series labels in the persistent blocks. Adding another index for metadata means having another inverted index for metadata to look it up easily - same as current series.

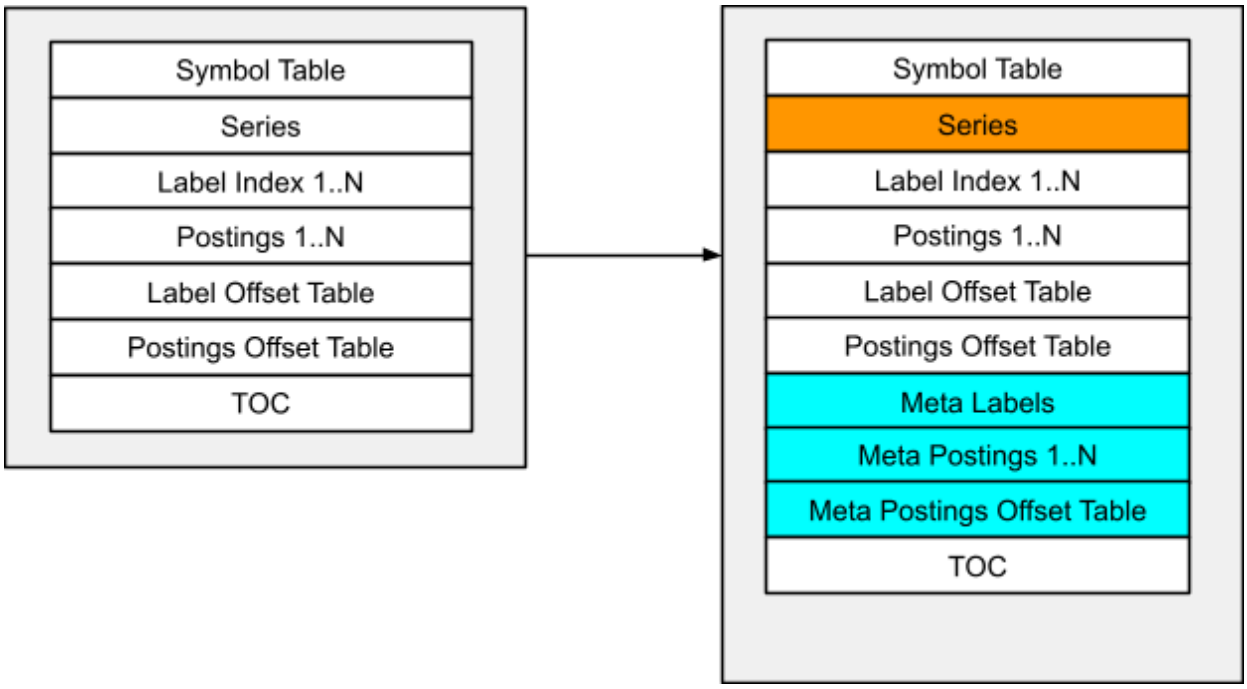
We also need to know what metadata belongs to what series. With the above index, we will have an 8 byte ID for each series and metadata items, and we can use this to map series with metadata. Depending on the requirements we can have only a map of `series->list of metadata` (say `M1`) or additionally `metadata -> list of series` (say `M2`). Further discussion only considers M1 for now.

To represent the M1, we extend the current series representation in the index to include a list of metadata items that is linked with it.

Logical representation
(metadata mapped to series in the index)

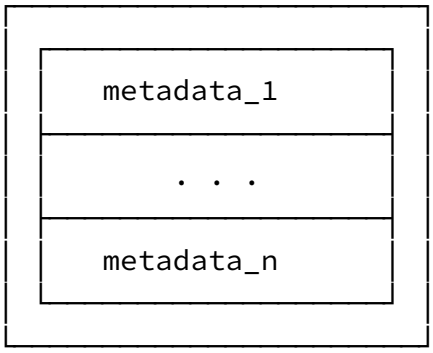


Change in the index on the disk

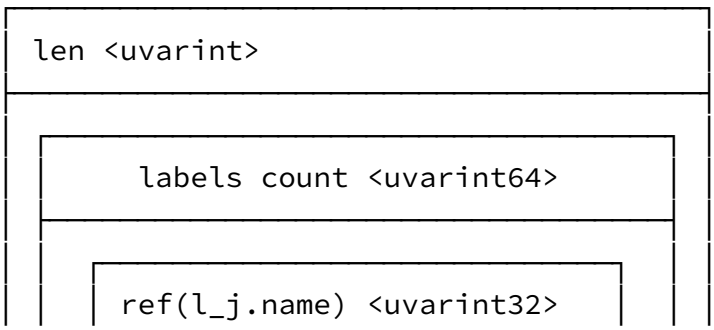


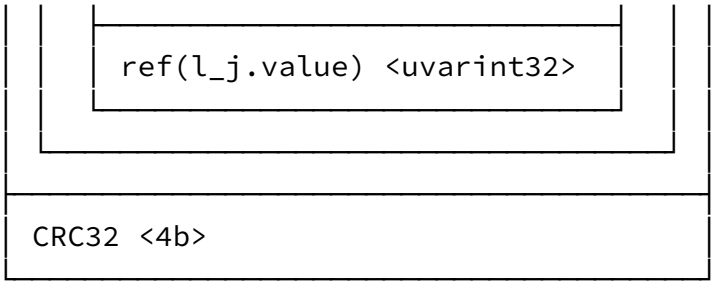
`Meta Labels`

Simplified version of `Series` section. Only label value pairs. The byte offset where a particular `meta_labels_i` starts in the file is its reference/ID (i.e. `ref(meta_labels_i)`)



`metadata_i`

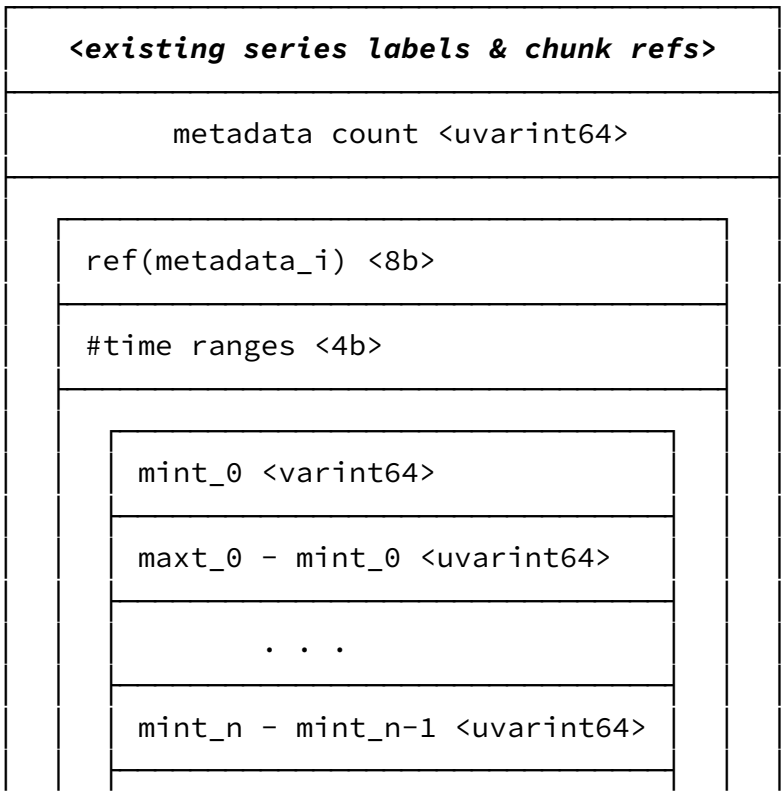


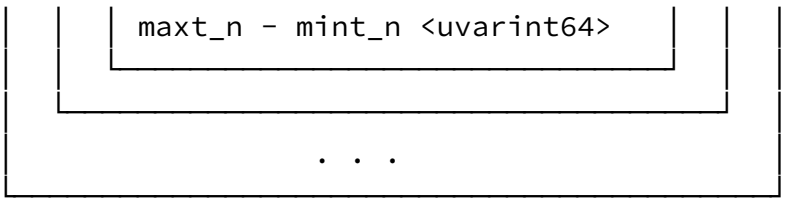


`Meta Postings 1..N`
Same as Postings 1..N, but with labels and postings replaced with those of Meta Labels.

`Meta Postings Offset Table`
Same as Postings Offset Table, but with labels and postings replaced with those of Meta Labels.

Extension to Series entries
Each series entry in the existing Series section can be extended like below. The list of metadata items are the metadata associated with this series. Since the metadata can be active for the series for different durations, we store the active time ranges with each associated metadata. Note that these time ranges only apply to the associated time series. We may choose to optimize it for cases where the metadata applies for the entire time range of the block by storing some specific numbers for `#time ranges` (example: -1 meaning it applies to the entire block).





How does it solve the use cases?

- These following use cases can be directly solved by storing the information as new metadata series and linking them to appropriate series using the map. Some of the information can be clubbed in a single metadata series.
    - Existing in-memory metadata
    - Info metrics
    - OTel resource attributes
    - Alternative to metric re-labelling
    - Store the metric type
        - If stored as a separate metadata series, identifying the type can be optimized.
    - Store the intended scrape interval
        - This can be as part of the metadata series for the existing in-memory metadata
- Simplified PromQL
    - Series won't have the metadata labels automatically with this storage and will need to explicitly attach them with future PromQL extensions.
- Created timestamp in metadata + track information about first and last sample of series
    - On paper this information can be stored in a metadata series of its own and linked to the series, but this can lead to the number of metadata series being >= number of series (assuming this information will be different for almost all series).
    - A more efficient way to store this would be to store it in the `Series` section of the index, with the series definition.

Pros

- TSDB extensions are in-line with how the rest of the index is designed. Simpler to understand, implement, and maintain.
- Solves most of the use cases.
- List of associated metadata is directly available when we lookup a series

Cons

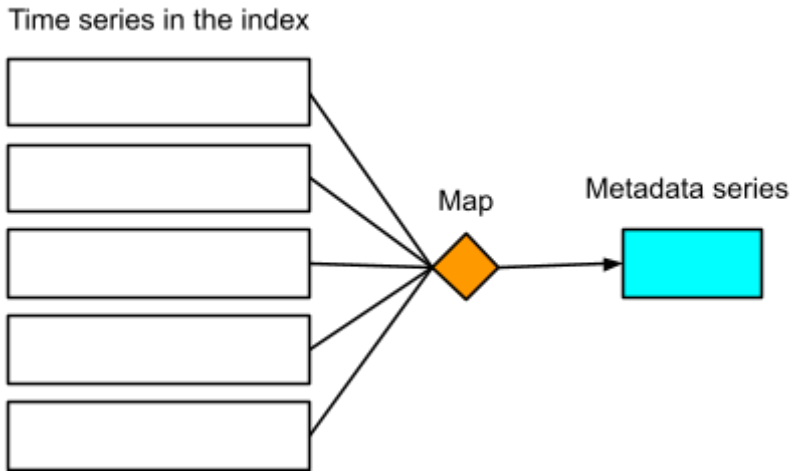
- Does not really help store created timestamp, and information about first and last sample efficiently.
- Maybe trickier to handle backwards compatibility (but maybe similar level of complexity as 1.1 below).

- Increasing the size of series entry will amplify this problem with the index size. Series section (more specifically file size until the end of Series section) cannot go beyond 64GiB in size at the moment. There are potential fixes but with performance implications. Need the linked workaround in place if we choose this proposal.

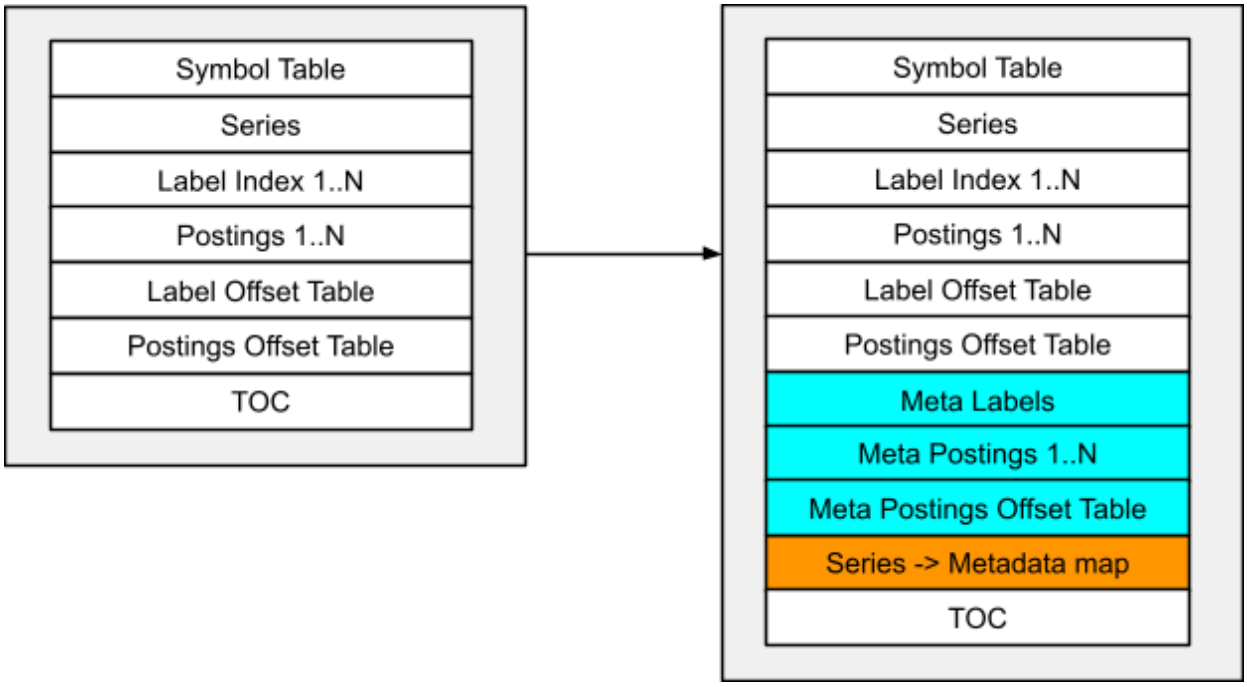
Proposal 1.1: Separate out the Series -> Metadata map

If we want to avoid the Series section increasing in size and provide better cache hits when only looking up metadata for series (without requiring samples), we can store the series -> metadata map separately. The only change from proposal 1.0 is that we do not extend the Series section and store this map separately in the index. The Metadata series index remains the same.

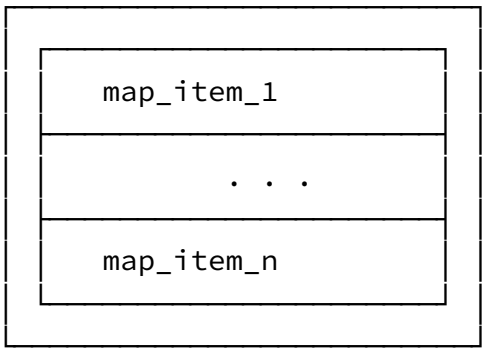
Logical representation
(metadata mapped to series in the index)



Change in the index on the disk

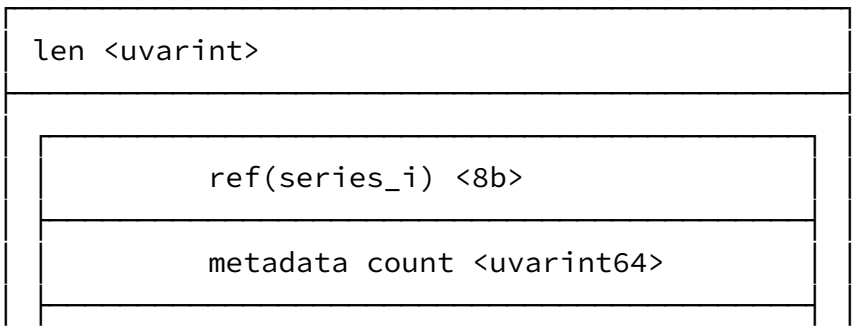


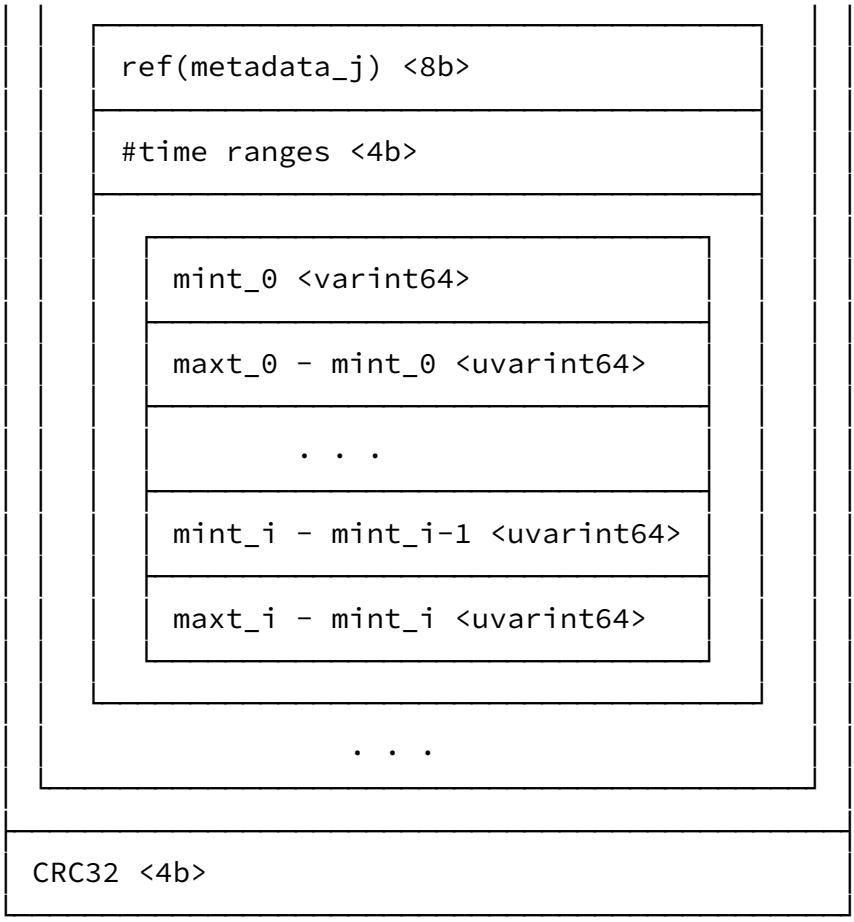
Series -> Metadata map



map_item_i

`ref(series_i)` is the “key” of this map. And the list of metadata items are the metadata associated with this series. The format of the metadata items stored here is exactly the same as proposal 1.0.





Pros over 1.0

- Won't need a fix for the 64GiB index size limit immediately (one of the con of proposal 1.0, check them for more details)
    - This is assuming that we will use 64 bit postings for metadata (compared to 32 bit postings for series that is the limiting factor).
- Better cache hits when only querying metadata for multiple series
- Easier to manage backward compatibility since the map is a separate section in the index

Cons over 1.0

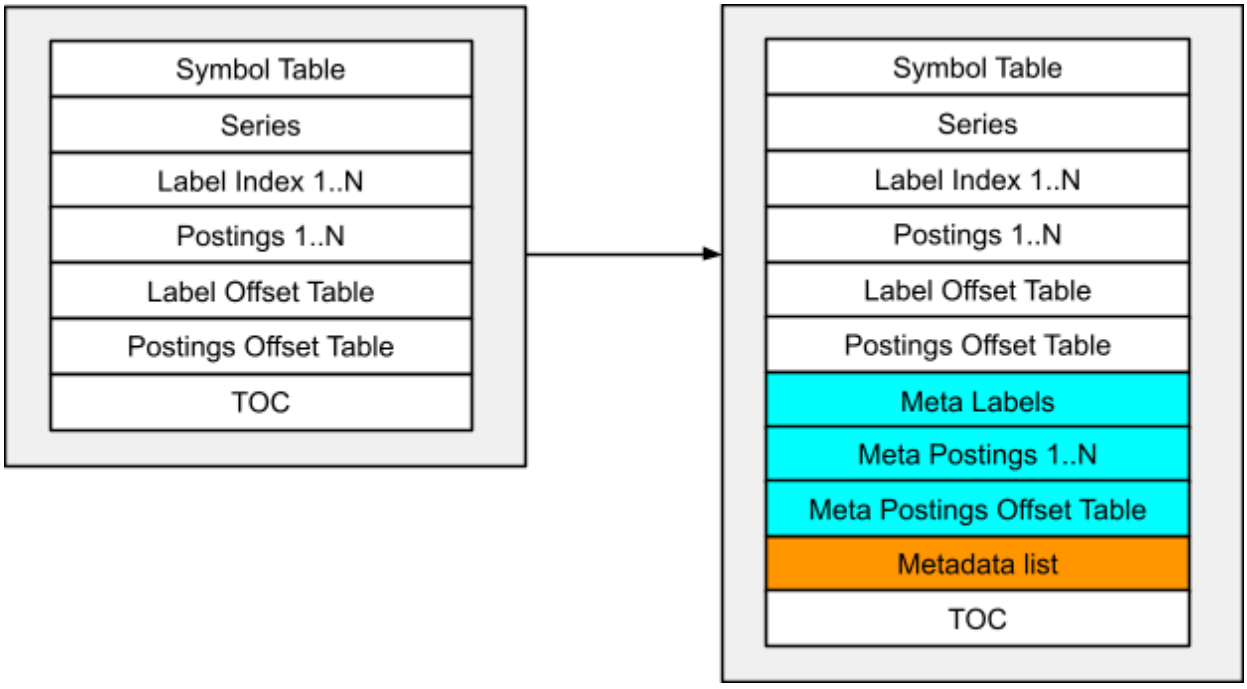
- Need to hold this new map in the memory (mostly partially)
- Linked with above: we need to do two fetches of series data when we want both sample data and metadata information linked with a series. Takes a cache hit in this case.
    - This is because when we store the partial map in the memory, we still have to touch the disk to access most of the entries.

Proposal 1.2: Mix of 1.0 and 1.1

This proposal is to avoid having a dedicated mapping section for series->metadata list and to also not increase the size of series record substantially to keep index size under 64GiB.

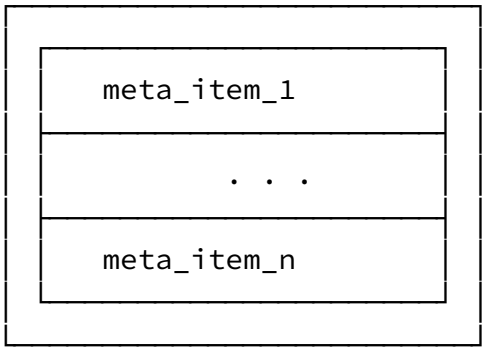
Proposal (as a diff from proposal 1.1):

- Remove the `ref(series_i) <8b>` from the `map_item_i`
- In the Series record, store a reference to the byte offset of the metadata list that belongs to this series. This is similar to how Postings Offset Table references to Postings i.



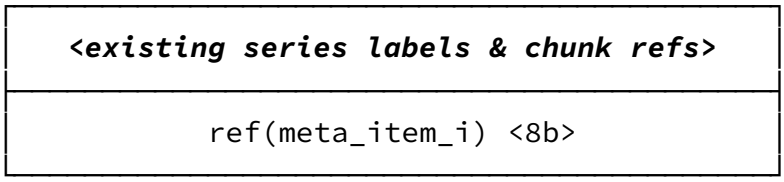
`Metadata list`

Same as Series -> Metadata map from proposal 1.1



`meta_item_i`
Same as **`map_item_i`** from proposal 1.1 but without `ref(series_i) <8b>`.
ref(`meta_item_i) =` byte offset where it starts.

Extension to Series entries
An additional reference to the metadata list

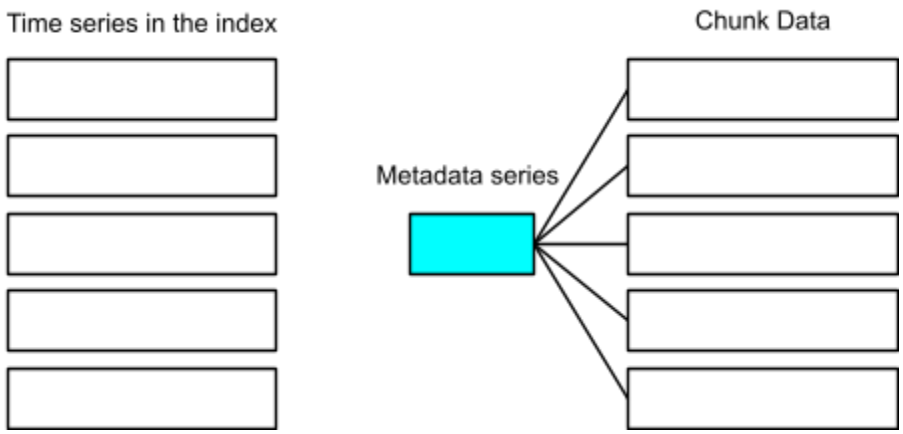


Proposal 2.0: Store the metadata references in the sample chunks

This proposal is inspired by the Loki structured metadata store, which essentially allows for tying each log line to a collection of metadata key-value pairs. It does so by storing the metadata in the chunk header, meaning the metadata applies to all the log lines in that chunk (see Loki's chunk format here).

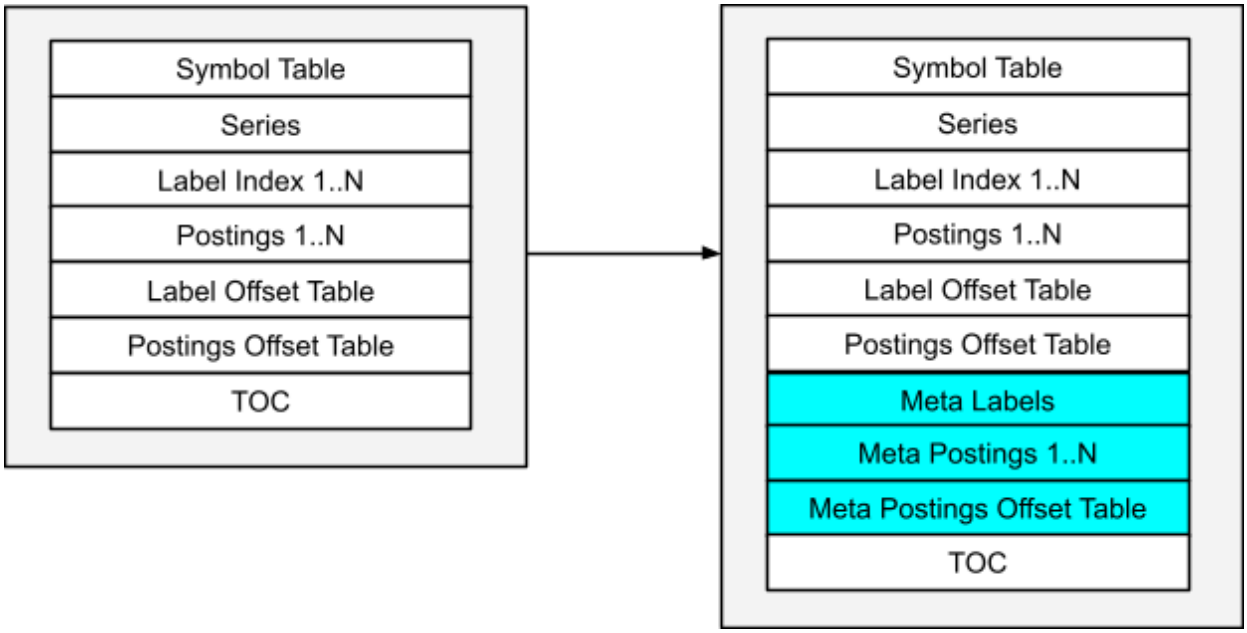
The design below proposes to store the metadata at chunk/sample level. But since TSDB's sample data is much smaller than Loki's log data in a chunk (float64 vs log string), it may not be efficient to store the actual metadata in the chunk even with string interning using the symbol table, since it might end up becoming a significant % increase of the chunk size (hence memory and storage). So the proposal is to store the Metadata labels similar to Proposal 1 above, while storing the metadata series references in the chunk.

Logical representation
(metadata referenced in the chunks)



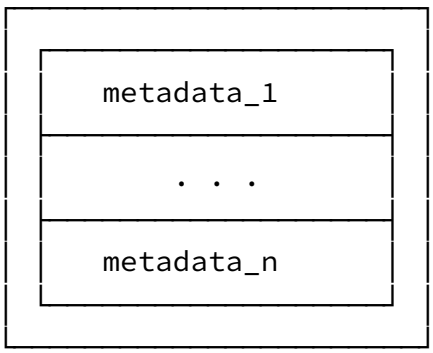
Change in the index on the disk

(same as proposal 1, minus the map)

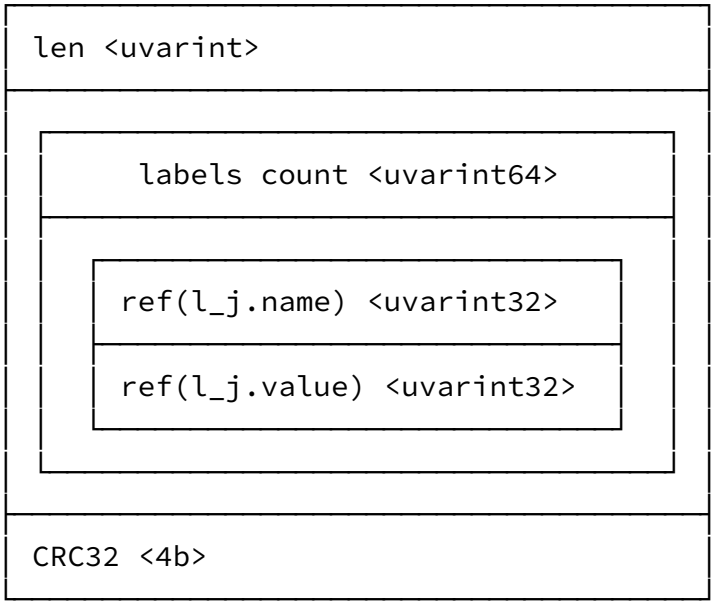


`Meta Labels`

Below is exactly the same as proposal 1.



metadata_i

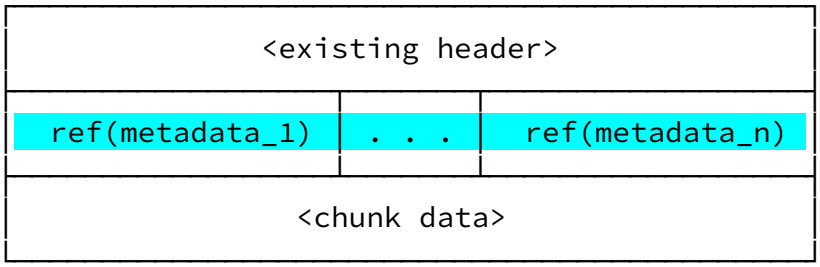


Since metadata can change in between, two options to store this change in the chunks

- Store the metadata reference with samples associated with it, like for example `[ref(metadata_i), sample_index_start, sample_index_end]`, where `sample_index_start` and `sample_index_end` are the indices of samples in the chunk, determining the range of samples that `metadata_i` applies to.
- Another option is to cut a new chunk whenever there is a change in metadata for the series. This makes the chunk simpler by only having the metadata references with no range of samples associated with them.

Assuming the metadata does not change very frequently, the below proposal is to store only the references of metadata in the chunk that applies to the entire chunk (hence cut a new chunk whenever there is a change in the metadata).

Updated chunk format:
`ref(metadata_i)` is the reference of associated `metadata_i` in the `Meta Labels` table.
The highlighted part is the new addition.



How does it solve the use cases?

Compared to proposal 1, the only difference is how we store the mapping of series to metadata. All other functionalities (like how the metadata series looks like) remain the same. So this proposal solves the use cases in the same fashion as proposal 1.

Pros

- Don't need to look up the index again for the metadata corresponding to the time range for the series. It becomes immediately available at the place of querying samples.
- TSDB index extensions are in-line with how the rest of the index is designed. Simpler to understand, implement, and maintain.
- Solves most of the use cases.

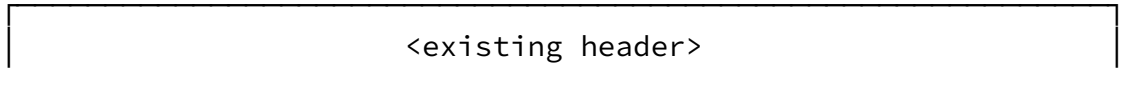
Cons

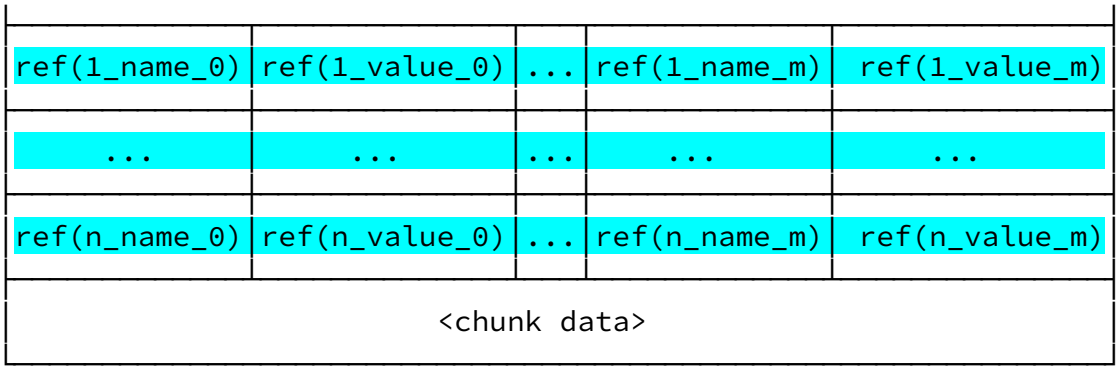
- The metadata references can/will be different for different blocks and definitely different compared to in-memory TSDB for the same metadata. Which means, when we are cutting a persistent block from in-memory data, we will have to re-write these references in *all* chunks. We will have to re-write the metadata references when compacting blocks further as well (to merge blocks or otherwise). This can end up being expensive and complex.
    - Storing metadata labels (as strings) directly is not an option because it will explode the side of chunks. The alternative of storing the string references of metadata labels using the symbol table has the same problem of string reference changing.
- Looking up only the metadata for a series will require touching all the chunks (or only the chunks for the time range asked). This can perform poorly for downstream users like Cortex/Thanos/Mimir when using object storage.
    - This is a con assuming that there will be use cases and there will be an API to query for metadata of the series without the samples.
    - We have faced this problem with series API before where it used to touch the first chunk of the series and caused issues.
- Does not really help store created timestamp, and information about first and last sample efficiently. Similar to proposal 1.

Proposal 2.1: Don't store the metadata series in the index

The design is similar to the 2.0 proposal above, but instead of storing the metadata series in the index and referencing them in the chunks, we instead skip storing the metadata series in the index and store the string references associated with the metadata series in the chunk instead.

In the below updated chunk format, each line is a metadata series, with their string references for label name and value stored. . The highlighted part is the new addition.





Cons over 2.0

- Much bigger chunks
- No easy way to just lookup a metadata - will need to scan *ALL* the chunks to know the total set of metadata
- Updating the references after compaction will likely be more resource intensive

Only pro

- Reduced index size (but at a bigger cost)

Proposal 3: External metadata store

Instead of putting the metadata into the index, we can use something external, for example SQLite (per block) and store metadata in that. We can then use references from this external storage and map it with series using one of the designs in proposals 1.0 or 1.1 or 2.0.

Pros compared to above proposals

- Possibly easier to modify schemas
- Don't have to design large part of storage ourselves

Cons

- Extra work to evaluate and choose the new technology.
- Have to learn new technology and integrate with Prometheus - may be more effort to learn and make the external storage adapt to our use cases than implementing the above proposals.
- Adds a major external dependency. Less control over the performance.

- Unknown implication of new incoming label names and how the storage will adapt to it in-flight.

Proposal 3b: Store metadata in Parquet

Similar to proposal 3, with the twist that we store metadata in Parquet files. I'm prototyping this in a PR. Beneath are screenshots of the demo CLI app:





OTel resources/entities

I have prototyped support for storing OTel resources and associated entities (an OTel resource may be composed of entities, in an under development version of the specification). This prototype has an endpoint `/api/v1/resources`, with parameters: `start` and `end` timestamp, `limit`, `match[]` (PromQL series selectors). The response schema looks as follows (the versions list allows for each time series' associated resource to change over time):

```
{
  "status": "success",
  "data": [
    {
      "labels": {
```

```
        "__name__": "metric_name",
        "job": "...",
        "instance": "..."
      },
      "versions": [
        {
          "resource_attributes": {
            "identifying": {
              "service.name": "...",
              "service.namespace": "...",
              "service.instance.id": "..."
            },
            "descriptive": {
              "host.name": "...",
              "cloud.region": "..."
            }
          },
          "entities": [                      // omitted if empty
            {
              "type": "service",
              "identifying": { "service.name": "..." },
              "descriptive": { "deployment.environment": "..." }
            }
          ],
          "min_time_ms": 1234567890000,
          "max_time_ms": 1234567890000
        }
      ]
    }
  ]
}
```

Pros

- Should be similar to proposal 3

Cons

- Should be similar to proposal 3

Proposal 4: Replace TSDB with Parquet

There has been an experiment to replace Prometheus' TSDB storage with Parquet. Jesús Vázquez should have context on who is working to continue this effort (Michael Hoffmann, Alan Protasio?). See Slack channel. If TSDB gets replaced with Parquet, storing metadata should be trivial.

Pros

- Trivial storage of metadata in addition to labels

Cons

- Difficulty of replacing TSDB with good enough performance

Consensus

Document the discussion and opinion here, and which of the above options is the agreed upon approach once they have been discussed - discussion can occur within the document through comments, or by having a meeting where the document is read in advance and the proposals debated. Be sure to document *who* this consensus represents.

- Ganesh:
- Arve:
- Bartek:
- Jesus:
- <add your name and choice, with a reason why you like that over the others>

Other notes

See similar Prometheus issue by Jesús Vázquez.

Below are some thoughts and discussion points for other parts of the metadata story that need to be designed. They will probably require their own design docs.

Storing created timestamp and first/last sample information

As briefly mentioned in proposal 1, technically this information can be stored in a metadata series of its own and linked to the series in above proposals, but this can lead to number of metadata series being >= number of series (assuming this information will be different for almost all series). Since this metadata is something very specific with defined use cases, we may choose to store it in the `Series` section of the index, along with the series definitions.

How will we expose/ingest metadata into Prometheus? Interface and other considerations

- Scraping of metadata has to happen at longer intervals than sample scraping to keep the network traffic lower (and scraping load). Maybe every 5 mins controlled by request header?
    - Maybe not from the comments
- All the live metadata for a series needs to be sent at those intervals, like a heartbeat, to know if any metadata is still active for a series. This will allow us to keep the active ranges of a metadata for a series.
- There is an idea of passing metadata as labels of a series with a reserved pattern, for example `__meta_otel_pid__`, `__meta_prom_created_timestamp__`, etc, but still store them separately as above. Note how the second term “otel”, “prom”, can nicely define the *family* of the metadata, hence labels under same metadata family can be clubbed into a single metadata series, `__meta_otel_pid__=”X” + __meta_otel_command__=”Y”` gives the metadata series `{meta_family=”otel”, pid=”X”, command=”y”}`. If we decide to go this route, thought needs to be given on how we represent labels that have multiple values for the same series (hence having more than one metadata item under a metadata family linked with a series).
- Alternative idea for ingestion is to separately call the existing UpdateMetadata() function to update the metadata for the series instead of passing them along with series labels. It would however be more efficient to extend the Append() function to accept an optional metadata field, to avoid multiple lookups of the same series.

How will we query the metadata?

- If we use the `__meta_X__` labels from above, the query could also allow these labels in the series selectors, and attach the specific metadata labels to the series result when the selector like `__meta_otel_pid__=~”.*”` is explicitly added. Otherwise don’t attach metadata labels by default.
- May still need an API to query metadata separately, to either get just the list of metadata items and/or API to fetch metadata linked to a series. Probably the series API can be extended to return metadata attached to the resultant series, with matchers/filters on the metadata as well.

How will we send metadata over remote write?

- Maybe send via remote write at the same frequency of scraping / ingesting in local Prometheus. The downstream TSDBs will likely follow the same interval to call a metadata stale if we standardize on something like that in Prometheus.
- By writing to WAL whenever we ingest in local Prometheus, this part gets simplified.

Loki’s structured metadata store

Loki has a structured metadata store, where every data point (i.e. a log line) points to its attached structured metadata in the chunk (as a symbol number). Using TSDB’s technique of string interning, each chunk stores all the metadata key/value pairs as symbols. See Loki chunk layout documentation. See also the structured metadata design doc.

References

- TSDB Index Disk Format