August 28

KEYSPACE

Amsterdam

Harder, Better, Faster, Stronger: Building Valkey-Timeseries

Clayton Collie

Valkey

Background The Horsehead Nebula and its surroundings. The reflection nebula NGC 2023 in the bottom left corner. / Stephanh / License: CC BY 4.0

Who Am I?

I’m Clayton Collie, an independent developer and long time open-source contributor, now a contributor to the Valkey Project.

github: github.com/ccollie
email: collie.clayton@gmail.com

What Is ValkeyTimeSeries?

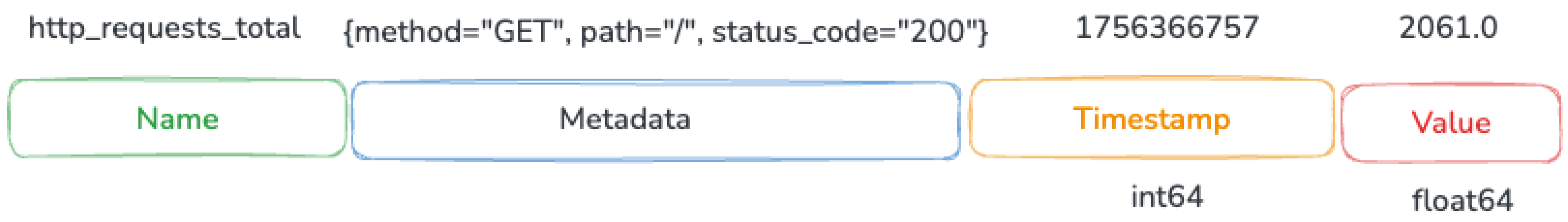
ValkeyTimeSeries is a high-performance, scalable time series database built as a module for Valkey.

- It is designed to handle large volumes of time-stamped data with low latency and high throughput, making it ideal for applications such as monitoring, IoT, and real-time analytics.
- It is intended as a drop-in replacement for RedisTimeseries, with a focus on performance, scalability, and usability.

What is a Timeseries Database?

A timeseries database is a type of database optimized for storing and querying data that is indexed by time.

A timeseries measures `metrics` - an observation of some value at some point in time.



- Name - The name of the metric describes what this metric measures.
- Metadata - Information about the metric represented as label-value pairs.
- Value - The observation itself.
- Timestamp - The time the observation was recorded.

A combination of a metric name and its metadata defines a time series.

What is a Timeseries Database?

You'll find TSDBs at the heart of:

- Monitoring & Observability: Server metrics, application performance monitoring (APM), network data.
- IoT & Sensor Data: Smart home devices, industrial sensor networks, vehicle telemetry.
- Financial Analytics: Stock ticker prices, trading volumes, algorithmic trading.
- Application Analytics: User activity events, clickstreams, ad performance tracking.
- Environmental Data: Weather stations, smart agriculture, energy grid management.

A Quick Introduction to ValkeyTimeseries

Imagine we have a network of temperature sensors in different rooms of a building.

Adding Data

The `TS.ADD` command adds a new sample to a time series. If the time series does not exist, it will be created automatically. As an example:

```
TS.ADD temperature:living_room * 20.5
```

Let's break down this command:

- `temperature:living_room`: This is the key, or name, of our time series.
- `*`: This tells ValkeyTimeSeries to use the current server time as the timestamp for this sample. You can also provide a specific timestamp in milliseconds.
- `20.5`: This is the value of our sample, in this case, the temperature in Celsius.

A Quick Introduction to ValkeyTimeseries

Adding Data

After 10 minutes, we add another reading:

```
TS.ADD temperature:laundry_room * 21.5
```

Let's create another time series for a sensor in the bedroom and add labels to it.

```
TS.ADD temperature:bedroom * 22.1 LABELS room bedroom building main
```

We've added two labels: `room` with the value `bedroom`, and `building` with the value `main`.

We can add multiple samples to multiple series at once using `TS.MADD`.

```
TS.MADD temperature:living_room * 25.8 temperature:bedroom * 22.2
```

A Quick Introduction to ValkeyTimeseries

Querying Data

To get all the data from the `temperature:living_room` series, we can use:

```
TS.RANGE temperature:living_room - +
```

- `-` : Represents the earliest possible timestamp.
- `+` : Represents the latest possible timestamp.

You can also specify a time range. For example, to get the last hour of data:

```
TS.RANGE temperature:living_room -1hr + AGGREGATION avg 3600000
```

This would give us the last 10 entries from the last hour, with an aggregation of the average.

A Quick Introduction to ValkeyTimeseries

Querying with Labels

Labels allow us to query data from multiple time series that share common characteristics. You can use `TS.MRANGE` to query across multiple series based on a filter.
For example, to get the readings from all rooms in the `main` building between January 1 and 2 of 2025 (GMT):

```
TS.MRANGE 1735689600 1735776000 FILTER building=main
```

This will return the data from all time series that have the label `building=main`.

A Quick Introduction to ValkeyTimeseries

Deleting Data

If you need to delete a range of samples, you can use the `TS.DEL` command.

```
TS.DEL temperature:living_room -1hr *
```

This command deletes all samples from the `temperature:living_room` for the last hour.

A Quick Introduction to ValkeyTimeseries

Downsampling and Compaction

Downsampling allows users to create lower-resolution versions of time series data for long-term storage and analysis. For example:

```
TS.CREATERULE temperature:living_room temperature:living_room:1h AGGREGATION avg 3600000
```

creates a downsampled series `temperature:living_room:1h` which is updated with the average every time a new sample is added to `temperature:living_room`.

We can query the downsampled series just like any other time series:

```
TS.RANGE temperature:living_room:1h - +
```

Improvements over RedisTimeseries

Active Expiration

We support active pruning of expired samples in the background. RedisTimeseries prunes lazily on query.

Multi-Db Support

Timeseries can be created in multiple dbs, with proper query isolation. This includes proper support for SWAPDB.

Rounding

Support for rounding sample values to specified precision. This is enforced for all samples in a time series.

Improvements over RedisTimeseries

Dependent Compactions

Support for creating compaction rules based on other compactions.

```
redis> TS.CREATE visitor:count:1m
OK
redis> TS.CREATE visitors:count:1h
OK
redis> TS.CREATE visitors:count:1d
OK

redis> TS.CREATERULE visitors:count:1m visitors:count:1h AGGREGATION sum 1h
OK
redis> TS.CREATERULE visitors:count:1h visitors:count:1d AGGREGATION sum 1d
OK
```

Improvements over RedisTimeseries

Query Filter Enhancements

We support full Prometheus style series selectors (essentially an Instant Vector) in addition to the RedisTimeseries filter syntax. For example:

```
TS.QUERYINDEX latency{region=~"us-west-*",service="inference"}
```

```
TS.QUERYINDEX -6hrs -3hrs request_error_total{status="400", path="/auth", region=~"us-east-?"}
```

Improvements over RedisTimeseries

Query Filter Enhancements

We also support `"OR"` matching for Prometheus style selectors. For example:

```
TS.QUERYINDEX queue{job="app1",env="prod" or job="app2",env="dev"}
```

will return the series with the `{job="app1",env="prod"}` or `{job="app2",env="dev"}` labels.

Improvements over RedisTimeseries

Compaction Policy Filters

Default compactions can specify a filter expression to select which keys they are applied to. For example, certain aggregations (e.g. `min`) are appropriate to gauges and not counters, whereas in RedisTimeseries the defaults are applied to all rules.

```
redis> CONFIG SET ts-compaction-policy avg:2h:10d|^metrics:memory:*;sum:60s:1h:5s|^metrics:cpu:*
OK
```

Improvements over RedisTimeseries

Metadata Commands

We support returning cross-series index metadata (label names, label values, cardinality)

For example, to get the top 10 label names for series matching a filter:

```
> TS.LABELNAMES LIMIT 10 FILTER up process_start_time_seconds{job="prometheus"}

1) "__name__",
2) "instance",
3) "job"
```

Developer Ergonomics

Support for relative timestamps in queries, e.g. `TS.RANGE key -6hrs -3hrs`, unit suffixes (e.g. `1s`, `3mb`, `20K`), and a more expressive query language.

Improvements over RedisTimeseries

Joins

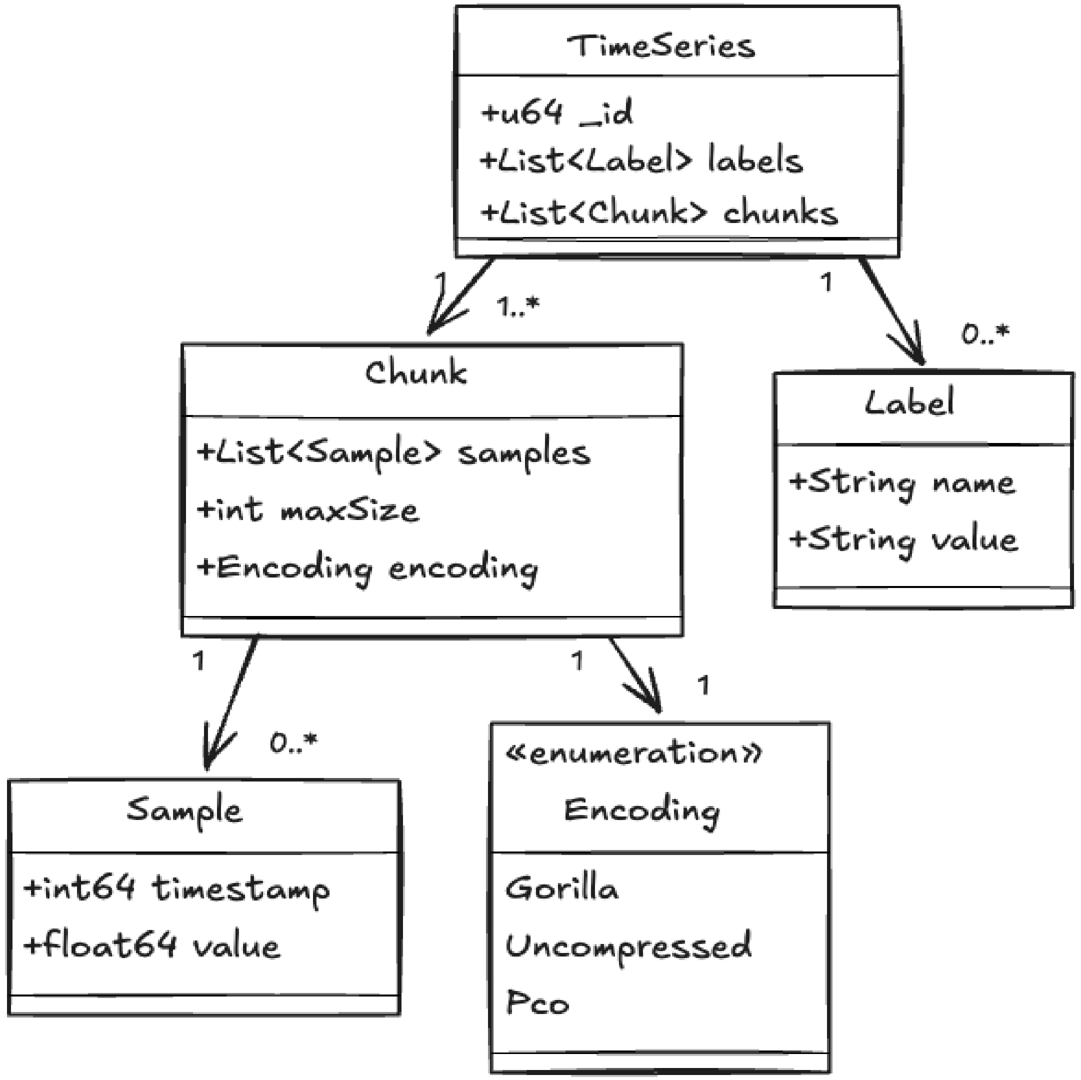
ValkeyTimeSeries supports joins between time series, including INNER, OUTER, and ASOF joins

For example, in a hypothetical trading app tracking buys and sells of various assets

```
TS.JOIN trades:BTC:buy trades:BTC:sell -1hr * ASOF NEAREST 2ms REDUCE sub
```

would calculate the spread between buys and sells of `BTC` occurring over the last hour, matching buy/sell trades which happen within 2 milliseconds of each other.

The Timeseries Structure



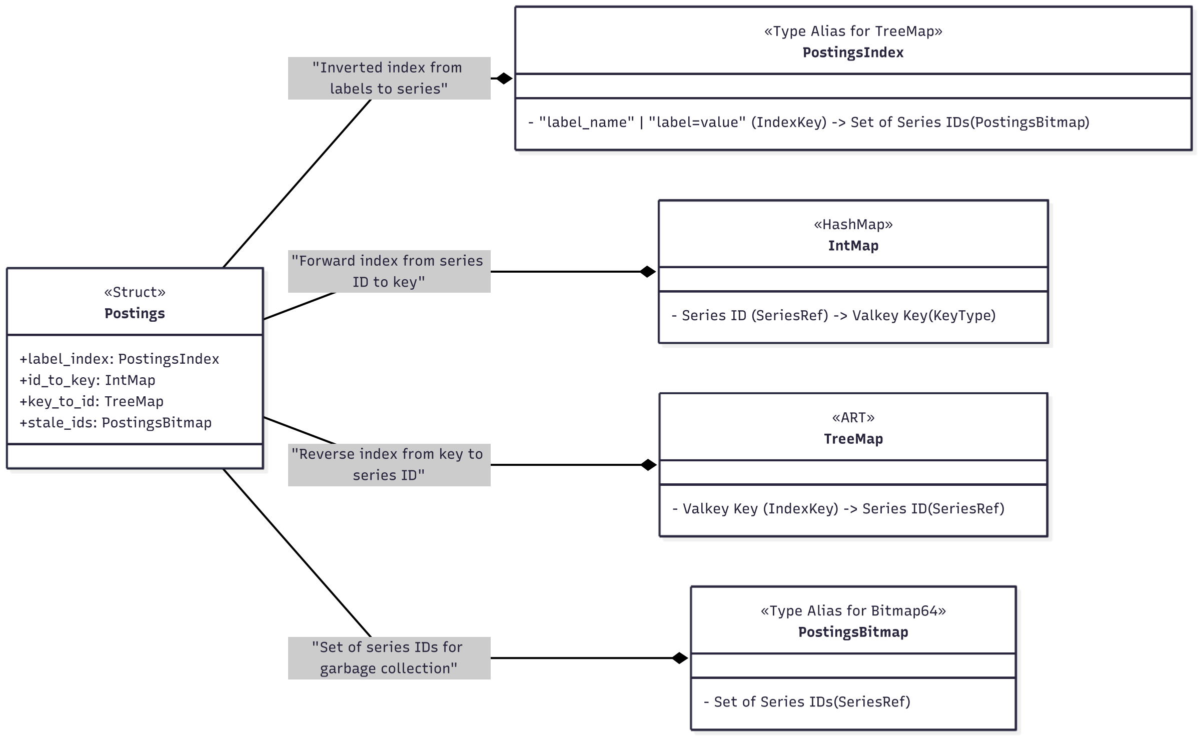
Timeseries Structure: Definitions

Chunks: A list of Chunk objects - ordered containers for samples with non-overlapping time intervals. Samples data can be stored uncompressed or encoded for memory efficiency.

Labels: A list of Label objects, providing metadata about the time series. Each label consists of a name-value pair.
Sample: a timestamped value (a 64bit timestamp and a 64bit float)

Timeseries Indexes

ValkeyTimeseries uses an inverted index to efficiently query time series by labels.

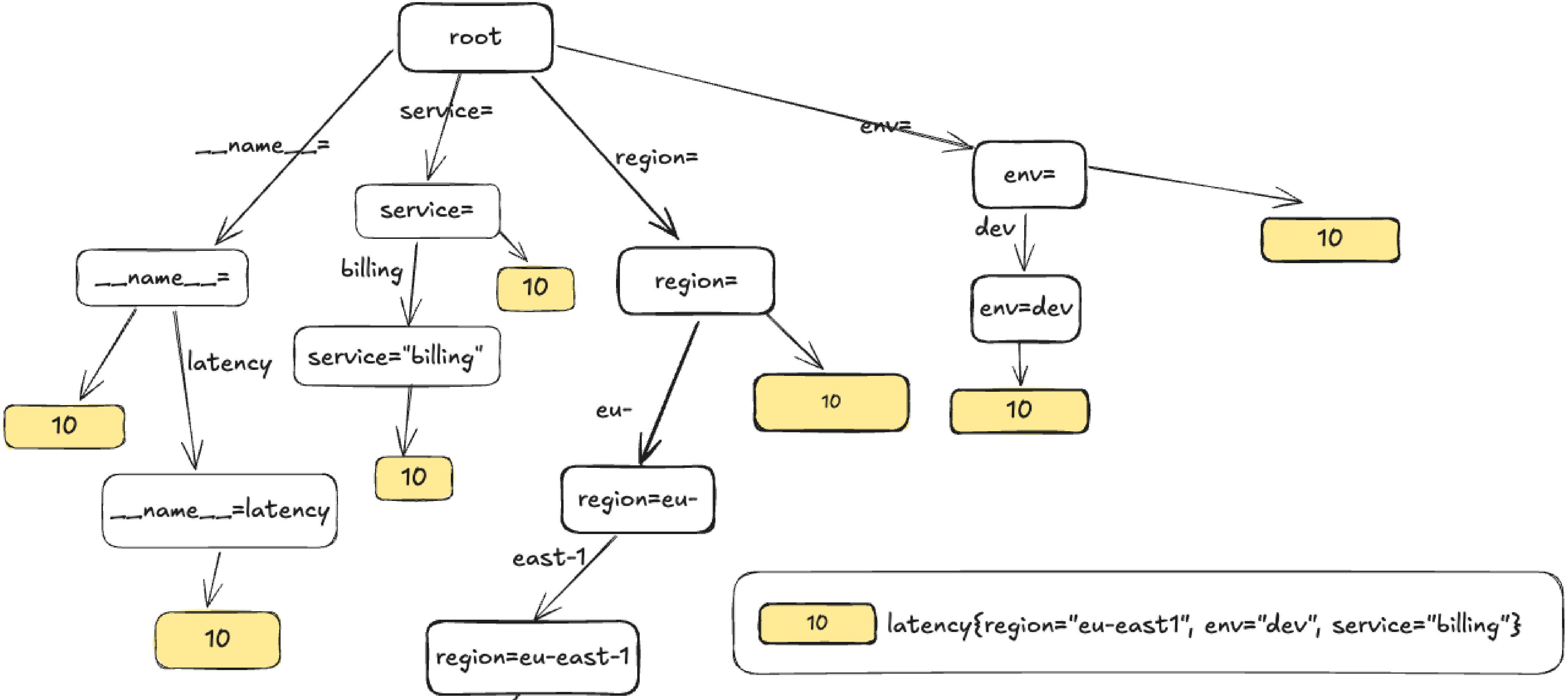

«Struct»
Postings
+label_index: PostingsIndex
+id_to_key: IntMap
+key_to_id: TreeMap
+stale_ids: PostingsBitmap
"Inverted index from labels to series"
"Forward index from series ID to key"
"Reverse index from key to series ID"
"Set of series IDs for garbage collection"
«Type Alias for TreeMap»
PostingsIndex
- "label_name" | "label=value" (IndexKey) -> Set of Series IDs(PostingsBitmap)
«HashMap»
IntMap
- Series ID (SeriesRef) -> Valkey Key(KeyType)
«ART»
TreeMap
- Valkey Key (IndexKey) -> Series ID(SeriesRef)
«Type Alias for Bitmap64»
PostingsBitmap
- Set of Series IDs(SeriesRef)

TimeSeries Indexing

A time series is uniquely identified by an opaque unsigned 64bit int. Each label-value pair is mapped to the id of each series which contains that attribute. The mapping is implemented as an Adaptive Radix Tree (ART) (pdf), where each node is a 64bit Roaring BitMap.

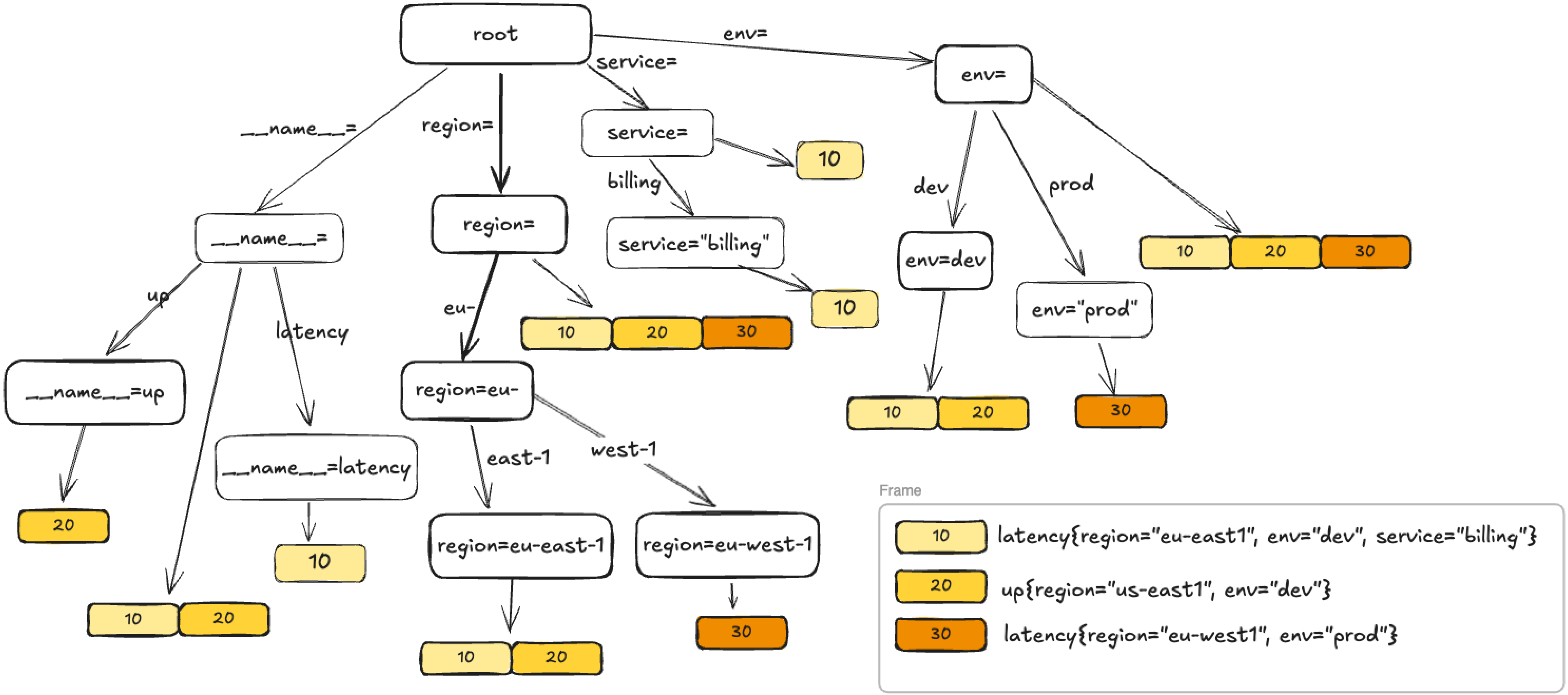
TimeSeries Indexing

Indexing Example



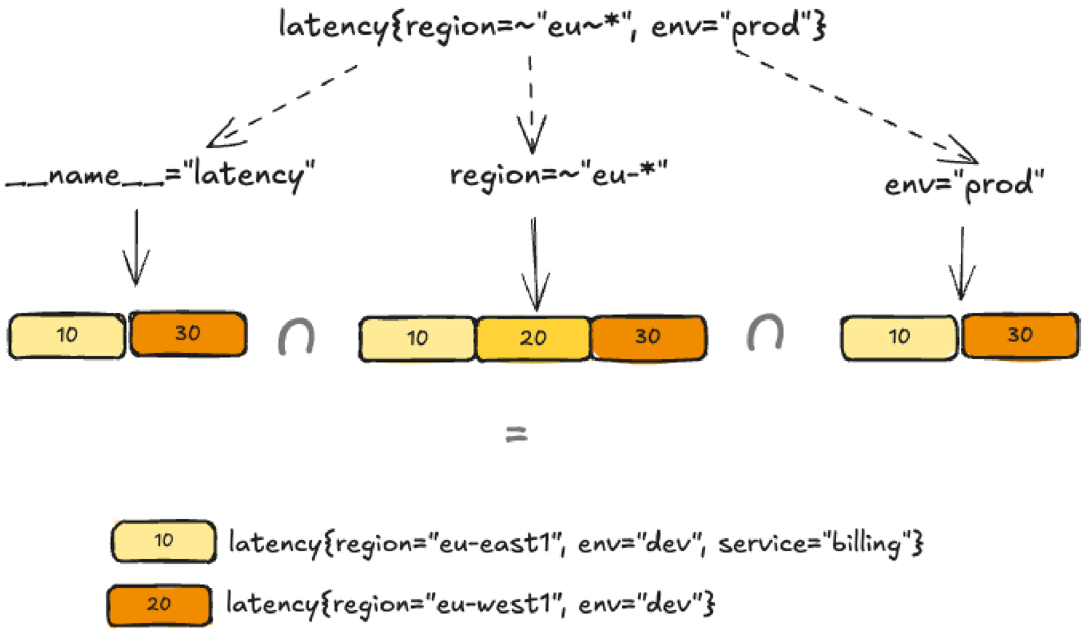
TimeSeries Indexing

Indexing Multiple Series



TimeSeries Indexing

Query Example



latency{region=~"eu~*", env="prod"}

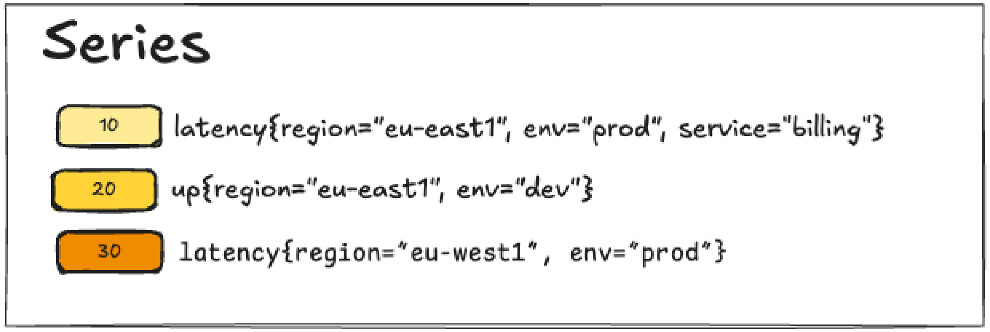
__name__="latency"

region=~"eu-*"

env="prod"

10 latency{region="eu-east1", env="dev", service="billing"}

20 latency{region="eu-west1", env="dev"}



Series

10 latency{region="eu-east1", env="prod", service="billing"}

20 up{region="eu-east1", env="dev"}

30 latency{region="eu-west1", env="prod"}

Performance

Query Parallelization

ValkeyTimeseries parallelizes query operations across series and chunks.

Example

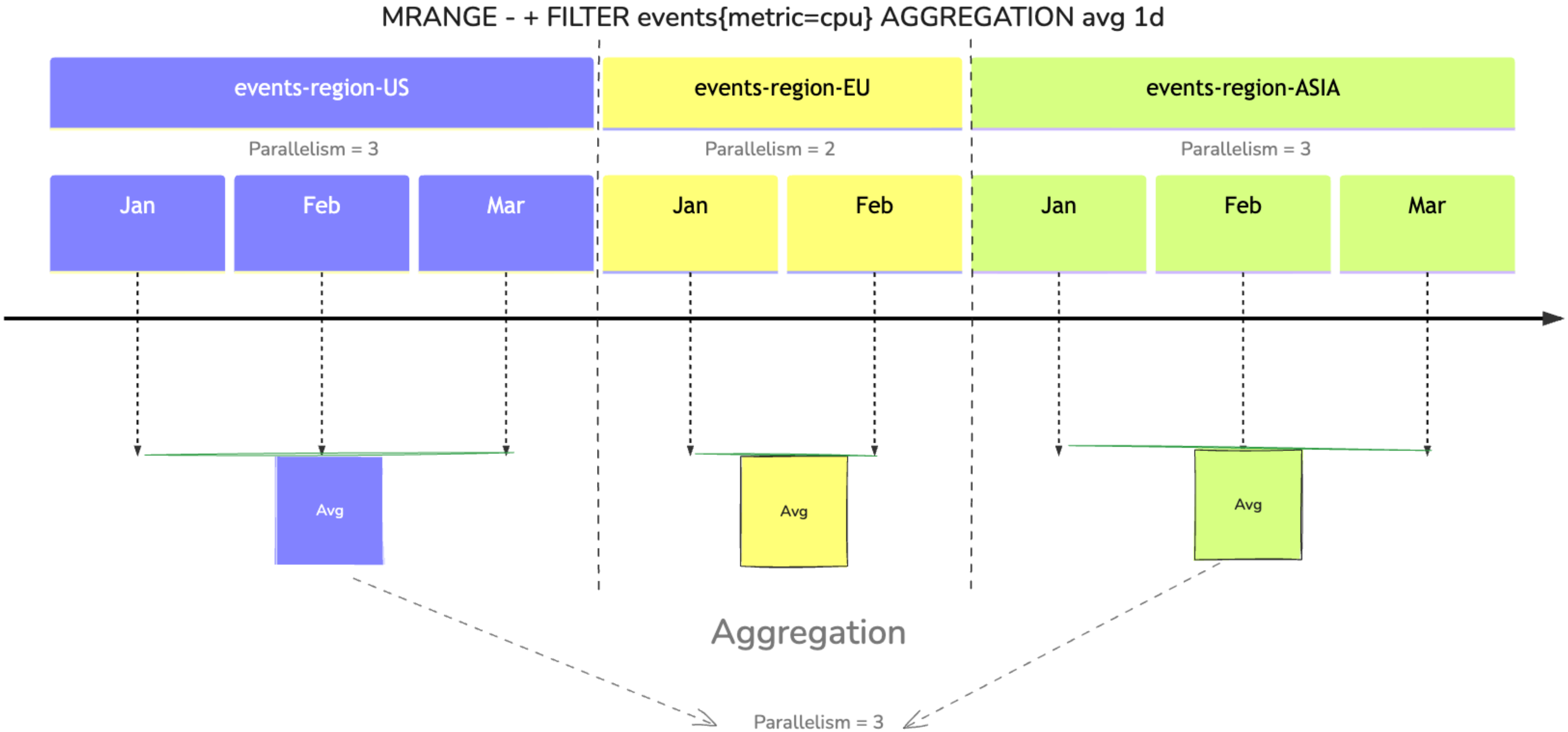
As a demonstration, let's consider the following scenario:

- We have event data stored in 3 timeseries representing 3 regions in chunks of 1-month each.
- We want to get a quarterly average of the values across all regions.
- We use compressed chunks (default) for memory efficiency

This involves scanning all 3 series and aggregating the data.

Performance

Query Parallelization - Example



Performance

Faster Ingestion with Input Batching

As a reminder, you can add multiple samples at once using TS.MADD:

```
TS.MADD key1 1609459200000 42.0 key2 1609459200000 36.5 key1 1609459260000 43.0
```

We process `TS.MADD` in parallel across multiple threads.

- **Chunk processing**: Samples are grouped by the chunks they belong to
- **Parallel chunk operations**: Processes multiple compressed chunks simultaneously
- **Bulk operations**: Uses a merge operation instead of individual `add` calls to reduce per-sample overhead

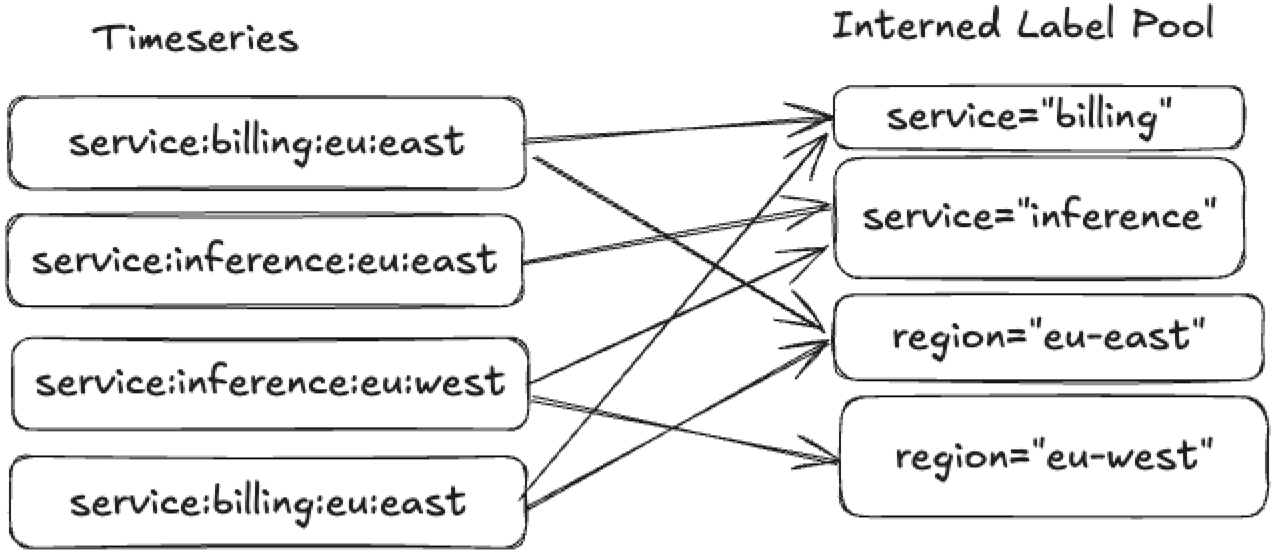
Performance

Downsampling Efficiency

When the parent time series changes, compaction rules are processed in parallel by worker threads,
ensuring that downsampling has minimal impact on real-time data ingestion and querying.

Memory Efficiency

String Interning



Instead of storing multiple identical label-value pairs in memory, a single copy of each unique pair is shared among all timeseries with that combination.

Memory Efficiency

String Interning

For typical workloads, as db would contain:

- **Label Names**: Often only a small number of unique label names over even thousands or millions of series
- **Label Values**: Common values like environment names, service names, and region identifiers are heavily repeated

In practice, string interning can reduce memory usage by:

- **50-90% reduction** in label storage overhead

Repository

repo: https://github.com/ccollie/valkey-timeseries

rfc: https://github.com/ccollie/valkey-rfc/blob/main/TimeSeries.md