> Embedded Time Series Storage: A Cookbook

Andrey Pechkurov



About me

- Java (for a long time) and Node.js (for quite a long time) developer
- Node.js core collaborator
- Interests: web, system architecture, distributed systems, performance
- Can be found here:
 - https://twitter.com/AndreyPechkurov
 - https://github.com/puzpuzpuz
 - https://medium.com/@apechkurov

hazelcast IMDG

- Hazelcast IMDG Management Center (MC)
- Monitoring & management application for IMDG clusters
- Supports stand-alone and servlet container deployment
- Self-contained application, i.e. .jar file and Java is everything you need to run MC
- Frontend part is built with TypeScript, React and Redux
- Backend part is built with Java, Spring and IMDG Java client

Agenda

- A quick intro
- The problem
- Considered options
- Decisions made
- Results and plans

> A quick intro



Terminology

Metric - a numerical value that can be measured at particular time and has a real world meaning. Examples: CPU load, used heap memory. Characterized by name and a set of tags.

Data point - a metric value measured at the given time. Characterized by metric, timestamp (Unix time) and a value.

Types of metrics

- Gauge (e.g. CPU load, memory consumption)
- Counter (e.g. number of processed operations)
- Histogram (e.g. operation processing latency) not supported yet

What we mean by "time series"

"Time series" (TS) stand for series of metric data points

```
class DataPoint {
    String metric;
    List<Map.Entry<String, String>> tags;
    long time;
    long value;
}
```

Sample data point

metric

tags

time

value

map.totalGetLatency

name=test-map,unit=MS

1532689094000

10234

Simple math

```
10 members
6,000 metrics per each
3 sec interval
```

Simple math

```
10 members
6,000 metrics per each
3 sec interval
```



20K data points per sec 1,728M data points per day 27.6GB of raw data (time + values)

Summary

Time series data (usually) implies:

- Lots of writes. Thus, large data volume
- Significantly less reads
- Raw and aggregate queries

Storage formats

- Columnar storage format
- Log-structured merge-tree (LSM tree)
- B-tree
- Their variations

Columnar format

LSM tree

Data compression

- Integer compression
 - Delta encoding
 - Delta-of-delta encoding
 - Simple-8b
 - Run-length encoding
- Floating point compression
 - XOR-based compression
- Type-agnostic compression
 - Dictionary compression

> The problem

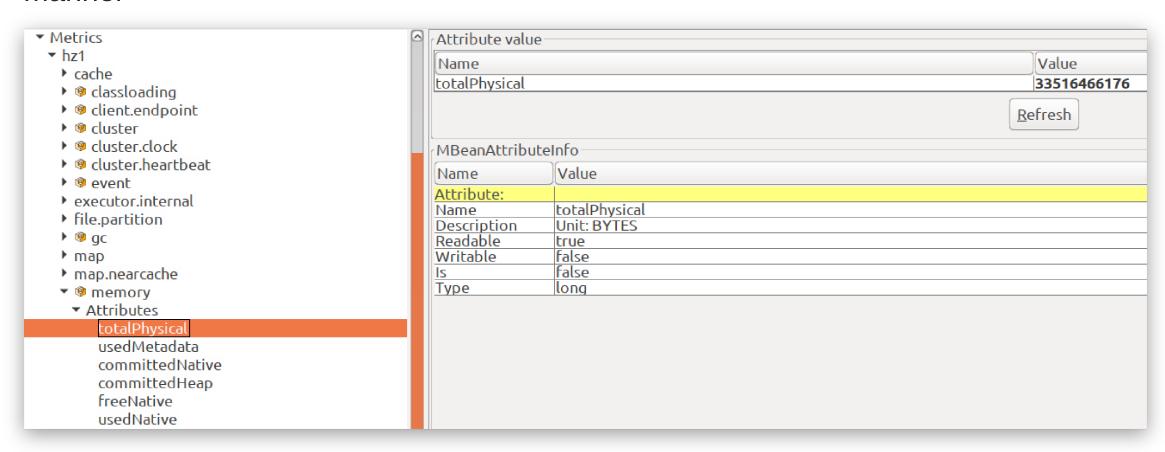


The problem

- In the past IMDG clusters were reporting their metrics as a large JSON object
- MC was storing collected JSONs into a key-value storage (in-memory and/or JDBM)
- Such approach has some downsides that are critical for us
- Say, it requires changes in many places when we had to add new metrics

The solution

IMDG v4.0+ is capable of reporting collected metrics (probes) to MC in a generic manner



The challenge

- MC has to store those metrics somehow
- Thus, we need a Time Series Storage
- Here comes the challenge...

Requirements - must haves

- Embedded time series database or storage
- In-memory and optional persistent modes
- Data compression to achieve low disk footprint in the persistent mode
- Support data retention to avoid running of disk space
- Durability and fault tolerance in the persistent mode

Requirements - nice to haves

- Good write performance (100Ks data points/second on average HW)
- Good enough read performance (10Ks data points/second on average HW)
- Use existing stable SW, when possible

Considered options

TS DBs

- OpenTSDB
- InfluxDB
- TimescaleDB
- Prometheus
- ClickHouse
- Kdb+
- Graphite
- etc.

Embedded TS DBs/storages

- Akumuli (C++) https://akumuli.org
- QuestDB (Java) https://www.questdb.io

Embedded non-TS DBs/storages

- SQL DBs
 - ∘ H2 DB (B-tree)
- Key-value storages
 - H2's MVStore (B-tree)
 - MapDB (HTree, B-tree)
 - RocksDB (LSM tree)

Decisions made



Initial ideas

After initial research and experiments we decided the following:

- Build a TS storage on top of a key-value storage
- Keep the storage API simple

Primitive data layout

key value

1532689094000@map.totalGetLatency@name=test-map,unit=MS

10234

Draft API

```
public interface MetricsStorage extends AutoCloseable {
    void store(Collection<DataPoint> dataPoints);
    DataPointSeries queryRange(Query query);
    Optional<DataPoint> queryLatest(Query query);
}
```

TODO list

- 1. Choose one of embedded key-value storages
- 2. Come up with a way to reduce number of persisted entries
- 3. Think of sufficient data compression for the persisted data

Item 1: embedded key-value storage

After some experiments we picked up two candidates

- MapDB (Java)
- RocksDB (C++ with JNI bindings)

Item 2: number of persisted entries

- We need to group multiple data points into a single entry somehow
- What if we store data points in buckets? Say, a bucket per minute

Bucketed data layout

key value

1532689080000@map.totalGetLatency@name=test-map,unit=MS

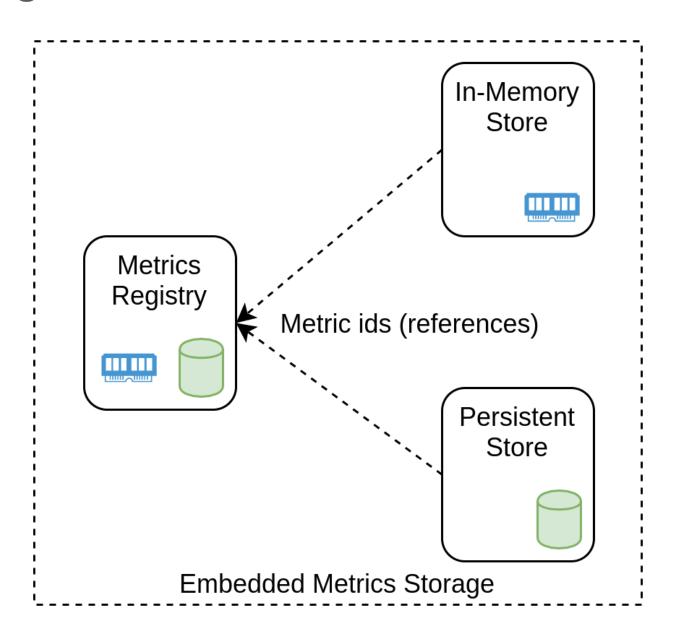
[10234,...,10480]

minute start new long[60]

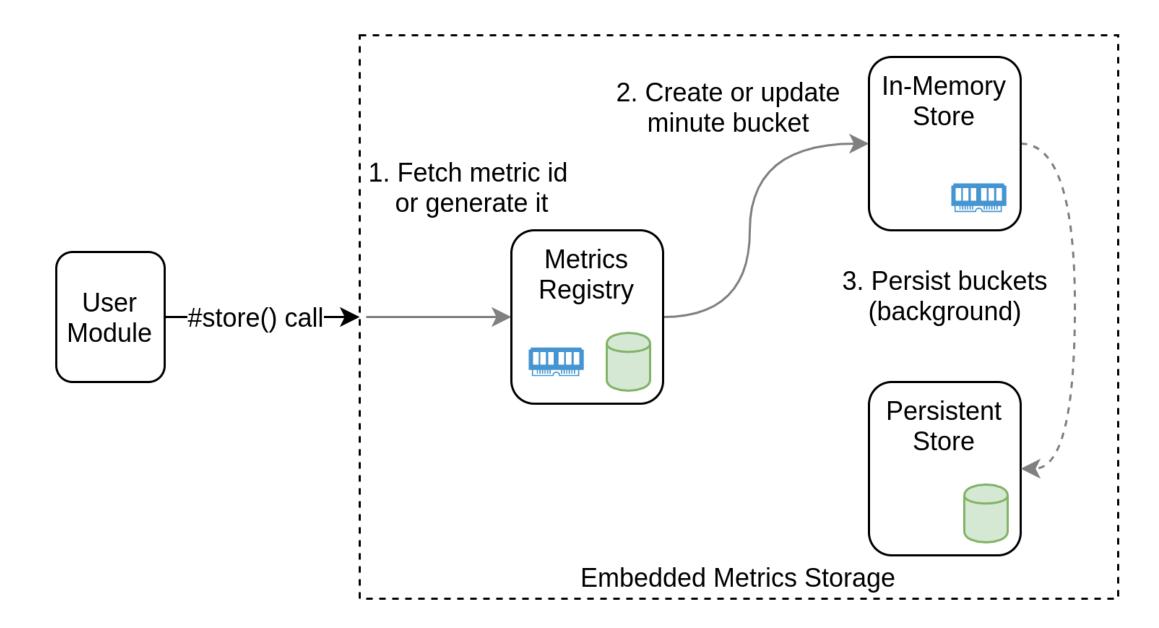
Item 3: data compression

- Keys
 - We could use dictionary compression for metrics
- Values
 - For each minute bucket we could use compression methods for integer numbers, like delta encoding

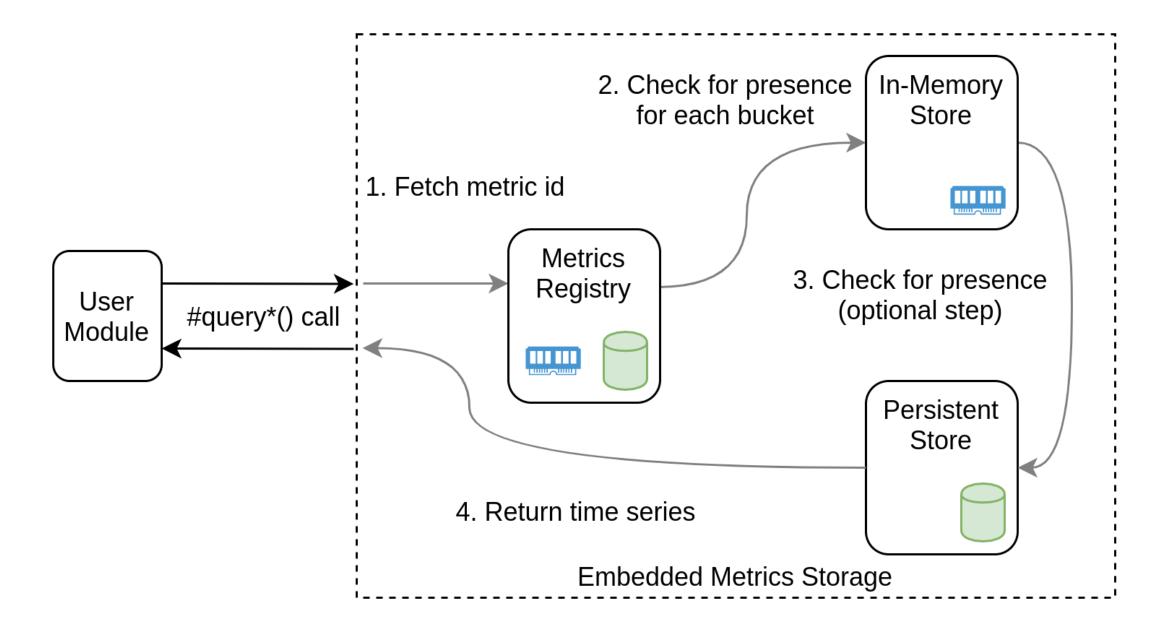
Overall design



Writes



Reads



Metrics Registry

key value

map.totalGetLatency@name=test-map,unit=MS

42

metric name + tags

metric id (int)

Data compression: keys

old key format

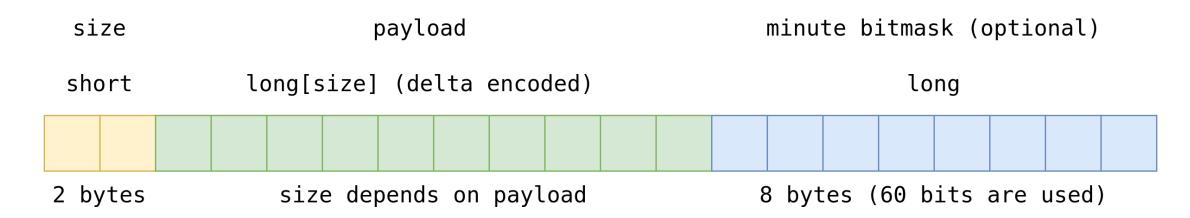
1532689094000@map.totalGetLatency@name=test-map,unit=MS



new key format

1532689094000@42

Data compression: values



Values compression efficiency

Scenario	Raw* (bytes)	Delta compressed* (bytes)	Compressed (bytes)	Ratio (vs. Raw)
Const int (3 sec)	480	360	13	x37
Random int (3 sec)	480	364	156	x3

^{*} Long.MIN_VALUE is used to represent missing values

Other features

- Data retention
 - Based on per entry time-to-live (TTL) in RocksDB
- Data durability
 - Pending minute buckets are persisted on graceful shutdown
- Aggregation API
 - Built on top of the storage

> Results and plans

Benchmark results

- Scenario:
 - Emulates 10 members, 120,000 metrics, 3 second interval
 - Random values from 0-1000 range
- Writes* 400K data point/sec
- Random minute series reads* 19K ops/sec
- * Results were obtained on a laptop

Further plans

- Support additional indexes for metrics
- Expose diagnostics information in runtime
- Perform additional testing and optimization



Call to action

- You may want to give a try with IMDG and MC: https://hazelcast.org/
- Open source contributions are welcome as well!

Thank you!



Helpful links

- https://docs.hazelcast.org/docs/4.0.1/manual/html-single/index.html#metrics
- https://blog.timescale.com/blog/time-series-compression-algorithms-explained/