[Parca-RFC-001] Column Store for Profiling Data

Authored: 2021-11-02
Author: Matthias Loibl , Thor Hansen , Frederic Branczyk

Abstract

The Parca project wants to offer a built-in storage that is well suited for storing and querying profiling data on any dimension and can handle short-lived serverless or CI environments just as well as long-running applications. This RFC covers only the portion of the database that actively accepts writes, and everything detailed here is entirely in-memory. An RFC for persistent storage will follow.

Problem

The Parca server (and before it the Conprof project) has gone through various iterations of storage strategies including:

- Prometheus time-series style series recording full application profiles for each timestamp
    - In raw bytes and later compressed via zstd
- Store pre-calculated trees of profiling data to make generating iciclegraphs/flamegraphs simple and fast
    - The complexity of diffing, merging and storing trees were deemed unmaintainable
- Time-series of stack-traces grouped by workload labels to prevent each stack trace being in the index

All of these approaches failed to provide the flexibility for querying that we envision for the Parca project (separate RFC to follow).

The current storage architecture resembles a time-series database a lot. The way that this manifests itself the most is in trying to accumulate chunks of data to be compressed together as a series of values. The downside of doing this is that we have a worst case active chunks/series of

Processes x Pprof labels x Unique Stack Traces

It is difficult to tell whether this ends up being a problem in reality because the number of stack-traces are realistically bound, but it is equivalent to the Prometheus anti-pattern of putting IDs into label-values.

Proposal

A columnar database layout of the data could alleviate the cardinality issues, by not maintaining data by series at all, but rather batching many “rows” to then be written at once to persistent storage. A columnar database at first resembles a traditional relational database a lot, but contrary to it, as the name says, a columnar database stores the data by columns as opposed to by rows. This allows columnar databases to be excellent for data intensive operations.

High-level In-memory Layout

A potential columnar table layout for Parca’s profiling data could be:

Column Name	**Type**	**Dynamic**	**Encoding**	**Description**
Labels	String	Yes	Dictionary + RLE	**Workload labels** are those that uniquely identify the process that produced a profile. Workload labels are string columns that are dynamically created, meaning whenever a new key is seen, a new column is created to track it. Using this method has two advantages, arbitrary cardinality has the same cost as constant cardinality, and labels are tracked as part of the table directly, circumventing the need for an external inverted index. Each of the sub-columns is encoded using Dictionary encoding, which deduplicates the strings while retaining Arrow’s O(1) access guarantee.
Stacktrace	[]UUID	No	Dictionary + RLE	**Stacktrace** is an array of UUIDs that each reference a Location ID in the Metastore.
Pprof string labels	String	Yes	Dictionary + RLE	**Pprof string labels** are the string labels attached to a pprof sample. Similar to workload labels they are string columns created dynamically as they are seen for the first time and encoded using Dictionary encoding.
Pprof num labels	Int64	Yes	RLE	**Pprof num labels** are the integer labels attached to a pprof sample. Because num labels potentially have a unit attached to them, their column name is composed of label-name concatenated with the unit. Similar to workload and pprof string labels, pprof num labels are columns that are dynamically added, but contrary to the others they are of type int64.

Timestamp	uint64	No	Plain	**Timestamp** is the timestamp at which the profile as a whole was captured that this particular stack trace was a part of. Timestamps are plainly encoded. Future work might optimize the encoding by using double delta encoding, similar to the strategy of the gorilla encoding, since timestamps are already proposed to be stored in ascending order (see below), meaning when a new profile is inserted where all samples have the same timestamp they are likely to be inserted at different places of the table, so data wise timestamps are not going to be repeated but rather increasing in their layout. Similar to Prometheus, Parca and or Parca Agent would need to ensure to remove tiny amounts of jitter to increase the compression efficiency.
SampleType	String	No	RLE	**SampleType** represents the type of the value, eg. “cpu” or “allocations”.
SampleUnit	String	No	RLE	**SampleUnit** represents the unit of the value, eg. “nanoseconds” or “bytes”.
PeriodType	String	No	RLE	**PeriodType** represents the type of events between sampled occurrences. e.g "cpu" or "heap”.
PeriodUnit	String	No	RLE	**PeriodUnit** represents the unit of events between sampled occurrences. e.g "cycles" or "bytes”.
Duration	int64	No	RLE	**Duration** represents the duration over which the profile was taken.
Period	int64	No	RLE	**Period** the number of events between sampled occurrences.
TraceID	String	No	Plain	**TraceID**, in order to correlate tracing data directly with profiling data.
Value	int64	No	Plain	**Value** is the observed occurrences of the stack trace (talking about an example of a CPU profile). More generally it is the value attached to the stack trace.

All columns except duration, period and value make up the primary key, like we know from relational databases. When inserted all data is immediately sorted by the primary key in order of:

- SampleType
- SampleUnit
- PeriodType
- PeriodUnit
- Workload labels
- Stacktrace
- Timestamp
- Pprof string labels
- Pprof num labels

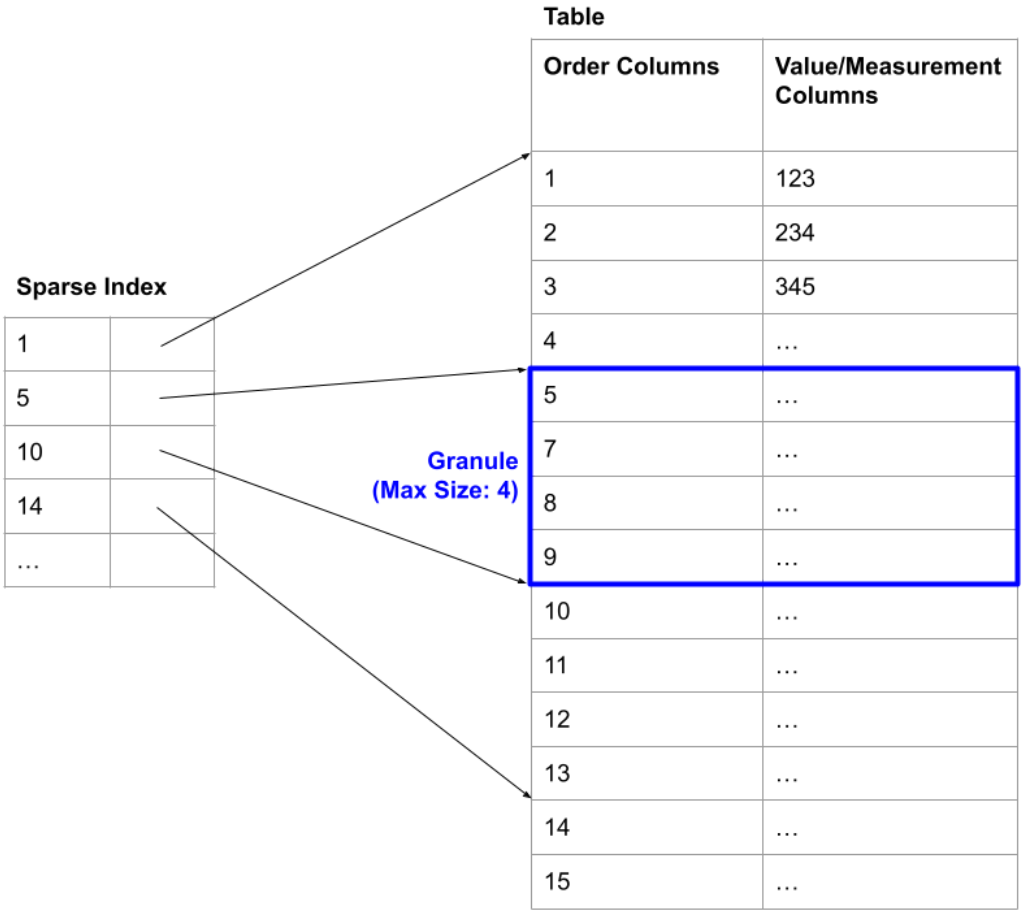
This sorting is just a proposal where the theory is that retrieving data for merging, single selects as well as stack trace searches are balanced. It is unclear what sorting is most optimal, therefore the implementation should be flexible enough to modify this afterwards. Another possibility for performance optimization could be to store the data in multiple layouts.

See example input data, their logical and physical representation in the accompanying table.

Columnar Storage Design

Everything in the design relies on writes to the column store to be in the order declared by the schema. With this property, an insert requires at most a single pass over the sparse index, which is held in memory.

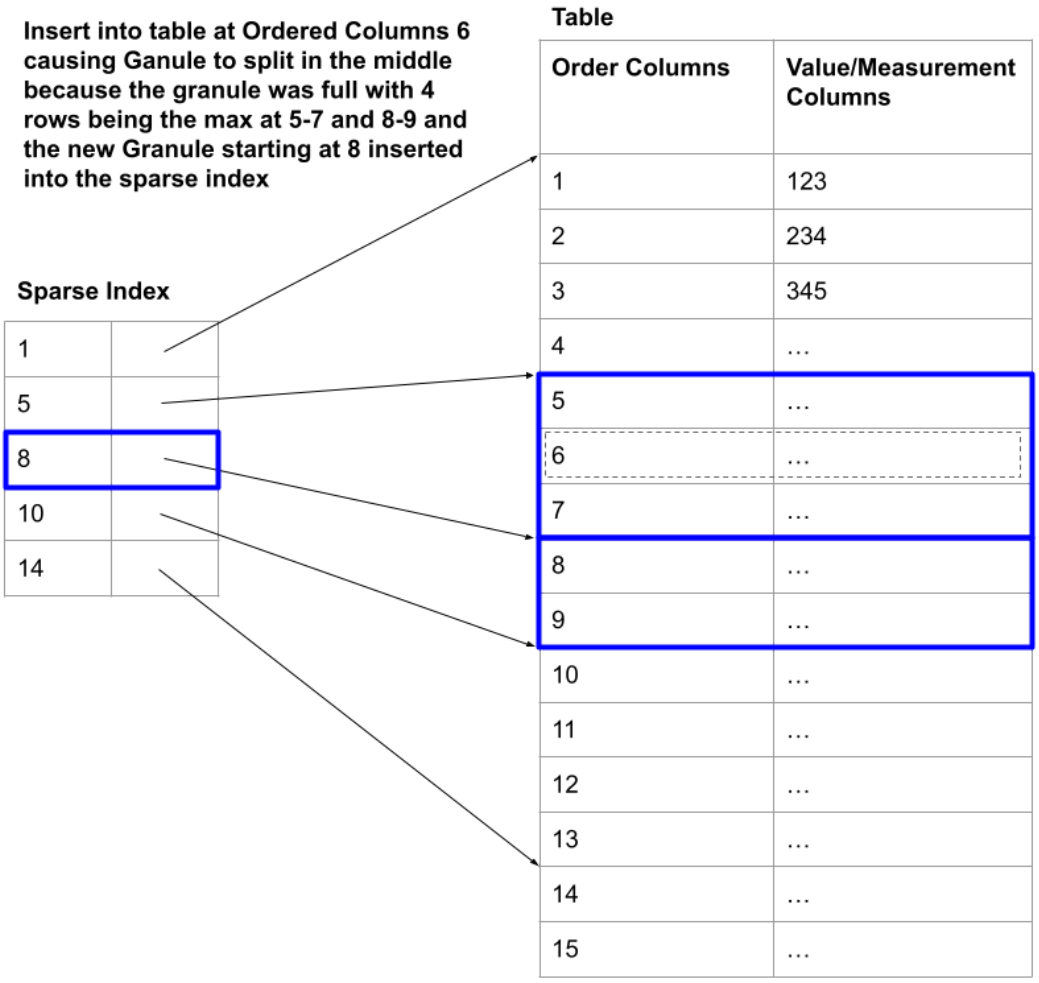
Writes are buffered in small chunks and regularly compacted into larger chunks, these chunks have a lower and upper bound of the sorted column values, these are called granules going forward in this design. Essentially an LSM-tree. For a start a simple B-Tree can be used such as the Go btree implementation by Google, building a sparse index similar to that of ClickHouse. If the performance of this btree implementation ends up being prohibitive, a Bw-Tree might be a good fit as it is designed with modern hardware in mind.



The start of each granule is part of the index. From blank, the storage starts with a single granule, and every time a granule grows over the specified rows or byte size it is split into two. Thus adding 1 more entry to the index (as the initial granule already had one). In the in-memory representation, each granule has X rows, which depending on the encoding of the column is inserted into, using an encoding specific strategy.

Special case: First granule

This makes the “first” granule created as well as the “first” granule in the index slightly special as they always need to represent the lowest possible values as their lower bound as the lowest row may be higher than rows inserted into it. If this wasn’t the case then the only alternative would be to create a new Granule and insert it, likely causing an unpredictable write amplification on the index as it would depend on the order of items inserted.



While logically new rows are inserted in their correct place, in reality only the low and high points of a granule are respected and within the granule there are multiple parts of sorted data that are merged in order at read time. Another advantage of this strategy is, it will make implementing isolation of commits/transactions trivial as new/incomplete commits are simply left out of queries when merging. Asynchronously, parts are merged even without queries being executed, alleviating most of the previous extra memory cost and CPU cost at query time. When total sorting is necessary for a query, then merging is done using a direct k-way merge using a min-heap.

When querying only whole granules are read, the query engine is responsible for filtering out results further.

For performance and compression benefits it makes sense to use custom Go code for the in-memory representation of this data. At query time, however, it is converted to Apache Arrow for processing of queries by the query engine, that way the query engine can be built generically on top of the Arrow format. For a start, the storage may return all data of a Granule, even data that may not match the requested matchers, the query engine needs to ensure additionally that all data matches queried constraints.

Persistence of this in-memory store will be covered in a future RFC.

Atomicity

A problem all databases face is allowing reads and writes to happen concurrently in a safe way. There are various levels of isolation. This section describes how it is implemented in this column store. If you are familiar with strategies for isolation, the scheme proposed here is multi-version-optimistic-concurrency-control (MVOCC) optimized for the insert-only nature of the store.

Each database of the columnstore maintains a list of active transactions each of which are assigned a unique, atomically incrementing ID. Granule parts inserted during a transaction are marked with their transaction ID, a transaction that performs writes obtains two transaction IDs, one when it starts to write and one when it is done inserting, this allows other transactions to inspect whether to include writes from transactions that may not be completed yet. It also means read-transactions will ignore partial writes of a write transaction whose transaction ID is larger than the read transaction ID, making a check for whether the write transaction has finished not necessary. For the same reason, the database maintains a version of the sparse granule index for each transaction that modified it.

Splitting and compaction are distinct actions on a granule, compaction can occur even without splitting, and it would never affect the granule index since lower and upper bounds would not change, it would merely be about optimizing the layout of existing data. Splitting both compacts and splits a granule. Nevertheless, compaction and splitting actions are conceptually the same. A compaction is a splitting action with a split factor of 1, meaning out of 1 granule 1 granule is created, with the difference being since no net new granule is created, the index does not require any inserts.

Splitting granules does present an additional challenge in regards to concurrent reading and writing of the granule. Since Granule parts that are actively inserted respect only the lower and upper bounds of the sorted data, when the upper bound changes (which is exactly what happens during splitting of granules), then the Granule parts inserted potentially concurrently, are no longer correctly inserted. For this reason the compaction that happens during splitting can only occur on granule parts lower than the splitting's transaction ID.

Since splitting does not delete or modify existing or add new data, but only optimizes the representation of existing data, it would be ideal if splitting did not block reading or writing of data. In order to achieve this, while compacting a granule, that granule maintains a list of the granule parts inserted concurrently and if the granule is being read, it uses the list of parts from before compaction was started, concatenated with the list of parts inserted during compaction.

Granule parts inserted concurrently are added to all splitted parts and until their next compaction are filtered by their (new) Granule's lower and upper bound. That way, even if parts

technically contain data outside of the bounds of the granule they are not returned when iterating over them.

Another transaction ID is obtained to update the sparse granule index with the new granule(s), which is also used to finalize the list of concurrently inserted parts duplicated to the splitted granules.

TODO: How do transactional inserts/updates to the sparse index work? => tl;dr Granule index entries have their transaction ID attached to them.

Further readings

- An Empirical Evaluation of In-Memory Multi-Version Concurrency Control
- Building a Bw-Tree Takes More Than Just Buzz Words
- Apache Arrow vs. Parquet and ORC: Do we really need a third Apache project for columnar data representation?
    - Paper referenced: Integrating Compression and Execution in Column-Oriented Database Systems
    - Some comments to Daniel Abadi's blog about Apache Arrow

Alternatives

Isolation

All transactions blocking each other, allowing all insertions, splitting and compactions to have no need for locking.

Insertion strategy

Insertions into a granule needs to be in order, therefore at insertion time when composite ordered columns are used the index is determined using the union of valid index ranges.

Insert
{namespace="my-namespace-1", pod="my-app-1", container="my-app-test-2"}
Value: 10

Insertion range of namespace is: [0, 3]
Insertion range of pod is (under constraint of namespace): [0, 2]
Insertion range of container is (under constraint of other columns): [1,1]

Therefore the index to insert the new data at is 1.

labels.namespace	labels.pod	labels.container	value
my-namespace1	my-app-1	my-app-test-1	7
my-namespace1	my-app-1	my-app-test-3	2
my-namespace1	my-app-3	my-app-test-3	6
my-namespace2	my-app-1	my-app-test-1	3

Storage mechanisms

- Note: Some of the previously tried and tested storage mechanisms are laid out on a very high level in the abstract and problem statement of this RFC. Over 1.5 years of trying and working on storage mechanisms to solve this problem have led to this point.
- We could use off the shelf columnar databases such as Clickhouse.
    - This approach would require an external dependency which we would like to avoid at all costs for the Parca project. Parca is and should always be a single statically linked binary for ease of use.
- We could use another embeddable columnar store Go library such as https://github.com/kelindar/column
    - kelindar/column was really the only embeddable column store written in Go that we could find and it didn't support the features and flexibility we were looking for.
        - It specifically optimizes for queries, while for Parca the overwhelming majority of interactions with the database are hot-inserts so Parca needs to optimize towards that case.
        - It does not support dynamic dictionary/map types, which is essential to the Parca query model utilizing label-sets.