CMS DAQ monitoring data storage technology evaluation

Summer student report

Tomas Möre

August 30, 2019

Contents

Abstract

Up until the summer of 2019 the CMS DAQ system monitoring data have been stored in compressed JSON files in so called 'snapshots' taken during 2-3 second intervals. This storage method has made analysis of the stored data an inconvenient process. To address this issue, an evaluation of some contemporary storage systems have been made and benchmarked with our monitoring data. The two most promising technologies, Elasticsearch and PostgreSQL with Timescale has been selected and evaluated on data for a two-month period.

1 Introduction

This report describes the work done and summarizes the benchmarking results undertaken during the summer student project 2019. The purpose of the project was to evaluate database storage technologies for the CMS DAQ monitoring data.

2 Motivation

Before this project, all monitoring data from the CMS DAQ monitoring system was stored in separate GZIP compressed JSON files on a server. This system has worked well for retrieving a specific snapshot, but has come with the downside that retrieving ranges of historical data is burdensome and slow. In order to improve our analytical capabiliters we've looking into replacing our monitoring system with a novel solution. The target system must be able to store monitoring data permanently and be capable of handling real-time and historical analysis. It should also provide a good performance and memory profile, and be reasonably simple to use and maintain.

3 Details about data

The exact details about the data stored varies depending on the configuration and set up of the system. The metrics provided here should not be interpreted as absolutes. These values are taken as the average over the year of 2018.

A snapshot file is a collection of about 1700 objects, each 'object' represents a set of metrics collected from a device in the monitoring system together with information about what other devices they are connected to. In total there are about 40 K metrics per snapshots.

This accumulates to roughly, 30 K snapshots per day, which in the current format is about 7.4 Tb worth of data per year if it were uncompressed or 1 Tb or year using compression.

4 System use-cases

The following three main use-cases was identified and used as the main goals of the project.

4.1 Real time

We need to display the monitoring data real-time on our monitoring tools in Control Room. This is currently done by polling for most recent monitoring data (1 full latest snapshot) every few seconds but: (a) not all monitoring data is displayed (b) polling introduces delay.

4.2 Post mortem - accessing past data

We want to see all monitoring data (1 full snapshot) for any given time from the past in the cases we debug some problems. It's similar to real time in the sense of what data we retrieve but instead of last snapshot we want to see data for any point from the past.

4.3 Post mortem - plotting monitoring data

We want to plot any value from multiple snapshots over time (for instance plot rate, accumulated events, dead time over last year, with some conditions, e.g. only during running in stable beams)

5 Initial survey

Initially, we wanted to perform a survey of possible technologies that may be suitable to our needs. Doing a proper judgment based on online resources is a difficult task as the information that exists on systems are prone to exaggeration. If benchmarks are available, the results tend to be in favor of the database system that the author is associated with. And even if they are impartial it may show that the benchmarks are not applicable to the use-case in interest and characteristics of our data.

Based on this survey we made a list of 16 systems that may be of interest together with an arbitrary list of the seemingly pros and cons of each system. In order to reduce this to a set of systems that would be feasible to test within the time constraints of the project we narrowed this list down to the four most promising systems: PostgreSQL, ArangoDB, OracleDB, Elasticsearch.

However, it was quickly decided that setting up a proper oracle database for evaluation may be tedious task we instead decided to try out two different models of PostgreSQL. Later on we changed the PostgreSQL implementation to TimescaleDB which is a PostgreSQL extension adding features related to time series data.

6 Database models

In order to evaluate the database systems a complete implementation of the data structure must be prepared for each system. This section gives a high-level description of the constructed models.

6.1 ArangoDB

The promise of ArangoDB is to deliver a so called "Document" database with the additional feature that the documents can be inserted into a graph such that all relations can be modeled inside this graph. In order to utilize this to the fullest we decided to model it such that all data from all objects of the data are put into separate collections. All relations in the data are modeled as edges in one big graph. Each edge here is modeled by the ID of the source of the relation and ID of the target of the relation as well as an extra field where additional information about the relation may be stored. This entails that any retrieval of elements with relation to some other objects is a simple graph traversal. Note that the graph structure exists in all other models as well, what ArangoDB provides is simply a 'theoretically clean' interface to work with it as such. As for the data itself it can simply be stored 'as-is' in the JSON format as all documents are dynamically typed.

For the in initial benchmarks, bulk insert is used by bulking every kind of object by itself, gathering the generated database IDs. Once this is done the graph is then constructed using these IDs as vertices in a last step. To get a full snapshot back, all objects belonging to a specified snapshot ID is traversed and the relations are gathered object per object by querying the database for the edges originating from the specified object.

6.2 Elastic Search

Elasticsearch is implemented by creating one index per object type in the data structure. Each index is fully defined type wise by declaring a 'mapping'. The data is inserted in the original convoluted JSON format meaning somewhat less pre-processing is needed compared to the other systems. An effect of this is that information stored about the relations are not capable of uniquely identifying the target by itself. To achieve this the device name and snapshot timestamp must be used. Even though the data is inserted in a 'convoluted' manner, internally Elasticsearch does flatten out the data. All textual data is specifically stored as Elasticsearch's 'keyword' type because the full text search capabilities are not desirable in terms of storage and the time it takes for ES to index it.

Because we only work on one server we only use one shard per index. Furthermore, replication was deactivated for the benchmarks.

6.3 PostgreSQL normalized model

Was modeled as what one could describe as the classical way. Each object type is encoded into tables where the data is inserted. For each kind of relationship that exists, a separate table is created linking the row ID of one object to another. For the relations where some extra information is needed, there are extra fields in the relationship table.

The major caveat compared to Elasticsearch and ArangoDB is that PostgreSQL tables does not allow data to be 'convoluted'. That is, fields in the data may not be stored as sub-objects. (Except if stored as JSON or JSONB types which makes querying these values considerably slower). In order to solve this problem, a flattening scheme was implemented

for the data such that all fields containing a single sub-object are renamed to the field name concatenated with the inner objects field names recursively.

Some of the string fields in the data are effectively enumerable, that is, the value of the string will only be of a small set of possible strings. This leads to an space (and slight time) optimization strategy where the columns can be defined to be of a custom "Enum" type in the database. This strategy is implemented for some values but is not fully implemented for all possible fields. Mainly because figuring out what textual fields are enumerable takes considerable amount of time and expertise of system experts, thus no full definition was available during the project. Enumerables does introduce a problem, if one were to add new kinds of fields these must be specified in the database as well.

In the initial benchmark, a similar technique to ArangoDB for creating the database relations was used. First, all objects are inserted in bulk grouped by their object types. Then all ID's are collected, connected and inserted into the relationship tables.

To retrieve the data each object performs one or many sub queries querying for the database ids of the relationship targets related to the type of that object.

Initially we used PostgreSQL 9.2 however it turned out that this system did not default to using multiple cores to process requests therefore, at first multiple cores was manually switched on and later we switched to PostgreSQL 11 which comes with these settings as well as some minor improvements to overall performance.

6.4 PostgreSQL denormalized model

The non-relational model is mostly similar to the relational. The difference is that each relation is instead stored directly in the rows encoded by the device name given in the snapshot. Instead of storing these names as pure strigns the 'Enum' type of PostgreSQL is used. For cases where there are many-to-many relationships the data is stored using the PostgreSQL Array type.

The main benefit of this approach is that when retrieving an object no sub-queries needs to be done to get the full information of the original snapshot data back. This way, one could argue that the model becomes way more similar to the Document based alternatives (ArangoDB and Elastic Search).

However, because the relations aren't unique keys anymore the unique identifier of each object now is its snapshot id (timestamp of the snapshot) and its name.

6.5 PostgreSQL denomalized + separated model

The so called 'separated' PostgreSQL model is similar to the non-relational model except for the fact that we separate the relations into a separate graph structure that stores the relations between the objects by name only. In that sense, the querying potential is the same as the non relational PostgreSQL but decrease the space requirements drastically as the data about the relations does not need to be stored per snapshot.

The graph edges are stored in one table, each row consists of four values, the graph id, the edge source device name, edge target device name, and a special column for extra

information.

Insertion of snapshots are per object type in bulk, .

Retrieval is done slightly differently and one could argue more lazily then the others as the objects are retrieved as is using the snapshot id.

7 Initial benchmarks

Initially we created a python program which was supposed to run all the benchmarks we cared about. The benchmarks were selected to measure the following set of properties of the selected systems:

- Writing speed
- Read speed
- Time to write until the data becomes readable again.
- A full time series retrieval
- Accumulated time series data
- Binned time series data

Unfortunately, at the time we wrote the program to perform these benchmarks we did not have access to proper benchmark machines this mean the benchmarks had to be performed on a personal laptop (Dell XPS 9550). Because of this limitation together with the fact that the laptop was not available outside of working hours we were practically limited to running the tests for small amounts of data. In this case 1129 snapshots was used, corresponding to the period 8am to 9am 2018-07-01 GTM +0.

Table 1: Speed benchmarks for initial tests

Test	Elastic Search	PG (Relational)	PG (Non-relational)	PG (Separated)	ArangoDB
Write speed (s)	0.379	0.608	0.268	0.254	1.84
Read speed (s)	0.0361	2.91	0.0216	0.0157	0.494
Full approximate read (s)	0.0394	2.29	0.0264	0.0177	0.438
Read and write (s)	0.523	0.630	0.297	0.246	1.84
Average value accumulation (s)	0.00101	0.428	0.156	0.0957	0.0196
Grouped value accumulation (s)	0.00231	0.0459	0.0270	0.0234	17.6
Time series retrieval (s)	0.307	0.00756	0.00141	0.00247	0.00203

Because the time series queries had a pretty high variance in these small benchmarks we are unable to use them for any intermediate conclusions.

These disk usage numbers are with compression with the help of compression capabilities of the ZFS file system.

The main conclusion from these tests were that we would not be able to insert the data snapshot-per-snapshot for a larger benchmark as the insertion time for one year would take about 40-70 days in total. Furthermore, at this point we decided to only do the larger benchmarks on the separated PostgreSQL model and Elasticsearch model.

Table 2: Space benchmarks

	Total size (Kb)	Size per snapshot (Kb)
Elastic Search	1854544	1519.8
PG (Relational)	2958608	2620.6
PG (Non-relational)	1686536	1493.8
PG (Separated)	543736	481.6
ArangoDB	957960	1383.7

At this point we decided to drop ArangoDB because the write time is too close to the 2 seconds real time insertion speed. Furthermore, we did not see any benefit of the graph structure from a development perspective, although nice in theory we concluded that it was too similar to for example PostgreSQL in reality.

To solve the insertion speed problem, a custom program was created whose sole purpose is to read and parse the pre-existing snapshots as quickly as possible and to insert them into databases. This tool was developed in the programming language Rust as we wanted to ensure that a good performance was achieved. This would not be possible to do Python as the language is intrinsically slow and does not handle parallelism well from a performance perspective.

While waiting for the benchmark servers to be ready we started doing some experiments with an extension to PostgreSQL by the name of TimescaleDB which is supposed to add functionalities that is better at storing time series data. We then decided to make a few tests using this extension with the same model as PostgreSQL except for the fact that the timestamps are stored as a "Time" format instead of simply the previous Unix epoch time.

Table 3: 10K snapshot insertion tests

Metric for 10000 snapshots	Time (ms)	Compressed Size (Kb)
PostgresSQL	268704	1161360
TimescaleDB	367928	1359745
Elastic Search	690731	3405021

Table 4: 10K snapshot query test

Metric for 10000 snapshots	PostgresSQL	TimescaleDB	Elastic Search
Time (s)	460 ms	327 ms	2178 ms

The query in question was an exercise where we wanted to use the graph structure to retrieve some data. In this particular case the query retrieved a value over time from the device that was connected to another pre-specified device.

The above results including some promises of some other benefits for larger amounts of data made us decide to start out our benchmark tests using the Timescale system instead of the pure PostgreSQL version.

One of the features of Timescale is that there are a possibility to automatically create materialized view of the tables that can represent summaries of the raw data tables. In our implementation we decided to try out this feature for all numerical data storing the average, maximum and minimum value in these periods.

8 Final benchmark

Due to late reception of benchmark machines the final results had to be a rushed. Once we got the server we had to spend time configuring them correctly. Furthermore, we noticed some inefficiencies in both the insertion program and the configuration of the databases once we started inserting a lot of data. Once these problems were fully resolved we only had less than a week left to perform the analysis.

This time the amount of data corresponded to a two-month period between 2018-08 to 2018-10 consisting of 2290071 snapshots.

Table 5: Insertion time and disk usage.

	Timescale	Elasticseach
Time (h)	16.85 h	84.7 h
Compressed size	406 Gb	958 Gb
Uncompressed size	1285 Gb	1126 Gb
Compression ratio	3.2 x	1.2 x

For the querying benchmarks we decided to run four queries:

1. Time series value retrieval on the smallest table
2. Time series value retrieval on the largest table
3. Individual time series value retrieval of the ten worst devices
4. Full accumulation of values binned on device names

These queries were run on four different time intervals, one hour, one day, one month. For each one of these periods we ran the queries up to a hundred times each on randomly selected distinct intervals. The actual number of intervals were the number of possible intervals for the data we had, limited to a hundred.

In an attempt to make the systems equal we cleared the operating system caches before the start of the benchmark run. This had the unfortunate side effect that the longer queries became faster over time as the caches were reconstructed during the runs. While not optimal there was no time left for corrections.

Table 6: Query 1: Time series value retrieval on the smallest table

Time-frame	Elasticsearch	Timescale (raw data)	Timescale 10 min table
hour	2278	25	51
day	3775	27645	253
week	297	1112	353
month	708	4433	22

Table 7: Query 2: Time series value retrieval on the largest table

Time-frame	Elasticsearch	Timescale (raw data)	Timescale 10 min table
hour	3137	6028	79
day	3816	131421	760
week	5418	20925	445
month	19852	98568	6322

9 Conclusion

We verified that both of the selected systems are feasible for storing our monitoring data. We learned the limitations of the evaluated systems. However, the final decision has been postponed until a full years worth of data has been inserted into the systems.

In an attempt to summarize what we have learned so far about the systems we composed the following list of pros and cons. The pros are prefixed with a '+' and cons with a '-'.

Elasticsearch	**TimescaleDB**
+ Easily scalable	+ SQL
+ Faster quires for long time ranges	+ More powerful querying language
- No join possible natively	+ Faster inserts
- No sub queries available	+ Smaller disk footprint
- Verbose query language	- Strict schema (except for JSONB fields)

10 Next steps

Because the summer student period ended before the project could be fully completed a few more steps needs to be taken for a final decision. This section gives a brief outline of what has been planned.

10.1 Benchmark related

- To be able to do proper technical evaluation an entire years worth of data should be inserted into the systems. New benchmarks should be planned to give a better evaluation.

Table 8: Query 3: Individual time series value retrieval of the ten worst devices

Time-frame	Elasticsearch	Timescale (raw data)	Timescale 10 min table
hour	348	62	4
day	593	3234	21
week	2018	28784	175
month	7582	1701294	929

- In the final benchmark we noticed that the insertion time for Elasticsearch was longer then expected. To address this issue so called time based indexes should be used instead.
- Elasticsearch currently stores all the relational data directly into the system, it may turn out that this is completely unnecessary, alternative approaches should be explored.
- Timescale version will 1.5 come with a new compression algorithm. The current claim is that this should be able to perform a compression ratio of up to fourty times. This feature should be activated and evaluated.

10.2 Expand use case requirements

- The current use cases does not have the details available to decide exactly what technical capabilities the system must have. These use cases should be explored further as to give more insight in this regard.