> 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





#### https://hazelcast.org

- Hazelcast In-Memory Data Grid (IMDG)
- Open-source distributed in-memory object store
- Supports a variety of data structures such as Map, Set, List, and so on
- Implemented in Java
- Supports embedded и standalone modes
- Has client libraries for many languages and platforms



# hazelcast IMDG

- Hazelcast IMDG Management Center (MC)
- Monitoring & management application for IMDG clusters
- Supports standalone and servlet container deployment
- Self-contained application, i.e. .jar file and Java is everything you need
- 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 set of tags\*.

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

\* We'll use term "metric" instead of "metric + tags".



### **Types of metrics**

- Gauge (e.g. CPU load, memory consumption)
- Counter (e.g. number of processed operations)
- Histogram (e.g. operation processing latency)





### What we mean by "time series"

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

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



### Sample data point

metric

tags

time

value

memory.usedHeap

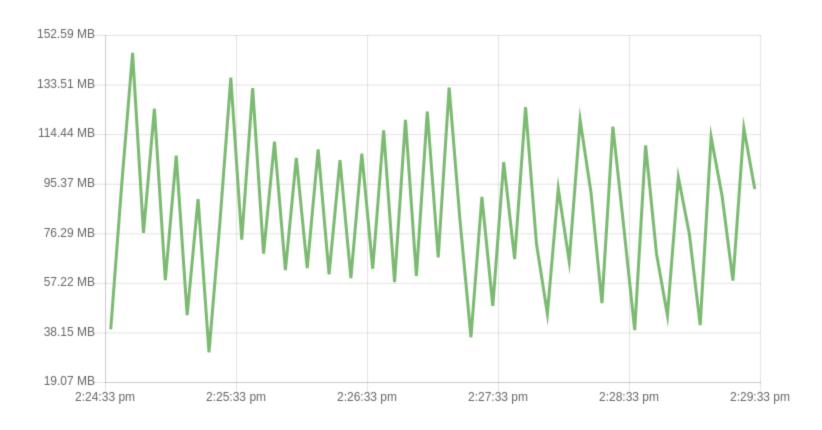
unit=BYTES

1532689094000

136314880



# Sample time series





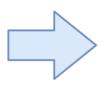
## Simple math

```
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:

- Large data volume
- Writes:
  - Lots of writes
  - Writes involve data points for all metrics or most of them
- Reads:
  - Significantly less reads
  - Reads assume one or several time series over a time range
- Raw and aggregate queries



> The problem





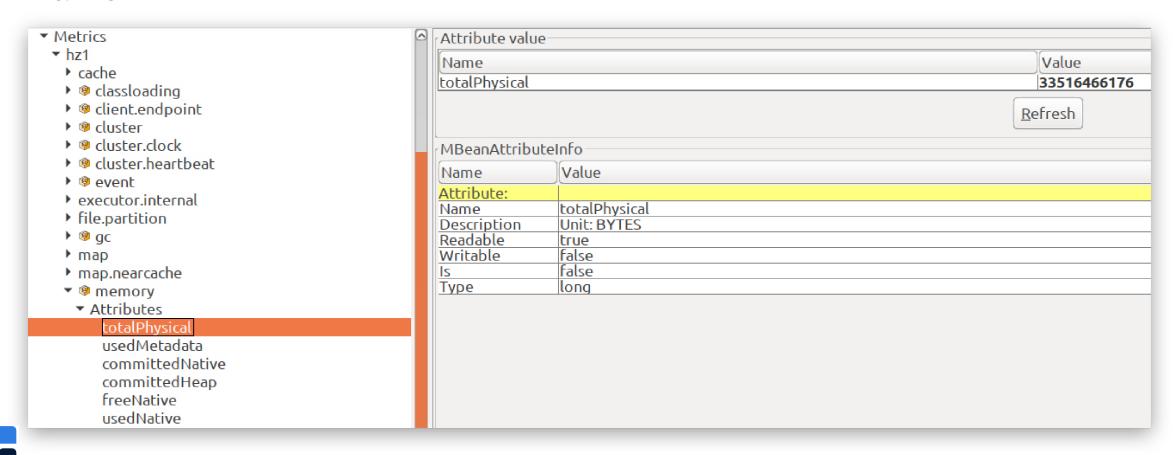
### The problem

- 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 database (or, at least, 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
- and many more













### **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 {
    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)

RocksDB won the battle in the end.



### 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, each bucket will contain data points within a minute



### **Bucketed data layout**

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

{10234,...,10480}

minute start

new long[60]

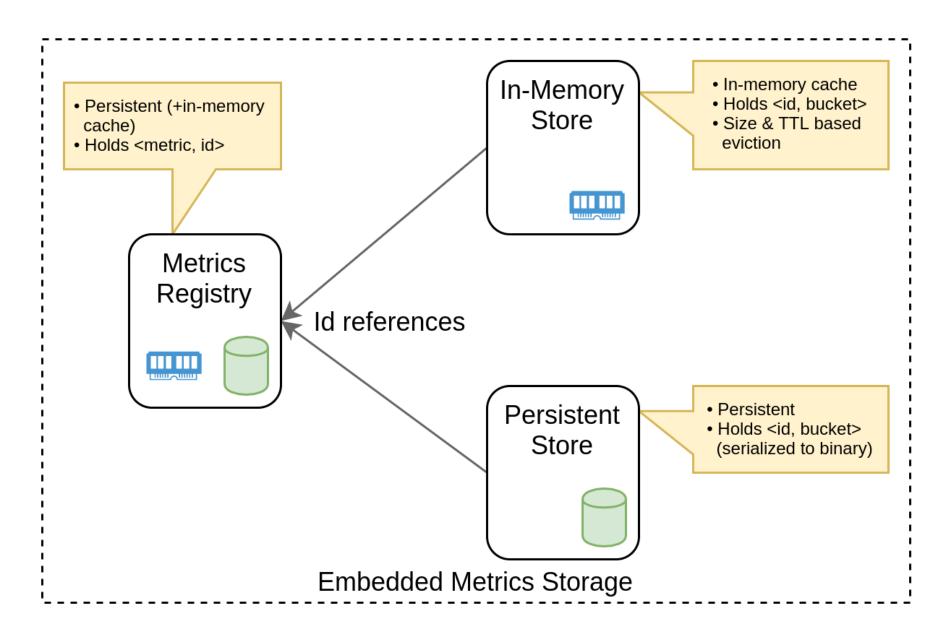


### **Item 3: data compression**

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

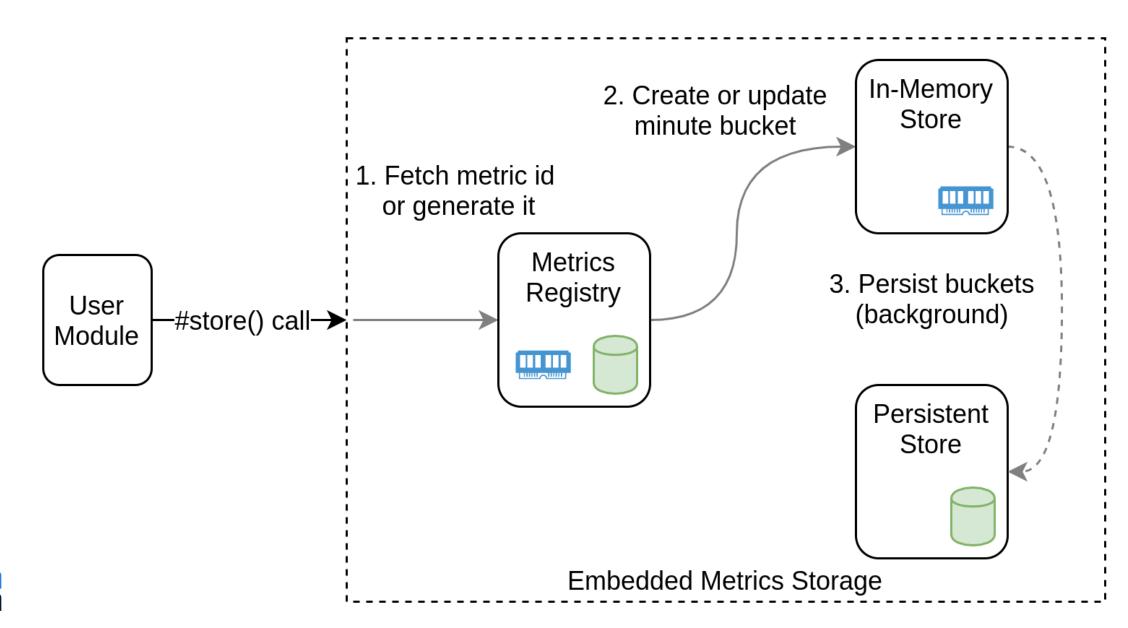


### Overall design



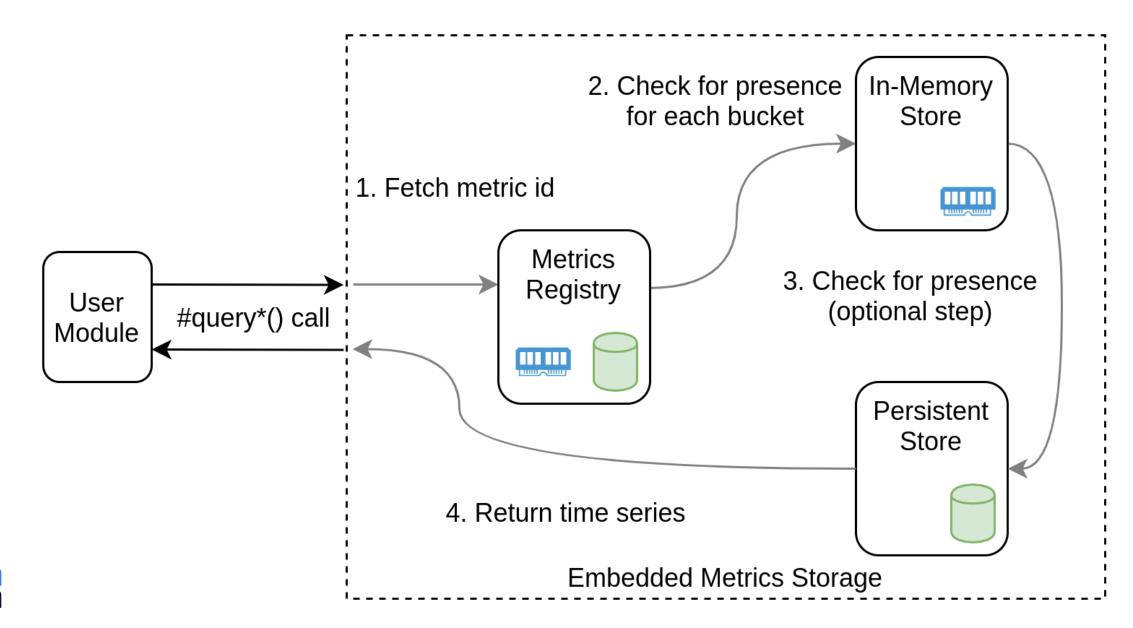


#### Writes





#### Reads





### **Metrics Registry**

key

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

metric name + tags

42

value

metric id (int)



### **Data compression: keys**

old key format

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

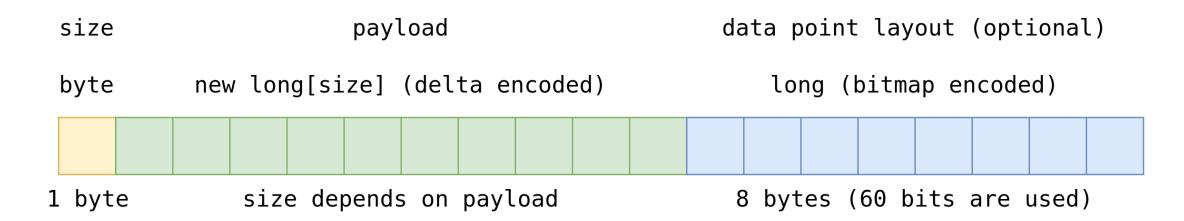


new key format

1532689094000@42



## Data compression: values





### Sample compressed value

payload data point layout

2 {10,+3} Ob1010...0



{10, Long.MIN\_VALUE, 13, ..., Long.MIN\_VALUE}



## Value compression efficiency

Scenario	Raw (bytes)	Compressed (bytes)	Ratio
Const long (1 sec)	480	12	x40
Const long (3 sec)	160	19	x8.4
<pre>Inc long (3 sec; &lt; byte )</pre>	160	55	x2.9
<pre>Inc long (3 sec; &lt; short )</pre>	160	93	x1.7
Random long (3 sec)	160	170	x0.9



### Disk cost of each data point

	Cost (bytes)
Raw	16*
Metrics Storage	5.25**
Prometheus	3.3
Gorilla	1.37

<sup>\*</sup> Binary size of timestamp + value



<sup>\*\*</sup> We consider "Inc long (3 sec; < short )" scenario

#### Other features

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



### **Design restrictions #1**

Problem:

Data point loss when in-memory cache size is insufficient

Potential solution:

Adapt the size dynamically



## **Design restrictions #2**

Problem:

Loss of last minute data on force shutdown

Potential solution:

Additional WAL or flush data on more frequent checkpoints



### **Design restrictions #3**

Problem:

Out of order writes (>1 minute time window)

Potential solution:

Merge buckets during background persistence



> Results and plans

### Simple benchmark

#### Scenario:

- Emulates 10 members, 120K metrics, 3 second interval
- Random values from 0-1000 range



#### **Benchmark results\***

- Write throughput:
  - 400K data point/sec
- Read throughput (random minute series reads):
  - ∘ 19K ops/sec
- \* Results were obtained on a laptop



### **Further plans**

- Add support for additional indexes over metrics
- Implement downsampling
- 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/
- Gorilla: a fast, scalable, in-memory time series database, 2015 https://dl.acm.org/doi/10.14778/2824032.2824078
- https://github.com/facebook/rocksdb/wiki/Leveled-Compaction
- https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2schema-design-best-practices





### **Bonus 1: storage formats**

- Column-oriented storage
- Log-structured merge-tree (LSM tree)
- B-tree
- Their variations and combinations

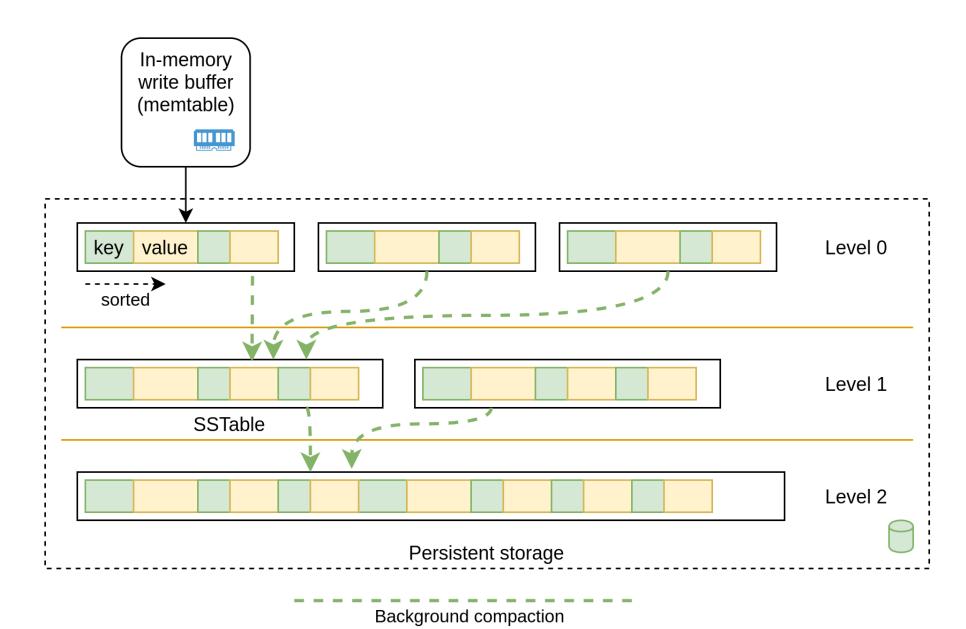


# **Column-oriented storage**

	File 1	File 2	File 3	
	time	cpu_load	memory	
Row 1	1532689094000	0.32	104857600	
	1532689096000	0.51	136314880	
Row N	1532689100000	0.63	137122810	
	1532689098000	0.75	158334976	



### LSM tree



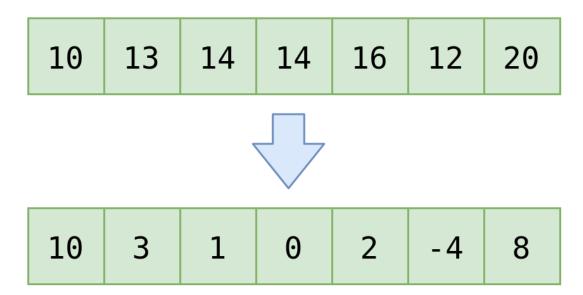


### **Bonus 2: data compression**

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

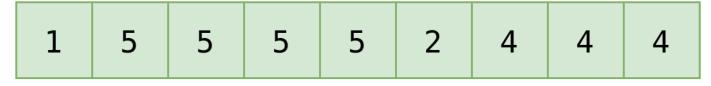


## **Delta encoding**





## **Run-length encoding**

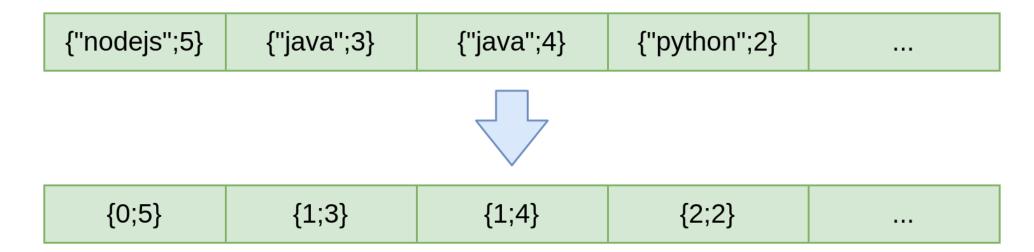




<i>{</i> 1;1 <i>} {</i> 4;5 <i>} {</i> 1;2 <i>} {</i> 3;4 <i>}</i>
--



### **Dictionary compression**



```
dictionary

{
    0: "nodejs",
    1: "java",
    2: "python",
    ...
}
```



### Bitmap encoding

City

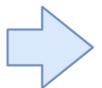
New York

Moscow

Moscow

Istanbul

. . .



New York

Moscow

Istanbul

#### Bitmap

1	0	0	0	
0	1	1	0	
0	0	0	1	



Row 1

