Performance evaluation of modern time-series database technologies for the ATLAS operational monitoring data archiving service

Matei-Eugen Vasile, Giuseppe Avolio, and Igor Soloviev

***Abstract*—The Trigger and Data Acquisition system of the ATLAS [1] experiment at the Large Hadron Collider [2] at CERN is composed of a large number of distributed hardware and software components which provide the data-taking functionality of the overall system. During data-taking, huge amounts of operational data are created in order to constantly monitor the system. The Persistent Back-End for the ATLAS Information System of TDAQ (P-BEAST) is a system based on a custom-built time-series database. It archives any operational monitoring data published online, resulting in about 18 TB of highly compacted and compressed raw data per year. P-BEAST provides command line and programming interfaces for data insertion and retrieval, including integration with the Grafana platform. Since P-BEAST was developed, several promising database technologies for efficiently working with time-series data have been made available. A study to evaluate the possible use of these recent database technologies in the P-BEAST system was performed. First, the most promising technologies were selected. Then, their performance was evaluated. The evaluation strategy was based on both synthetic read and write tests, and on realistic read patterns (e.g., providing data to a set of Grafana dashboards currently used to monitor ATLAS). All the tests were executed using a subset of ATLAS operational monitoring data, archived during the LHC Run II. The details of the testing procedure and of the testing results, including a comparison with the current P-BEAST service, are presented.**

***Index Terms*—database performance, operational monitoring, time-series databases**

I. Introduction

THE Trigger and Data Acquisition (TDAQ) system of the ATLAS experiment is a complex distributed system made out of a large number of hardware and software components. That means about 3000 machines and in the order of O(10^5) applications working to accomplish the data gathering function of the detector.

During data-taking runs, large amounts of operational data are produced in order to monitor the functioning of the detector. Currently, this data is being gathered and stored using a system called the Persistent Back-End for the ATLAS Information System of TDAQ (*P-BEAST*) [3]. *P-BEAST* is, essentially, a custom time-series database used for archiving operational monitoring data and retrieving the stored data for applications that require it. It stores about 18 TB of highly compressed raw monitoring data per year.

Since *P-BEAST* has been commissioned in 2014, more than a few new time-series database technologies have been released. A preliminary survey has been done in order to identify a short list of the most promising candidates for being evaluated for the purpose of being used as a new back-end for *P-BEAST*.

II. Background

The main requirements for any potential candidate were to support or provide emulation for all the data types currently used by the ATLAS operational monitoring (integers, floats, strings and arrays) and the capability to sustain the data injection rate observed during real data taking runs with *P-BEAST*. This means an insertion rate of approximately 200k metrics/s. A preliminary survey was done among time-series database technologies, columnar database technologies and key-value stores. As a result of the preliminary survey, two technologies were selected:

- **InfluxDB** [4] – a time-series database
- **ClickHouse** [5] – a columnar database

III. Data model and testing setup

ATLAS operational monitoring data are stored using a *class.attribute* data model. The smallest piece of stored data is a time series data point representing the value of an *attribute*. Each stored *attribute* belongs to an *object*.

In the first phase of this research [6], two ways of organizing the stored data have been tested: a single table data organization (see figure 1) and a multiple table data organization (see figure 2).

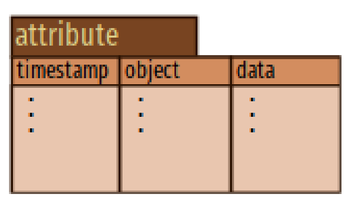

Fig. 1. Single table data organization.

In the single table setup, all the data points for a given *attribute* are stored in a single table which contains an *object*

August 11, 2022

Matei-Eugen Vasile is with the Institutul National de Cercetare-Dezvoltare pentru Fizică si Inginerie Nucleară Horia Hulubei (IFIN-HH), Măgurele, România (e-mail: matei.vasile@cern.ch)

Giuseppe Avolio is with the European Laboratory for Particle Physics, CERN, Geneva 23, CH-1211, Switzerland (e-mail: giuseppe.avolio@cern.ch)

Igor Soloviev is with the University of California, Irvine, CA 92697-4575, US (e-mail: igor.soloviev@uci.edu)

column used as an index. In the multiple table setup, all the data points belonging to an *attribute.object* pair are stored in their own table.

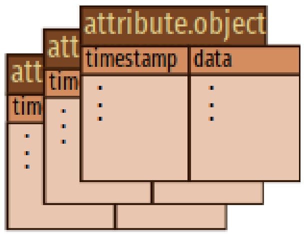

Fig. 2. Multiple table data organization.

The idea behind experimenting with both these approaches to storing the data was to test which one results in a faster write rate. In the end of the first round of testing, the results showed that both approaches have very similar performances, with a very slight advantage of the single table approach. This, combined with the fact that queries were easier to work with in a single table setup, the decision was made for this second round of testing to use the single table setup.

A. Test data

For the write performance testing, the tests have been run using various real *ATLAS* operational monitoring data archived during the *ATLAS Run 2* operation. Out of that *ATLAS* operational monitoring data stored using *P-BEAST*, four data types have been selected as being the most representative ones:

- arrays of 12 float64s
- strings of approximately 5500 characters
- float64s
- int64s

Then, for the read performance testing, the databases that have been created and populated with *ATLAS* operational monitoring data have been used for all read performance tests.

B. Software

The implementation of all tests has been done in the *Go!* programming language, which is the language used for developing *InfluxDB*, thus it has native support for *Go!* clients. For *ClickHouse*, there is *Go!* support via third-party libraries.

C. Hardware

All the tests have been run on a dual-CPU computer with the following specifications:

- 2 Intel Xeon E5-2630 v2 @2.60GHz CPUs (each with 6 cores and hypethreading, for a total of 24 threads)
- 32 GB of RAM
- An 18 TB RAID0 array using hard disk drives

IV. TESTING

A. Write testing

The first batch of tests were the write tests. These have been developed to fetch batches of 10000 data points from the existing *ATLAS* operational monitoring database using *P-BEAST*, and then write those batches into the prepared *InfluxDB* and *ClickHouse* databases.

P-BEAST stores data using an in-house format based on *Google Protocol Buffers*. Every data file stores time-series data for many objects of a single attribute. Inside a file the data are indexed by object name. The data files are compacted and compressed weekly. The *P-BEAST* measurements have been performed to serve as a baseline for the performance measurements of the other technologies.

InfluxDB: each object is stored in a *measurement* (*InfluxDB* table) - the timestamp and a *tag* (*InfluxDB* indexed column) containing the *object* name make up the primary key in each *measurement*.

ClickHouse: each object is stored in a single table - the columns containing the timestamp and the *object* name make up the primary key in each table.

Before importing historical operational monitoring data into the *InfluxDB* and *ClickHouse* databases, it was exported from *P-BEAST* and stored as text files.

Then, for both technologies, a separate test was developed to implement the same functionality:

1) initialize the database and create the necessary tables if this has not been done yet;
2) read the intermediate store of data and fetch records until a batch of 10000 data points have been filled;
3) write the prepared batch of data to the database.

Initially, testing several batch sizes was being planned, such as batches of 100, 1000, and 10000 data points. But the execution time of the tests was already very long, in the order of months, so only the 10000 data points batch tests were kept in the testing plan.

B. Read testing (synthetic)

The databases created and populated with data during write testing have been used for all read tests. They contain the same raw data for both *InfluxDB* and *ClickHouse* because the same original *P-BEAST* data had been written into them during write testing.

The first, and simplest, type of queries that can be used are those in which all data points are fetched from a specified time interval. However, in a production setting where *Grafana* is going to be used to display data, such simple queries would be meaningless in many, if not most, situations. Thus, more realistic queries needed to tested.

In order to make use of *Grafana*'s capabilities to display complex graphs with multiple data series, a more complex query that can fetch the data needed to display multiple time series on the same graph is needed.

The complexity of the query is caused by the need to fulfill one or both of the following two requirements:

1) separate tables into data series by a tag (in the case of *P-BEAST*, the tag being the *object* name)
2) aggregate measurements over a given time interval

By combining these two requirements we end up with four types of possible queries:

1) a simple query, like those mentioned earlier, without time series separation and without aggregation;
2) a query with time series separation but without aggregation;
3) a query without time series separation but with aggregation;
4) a query with both time series separation and with aggregation.

Out of these four query types, only the first and the last have been kept in order to prevent the analysis from becoming too verbose. As such, the following query types were used:

Type 1 query (simple query):
InfluxDB:

```
SELECT * FROM ifdb..ifdb WHERE
time > 1501121487282103000 AND time <= 1501133832282103000
```

ClickHouse:

```
SELECT time, object, value_uint32 FROM ch.ch WHERE
time > 1501121487282103000 AND time <= 1501133787282103000
```

Type 4 query (complex query):
InfluxDB:

```
SELECT mean(*)
FROM ifdb..ifdb
WHERE object =~
  /^(object1|object2|object3|object4|object5)$/ AND
  time > 1501121487282103000 AND time <= 1501133832282103000
GROUP BY time(600s), object
```

ClickHouse:

```
SELECT object, groupArray((t, c))
AS groupArr
FROM (
  SELECT
    (intDiv(time, 600000000000) * 600000000000) as t,
    object,
    avg(value_uint32) AS c
  FROM ch.ch
  WHERE
    (object LIKE 'object1' OR object LIKE 'object2' OR
    object LIKE 'object3' OR object LIKE 'object4' OR
    object LIKE 'object5') AND
    time > 1501121487282103000 AND time <= 1501133787282103000
  GROUP BY object, t ORDER BY object, t)
GROUP BY object ORDER BY object
```

A couple of issues became apparent with above queries.

1) Issue 1: Limitation of ClickHouse Grafana datasource plugin: The `groupArray` *ClickHouse* function returns an associative array as a list of tuples. Each tuple contains a key and a value. In these queries, they key is a timestamp, but the timestamp and the object name could be flipped around and the object could be used as the key, if needed. The value is whatever type of data is present in the table being queried (or some aggregation of it).

The encountered problem was that the *ClickHouse Grafana* database plugin used for testing was not able to handle tuples of any kind. As a result, the complex queries would have been impossible to test. After investigations, it turned out that only the *C++ ClickHouse* client could handle all the data types that *ClickHouse* can output. The testing setup had been using the *Go!* client since the beginning, so moving over to *C++* would have been a problem. So, the easier way was to implement tuple support in the *Go! ClickHouse* client library.

The only limitation of this feature's implementation is that it can work with all basic data types, but not with arrays. No arrays can be returned as values in the key/value pair of the tuple. This is because of the architecture of the *Go! ClickHouse* library, which would have required much more extensive modifications if it were to be modified to handle array tuple values as well.

2) Issue 2: Not all queries made sense for all attribute data types: The data types used in the tests, as mentioned earlier, are arrays, strings, floats and integers. The complex queries, because of the time series separation and of the aggregation, offer no easy solutions for compound data types such as arrays and strings. Complex queries can be run only on basic data types because:

- Time series separation, although conceptually possible, was not implemented for *ClickHouse* because of a limitation of the *ClickHouse* library mentioned in *Issue 1*;
- Aggregation, while theoretically possible for strings and arrays, is a more complicated topic which was considered outside the scope of this work.

For each test run, the server is started, checked that it started, the test is run and then the server is stopped. This is done for two reasons:

1) in order to avoid any caching artifacts that could skew the results, the server is stopped after each run;
2) because it was noticed in preliminary testing that some of the queries can be intensive enough to crash the servers, both for *ClickHouse* and especially for *InfluxDB*. By making sure that the server is not running at the end of a run, be it because it crashed or because it was stopped cleanly, each test in the test suite has the same starting point: starting the server and making sure it initialized properly before sending queries.

C. Read testing (Grafana)

The standard interface of *P-BEAST* is based on *Grafana*, so any potential technology that could be used as a new database engine for *P-BEAST* would need to be able to work as well as possible with *Grafana*. Thus, because even the complex query tests were still being used in a synthetic environment, a final round of read testing using *Grafana* has been set up.

The *Grafana* testing started from an existing *P-BEAST* dashboard (see figure 3).

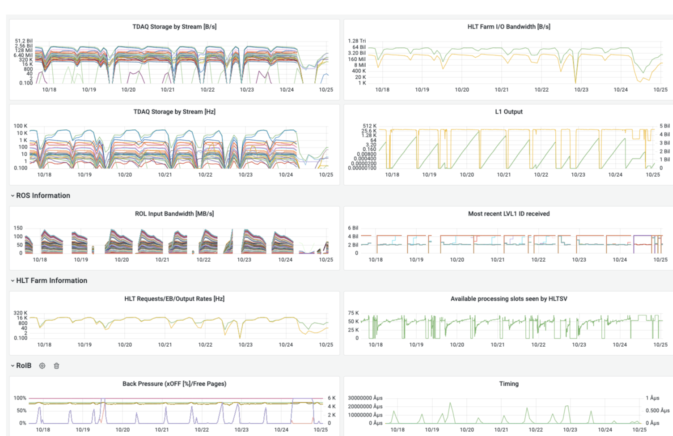

Fig. 3. *Grafana* screenshot: The ATLAS basic dashboard.

The first step was to replicate the functionality of this dashboard but using first *InfluxDB* and then *ClickHouse* as a backend. A one-month long time interval was selected and the The *ATLAS* operational monitoring data used in the chosen dashboard was imported both into a *InfluxDB* and into a *ClickHouse* database.

Then, once the data were imported into the databases, the original *P-BEAST* dashboard was recreated for *InfluxDB* and for *ClickHouse*. Care was taken to get the recreated graphs to be as close as possible to the original graphs. See figures 4 (from the *InfluxDB* dashboard) and 5 (from the *ClickHouse* dashboard) to get an idea of how similar to each other the two new dashboards were able to be created. As can be seen in the figures, there are still some small differences, the reason for this being the slightly different functionality of the *InfluxDB* and *ClickHouse* plugins of *Grafana*.

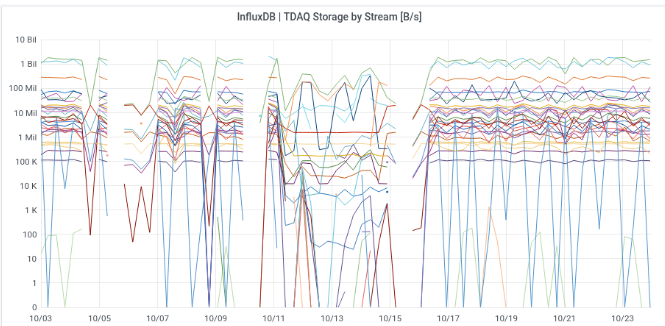

Fig. 4. *Grafana* screenshot: *InfluxDB* graph for the "TDAQ Storage by Stream [B/s]" attribute.

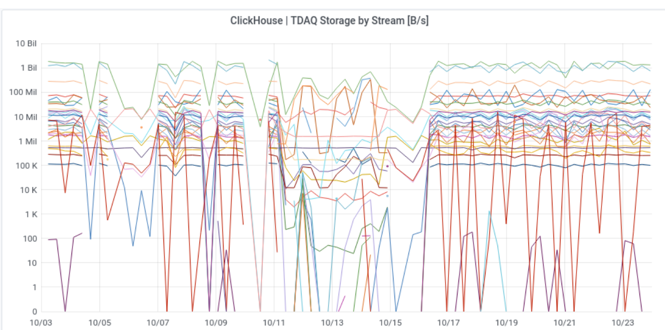

Fig. 5. *Grafana* screenshot: *ClickHouse* graph for the "TDAQ Storage by Stream [B/s]" attribute.

Again, like in the case of the write performance testing, the *P-BEAST* version was tested as well, for the purpose of using these measurements as a reference. Time intervals starting from 3 hours and up to 21 days have been selected. For each time interval, 30 measurements were taken while making sure that no caching is involved to skew the measurements. The 4 setups that were tested are:

- InfluxDB
- ClickHouse
- P-BEAST with caching
- P-BEAST raw (no caching)

V. Test Results

The conclusions of this batch of testing vary across the course of the various tests.

A. Write testing

For write testing, *P-BEAST* has demonstrated write performances above both *InfluxDB* and *ClickHouse*. Between the latter ones, *ClickHouse* has been the technology with better results across all the performed tests.

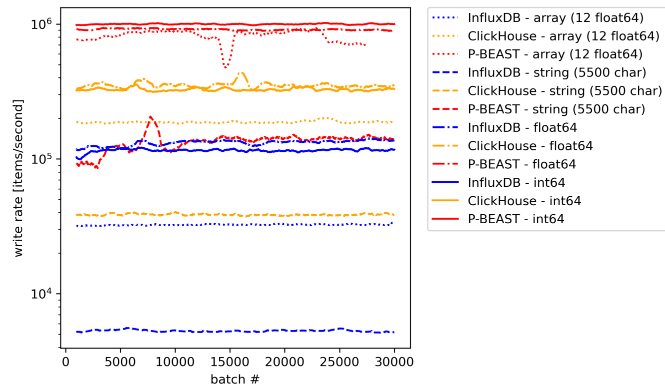

Fig. 6. Write rate for *P-BEAST*, *InfluxDB* and *ClickHouse* and all the tested data types.

Furthermore, *ClickHouse* has the advantage of the fact that has free built-in clustering support, which can be used to even further increase its write rate. The clustering support is a commercial offering for *InfluxDB*.

B. Read testing (synthetic)

For synthetic read testing, the results between types of queries are very different among the tested technologies.

In the case of simple queries, *ClickHouse* always shows better read performance than *InfluxDB*. See figure 7:

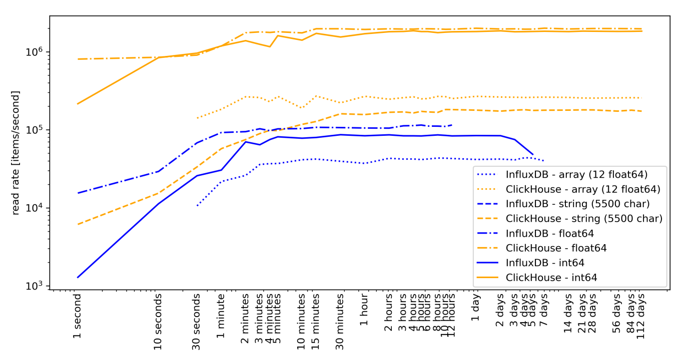

Fig. 7. Read rate for simple queries for *InfluxDB* and *ClickHouse* and all the tested data types.

However, in the case of complex queries, the performance of *InfluxDB* goes over that of *ClickHouse* from a certain query interval onward, and stays above for the tested intervals as shown on figure 8.

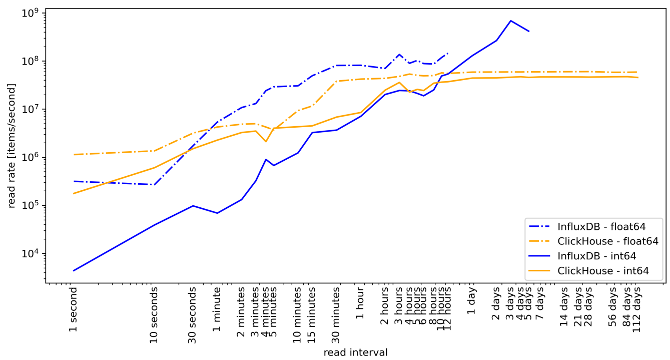

Fig. 8. Read rate complex queries for *InfluxDB* and *ClickHouse* and the data types that work with complex queries.

C. Read testing (Grafana)

During *Grafana* read tests, several tens of queries are started simultaneously by the dashboard, that is quite different from synthetic read tests, where every test was executed individually.

As suggested by the better complex query performance of *InfluxDB*, it was not unexpected to see that in the *Grafana* read testing, *InfluxDB* showed consistently better performance than *ClickHouse* as shown on figure 9:

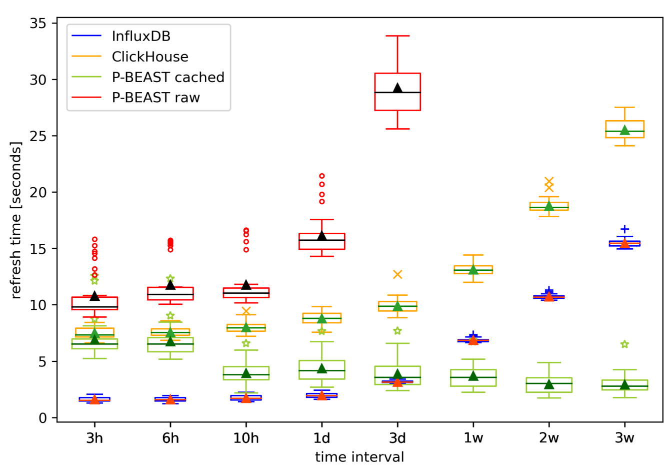

Fig. 9. *Grafana* dashboard refresh time for *P-BEAST* (both cached and raw), *InfluxDB* and *ClickHouse*.

The read performance of raw *P-BEAST* without caching enabled is below both *InfluxDB* and *ClickHouse*, that is explained by its primitive data files format (no data indexing by time inside weekly compacted and compressed data files).

When looking at the performance of *P-BEAST* with caching enabled, despite starting in a place similar to *ClickHouse* and worse than *InfluxDB*, for longer queried intervals it demonstrated better performance compared to both *InfluxDB* and *ClickHouse*.

This suggests that if either *InfluxDB* or *ClickHouse* would be used as a back-end, with caching, for a potentially modified *P-BEAST*, the performances of the upgraded *P-BEAST* would be above what is currently available.

VI. Conclusions

The test results demonstrated much better write performance of present *P-BEAST* implementation oven both *InfluxDB* or *ClickHouse* technologies. To sustain Run II data insertion rates, several times more more hardware resources will be necessary assuming linear increase of the write speed with the number of computers in the cluster.

The read performance of both *InfluxDB* or *ClickHouse* technologies is better than present *P-BEAST* without caching option enabled. It is expected, implementation caching for them will increase speed for both technologies.

A. Further research

Evaluating *InfluxDB* and *ClickHouse* as possible *P-BEAST* back-ends is the next step of this research.

References

[1] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST*, vol. 3, pp. S08003, 2008, DOI. 10.1088/1748-0221/3/08/S08003.

[2] L. Evans and P. Bryant, "LHC Machine," *JINST*, vol. 3, pp. S08001, 2008, DOI. 10.1088/1748-0221/3/08/S08001.

[3] G. Avolio, M. D'Ascanio, G. Lehmann-Miotto and I. Soloviev, "A web-based solution to visualize operational monitoring data in the Trigger and Data Acquisition system of the ATLAS experiment at the LHC," in *J. Phys.: Conf. Ser.*, Oct. 2017, pp. 032010.

[4] InfluxDB: Open Source Time Series Database https://www.influxdata.com

[5] ClickHouse: Fast Open-Source OLAP DBMS https://clickhouse.com/

[6] M. Vasile, G. Avolio and I. Soloviev, "Evaluating InfluxDB and ClickHouse database technologies for improvements of the ATLAS operational monitoring data archiving," in *J. Phys.: Conf. Ser.*, Apr. 2020, pp. 012027.