# Process Historian with Cassandra: 24 hour challenge

***Abstract*—Process historian is a time-series database that stores readings from, for example, SCADA devices, IoT sensors, and the like. Such database should be scalable, durable (should have certain resilience to node failures) and support fast write operations. In this short document we describe our 24 hour challenge in building such database using Cassandra NoSQL, masterless database. We use Python Flask as a IoT facing web server. The web server implements simple REST API for the integration with IoT devices.**

## I. Introduction

Process Historian is a must in modern IoT applications: such database is used for storing the time-series readings from various sensors. Process Historian should hold the following properties: (i) it should be durable to node failures, (ii) it should be scalable horizontally, (iii) it should guarantee fast writes to the disk, (iv) it should have clean design.

In what follows we present MVP for such database that uses Cassandara NoSQL database and Python Flask user facing web service. The solution is a result of 24 hours challenge. The source codes for our implementation can be found in [1].

In Figure 1 we show rather abstract architecture of our deployment. Thus in the setup we had 4 nodes deployed in the DigitalOcean cloud: (i) 3 nodes for Cassandra cluster; (ii) MySQL, Nginx, and single REST API server were deployed on single computing node; (iii) we had multiple data generators running on local machine.

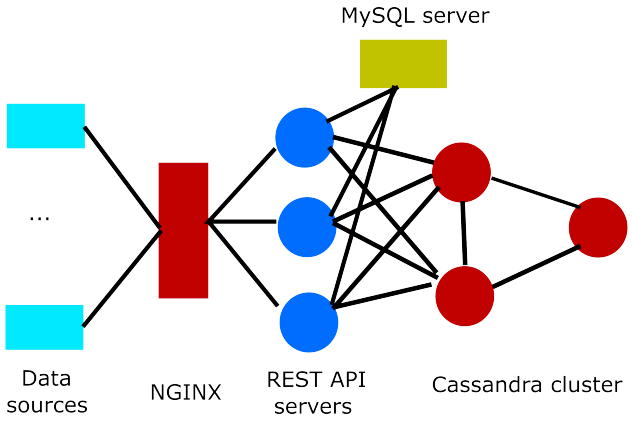


Fig. 1: Process Historian Architecture

The Cassandra row design is depicted in Figure 2. It is basically consists of a wide row (1 day long) and a composite partition key: tag plus date bucket (date bucket splits the data into 1 day periods). The parameters for Cassandra clusters are shown in Table I.

The remaining parameters where left unchanged and were configured during the release of Cassandra 4.1.9 by the developers.

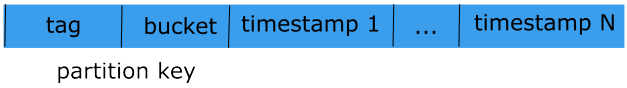


Fig. 2: Cassandra schema design

Overall, our MVP includes the implementation of the following functionality: (i) sensor metadata manipulation (addition, removal, and modification of the sensor information in the MySQL database), (ii) raw sensor data injection (we have also implemented a simple authentication layer, according to which every data source was creating an HMAC for each tag (this information was verified at the REST API server using the secret key it shared with particular sensor), (iii) raw sensor data retrieval using tag name, start, and end periods for filtering.

We should note that it is rather trivial to also add data rollup, aggregation (such as finding mean, max, min, etc.) and extrapolation (such as time-series prediction) features into the code. But all this was out of the scope in this study.

## II. Data collection methodology

To generate the load, we have implemented two different clients: (i) data writer and (ii) data fetcher. The writer was generating data for $n$ tags and pushing into the cloud over the Internet in batches of size 1000 samples per batch. The writer was concurrently pushing data for $n$ tags (one thread per tag). Here we were logging such parameters as batch start and end time, batch number, and tag (this allowed us to calculate the performance later on). Data fetcher was querying the data for $m$ tags for random 1-hour period. Here we logged start time, end time, iteration number, and tag name. Again, such information allowed us to calculate the performance for various numbers of tags. We did not experiment with the mix load when data was simultaneously written and read from the system.

## III. Data processing and basic results

In what follows, however, we will describe the performance of only read and write raw sensor data operations. The rest of the functionality is out of the scope.

We start with the data ingestion. Here, using the data source, we were generating data points for random 1-hour period, and we were writing this data in batches. We have used Python threads to simultaneously write data for 1, 10, and 50 tags (sensors) at the same time. We computed the average number of samples per second and computed 95% confidence intervals.

The results for write performance together with the 95% confidence interval are shown in Figure 3.

TABLE I: Cassandra parameters

| Parameter | Value |
|---|---|
| Number of nodes in the cluster | 3 |
| Replication factor | 2 |
| Node RAM size | 2 GB |
| TTL | 1 year |
| Number of seed nodes | 1 |
| Partitioner | uniform (Murmur3Partitioner) |
| Read and write consistency | quorum |

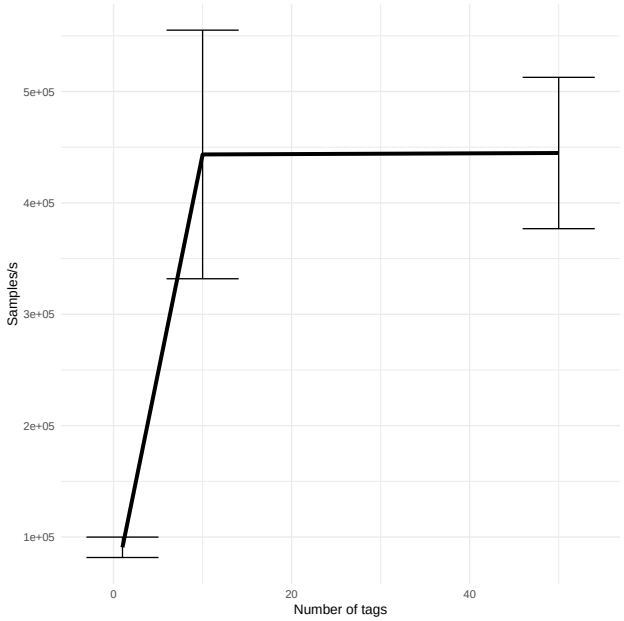


Fig. 4: Number of datapoints read per second with 95% confidence interval

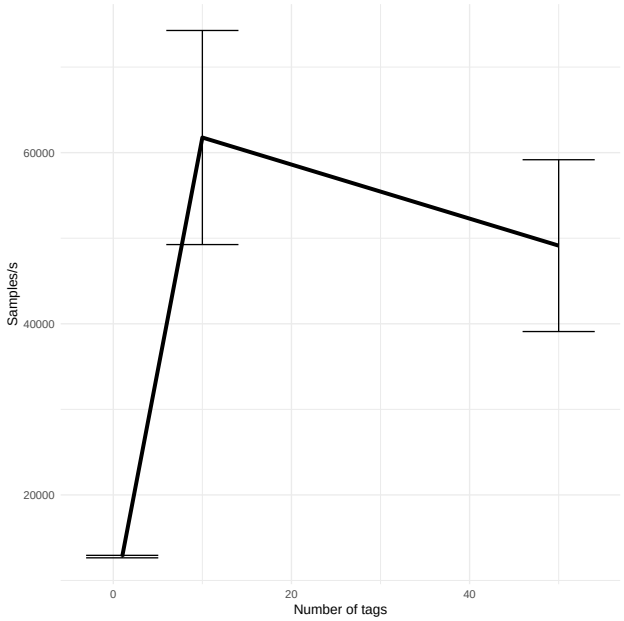


Fig. 3: Number of datapoints written per second with 95% confidence interval

The same results but for the read performance are shown in Figure 4.

The results are rather awkward - the read is way much faster than write, which is not usual for Cassandra database. We will continue to explore this direction further.

## IV. Conclusions

In this document we have described our effort related to building a process historian database using Python Flask and NoSQL Cassandra database. We have completed the entire challenge in roughly 24 hours. To measure the performance we have experimented with various read and write loads, but not the mixture. All-in-all we believe that the system can be used in rather small setups as it is, but should be well tuned for large scale deployments. Also, we should note that the further analysis is required: (i) we think that the performance should be measured for longer periods of time, and (ii) impact of compaction and data removal should be considered thoroughly. Furthermore, investigation of inadequate write performance deserves a separate attention.

## References

[1] Simple Process Historian database. https://github.com/dmitriykuptsov/process-historian-cassandra.