A persistent back-end for the ATLAS TDAQ online information service (P-BEAST)

Alexandru D Sicoe[1], Giovanna Lehmann Miotto[1], Luca Magnoni[1], Serguei Kolos[2], Igor Soloviev[2]

[1] CERN, CH-1211 Geneve 23, CH

[2] University of California Irvine, Irvine, CA 92697, USA

E-mail: asicoe@cern.ch

Abstract. This paper describes P-BEAST, a highly scalable, highly available and durable system for archiving monitoring information of the trigger and data acquisition (TDAQ) system of the ATLAS experiment at CERN. Currently this consists of 20,000 applications running on 2,400 interconnected computers but it is foreseen to grow further in the near future. P-BEAST stores considerable amounts of monitoring information which would otherwise be lost. Making this data accessible, facilitates long term analysis and faster debugging. The novelty of this research consists of using a modern key-value storage technology (Cassandra) to satisfy the massive time series data rates, flexibility and scalability requirements entailed by the project. The loose schema allows the stored data to evolve seamlessly with the information flowing within the Information Service. An architectural overview of P-BEAST is presented alongside a discussion about the technologies considered as candidates for storing the data. The arguments which ultimately lead to choosing Cassandra are explained. Measurements taken during operation in production environment illustrate the data volume absorbed by the system and techniques for reducing the required Cassandra storage space overhead.

1. Introduction

ATLAS is one of several experiments built along the Large Hadron Collider at CERN, Geneva. Its aim is to measure particle production when protons collide at a very high center of mass energy, thus mimicking the behavior of matter a few instants after the Big Bang. The detecting techniques used for this purpose are very sophisticated and the amount of digitized data created by the sensing elements requires a very large data acquisition system [1]. This consists of a few tens of thousands of applications that operate on the physics data since the moment it leaves the detector until it is archived on disk. All these applications publish their monitoring information in a central repository called the ATLAS on-line Information Service [2].

The experiment is successfully taking data since the end of 2008 and the trigger and data acquisition is now in production. Among the current development efforts is the addition of easy to use and intuitive tools to aid experts monitor different components or subsystems. P-BEAST is an example of such a tool. It archives monitoring data at unprecedented levels of granularity giving experts access to detailed information about what happened in the system at a certain point in the past.

1.1. Dataset and workload considerations

The Information Service (IS) represents the data source. It is important to analyse the behaviour of this system in order to understand the information volumes it is capable of generating, the way this data can be acquired and the dynamics of the data. IS consists of a configurable set of server processes where most TDAQ applications publish histograms and state variables. Examples of such variables are the total system memory used by a particular application or the number of physics events processed by an event filter application.

IS servers have two main types of clients: information providers and information receivers. As already mentioned, the providers are applications running in the TDAQ infrastructure. They can create new information objects in the IS repository or update existing ones. Receivers can subscribe to get updates of the information objects. The communication between the IS repository and information receivers/providers is implemented on top of CORBA [3].

Most TDAQ applications have several variables to publish. These are grouped in structures called information objects. An information object is the main unit of information passed between IS clients and servers. It has a unique name (based on the name of the source application), a fixed set of fields (one for each variable of the source application), a type and a microsecond precision time stamp. When such a TDAQ application is started, it first registers its information objects with a certain IS server. As the objects are updated, they are sent to the IS server. The rate at which this occurs varies widely: some are regular and others are highly dependant on runtime conditions thus exhibiting transient behavior with peaks of up to several tens of thousands of information object updates per second. Another key aspect to point out is that all the fields of an information object are sent in every update regardless of whether they changed their value or not. This leads to data duplication that has to be accounted for.

Currently the load of monitoring data from all 20,000 TDAQ applications is spread across a total of around 230 IS servers, depending on configuration. About half of them contain histograms, which are already being archived via different means. The remaining ones contain the operational monitoring data, most of which is volatile. At present the focus lies on the 87 servers which are used by the central DAQ and Trigger systems, though an extension of the archiving for sub-detector specific information could be envisaged in the future. The total number of information objects published within these 87 servers is approximately 200,000. On average there are 10 fields in each object giving a total of 2,000,000 variables that need to be persisted for long term access. These variables are of primitive types directly mapped to C++ basic types (8,16,32,64 bit signed and unsigned integers; boolean; float; double; string) and arrays of those.

2. Requirements

The goal of the P-BEAST project is to design and implement a generic mechanism to store a subset of operational monitoring data of the ATLAS TDAQ system into a suitable database and provide an efficient means for retrieving that data. The reason why this data should be persisted is to allow the data flow experts to analyze the quality of data taking a posteriori, accessing by means of specialized dashboards, information that is stored in a permanent location.

2.1. Users

P-BEAST will be used by any system expert who wants to have a more holistic view of certain interesting elements of the TDAQ data flow that took place in the past (days, weeks, months, years). Such functionality is useful for off line analysis: to understand the behavior of different parts of the system, to make comparisons between data taking sessions of the ATLAS detector, to investigate problems, to correlate data etc. This data can serve as evidence of system operation in reports, presentations or papers.

2.2. Functional requirements

2.2.1. Insertion path P-BEAST has to take into account the structural evolution of the data published within the Information Service. Since the information objects are completely configurable they can change over time. A recent example is the implementation of a logical subdivision of racks which lead to lots of changes in the information object names. The addition/deletion of fields of an information object, from one data taking session of ATLAS to another, is also possible.

The operational data must be permanently persisted. Data up to several months has to be readily available, while data older than that has to be moved to CERN's long term storage system: CASTOR [4]. The stored information should not loose its granularity over time.

P-BEAST should be able to replicate/transfer data outside of the ATLAS enclosed network where the data providers (IS servers) reside, in order to allow for accessibility.

2.2.2. Retrieval path The system needs to be able to handle random time based queries. Given the names of the information objects and fields together with a time interval, the requested data (plus associated meta data) has to be returned in an efficient manner. With this time series data at hand, the client, which could for instance be a web dashboard [5], can correlate and aggregate different data streams prior to display.

3. System architecture

Figure 1 shows the architectural view of P-BEAST: insertion and retrieval parts are clearly separated.

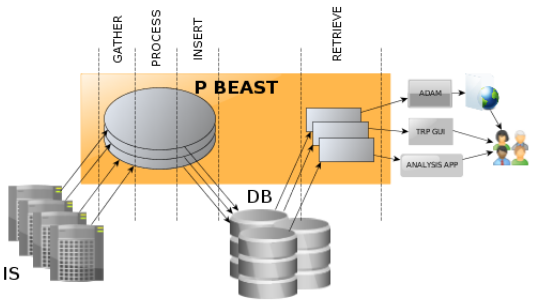


Figure 1: P-BEAST architecture.

3.1. The insertion part

Consists of multiple information gathering agents. These are completely independent and configurable applications that perform identical tasks: gathering, processing (decomposing, unmarshalling, filtering) and database insertion. At start up, each of them makes an initial subscription to a subset of IS servers, defined in their configuration file, and then waits to receive information objects. Each individual field is converted to the appropriate type and passed to a collection of filters. These share a common interface which allows them to be plugged in (the filter class names are specified in the configuration file) such that new filter implementations can be added without the need of modifying application code. Filters can be configured to target individual fields within a unique IS object or can have a broader reach.

Currently there are two filter implementations employed. The default is discarding duplicates generated by successive updates of information objects containing the same field value. This is a characteristic of the Information Service that simply resends the entire information object even if none of its constituents changed value since the last update. The second filter implementation

applies smoothing to numerical values only. A certain field value is rejected if it is within a predefined range from the last value inserted. This range is defined in terms of percentages.

Field values that are accepted by all configured filters are placed in a batch. When the batch reaches a certain size it is passed to a database access object that performs a batch insertion into the database.

3.2. The retrieval part

Is based on a similar concept as the insertion part with multiple independent clients. Each client will access the database through a common retrieval API. In the simplest scenario, the functionality provided by the API can be used by a custom analysis application. More advanced clients would receive HTTP requests (like the ones issued by the ADAM dashboard [5]) which they would map to P-BEAST API calls. The data returned would be arranged in a client specified format (JSON, XML, CSV etc.).

4. Database platform considerations

Initially relational database technologies were considered as the storage platform for P-BEAST. A simple exercise in mapping IS information to the relational model lead to the conclusion that the fixed tables cannot provide the flexibility necessary to easily and efficiently track information object structure over long periods of time. In order for this to be attained, de-normalization has to be employed. This technique places considerable burdens on maintenance. The complexity of retrieval queries grows and performance decreases as the data set grows larger. Moreover, the high rates of insertion are generally problematic for relational databases. The solution is scaling up by using more powerful machines in terms of RAM and CPU. For extremely large datasets this strategy becomes costly.

A whole range of alternatives were evaluated. Part of them belong to a technological trend called NoSQL [6] (e.g. Cassandra [7][8], HBase [9], MongoDB [10]). This term brings together a plethora of distributed structured storage systems that are very different in implementation and data models but which share a few common aspects like: flexible schema, horizontal scalability, non-ACID [11] etc, which can bring an advantage to P-BEAST in terms of ease of development, scalability and performance. Due to the abundance of solutions it was found that the fastest way to choosing a final candidate for the storage component of P-BEAST was to filter them against a set of criteria: open source, good fit to the use case of P-BEAST, read/write performance, community activity, ease of deployment and maintenance.

RRDTool [12], Graphite [13] and OpenTSDB [14] are specifically built for handling monitoring time series data but were ruled out for their lack of generality (only support numerical values) and loss of granularity. KDB [15] is a highly performing tool used for storing and processing financial time series data. Unfortunately it is a commercial product and just the 32 bit version is available openly with not much active development. HDF5 [16] is a file system with a library that provides functionality to access the files. It has a rich data set that supports multidimensional arrays but does not have any data management functionality. MongoDB is a document-oriented storage system that provides very high schema flexibility. It can be used as a fast storage system for analytics data but at the moment when it was evaluated it was not durable, meaning that data could be potentially lost after a machine failure because it is initially cached in RAM. Also, MongoDB performs really well only if the indexes fit entirely in memory. As the data size of IS is large this was most likely going to cause difficulties and would entail scaling up.

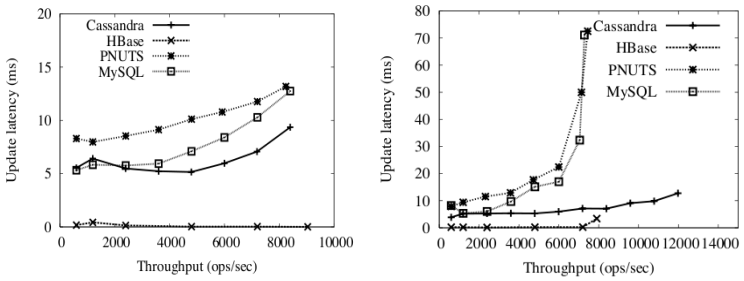


Figure 2: Update (write) latencies - from [17] p.9: a) Work load: 95% reads / 5% updates, b) Work load: 50% reads / 50% updates.

In evaluating Cassandra and HBase it was found that the primary use case of Cassandra, high availability and write performance with optimized queries for retrieving slices of values, is a better match for P-BEAST than that of HBase which is more focused on real time queries and performing computation across large data sets. The relative performance of the two is shown by a study published by Yahoo! [17] containing benchmarks under different load conditions. These were obtained using their cloud serving benchmarking platform [17] which is available for free and can be easily adapted to stress test any storage system.

Figure 2 shows the update (write) latency as a function of the load placed on each platform, for two different work loads: read intensive (95% of the operations are reads) and balanced (reads and writes occur in the same proportion). The second workload is closer to the intended usage of the platforms because the use case of P-BEAST entails writing for the majority of the time. In the read intensive workload (figure 2a) clearly HBase performs better but in the workload with more writes (figure 2b) even though HBase outperforms Cassandra's to start with, it proves incapable of handling loads higher than 8,000 operations/sec. Cassandra's write latency remains stable across higher loads. This is a desired feature to have in order to satisfy the potentially growing loads generated by the ATLAS Information Service. For the read latency under the same two workloads (figure 3), Cassandra presents superior performance showing it is a better fit for content retrieval.

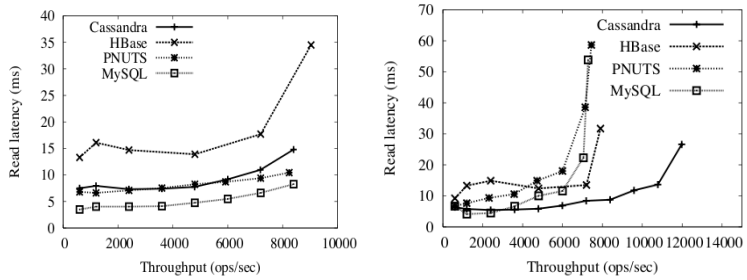


Figure 3: Read latencies - from [17] p.8: a) Work load: 95% reads / 5% updates, b) Work load: 50% reads / 50% updates.

The main reasons why Cassandra is in a better position than the others to become the platform of choice are:

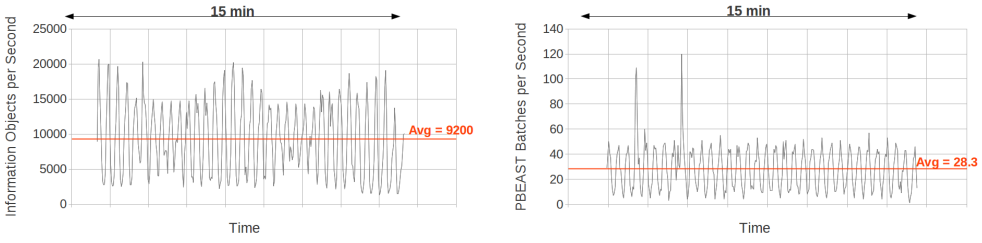
- Cassandra offers the best fit to the use cases of P-BEAST

  - Lots of writes with significalntly less read operations
  - Can absorb peaks in the write rates [18]
  - Designed to handle a very high write load (thousands of operations per second) [19] from multiple clients
- Easy to increase performance by just adding more machines (scales linearly with the number of machines assuming that keys are randomly distributed which is the case in real data sets)
- Ease of deployment (facilitates the fast prototyping approach used for P-BEAST)
- Low maintenance requirements (important for the long term intended use)

Cassandra is a key value storage system that operates as a distributed hash map. Due to its simple data model, it has a strong use case for storing large amounts of time series data (used by Cloudkick [19] for monitoring cloud infrastructure). Each value inserted has to have an associated key and a name. The key identifies the row where the name value pair is placed. Each row stores name-value pairs sorted by the name. This essentially provides two indexes for retrieving values: an index on row keys and one on names inside a row. Data is distributed within a cluster of nodes by applying MD5 function to the key. Each cluster node is assigned a particular range of MD5 hash values. All the name value pairs of a particular row will be stored on the node responsible for the range where that row key hashed to.

Cassandra was inspired from Amazon's Dynamo [20] where it got the distributed ring arrangement and its high availability. Furthermore it borrowed the storage data model and basic insertion and retrieval mechanisms from Google's Big Table [21]. It uses the SEDA [22] architecture for multi threading which allows more control on the computation process. To communicate with any Cassandra node there is an API on top of Apache's Thrift [23] protocol which allows remote procedural calls and facilitates communication between applications written in different languages.

5. Results



(a) (b)

Figure 4: Insertion rates measured while runnning during ATLAS recording physics data: a) Update transactions per second, b) Batch insertions per sec.

Deploying P-BEAST (with six gathering instances) in the TDAQ infrastructure while ATLAS is recording physics events reveals the rates shown in figure 4. The gathering agents are configured with only the default duplicates filter and send data to a three node Cassandra cluster (each node has a 4 core processor, 4 GB of RAM and separate rotational disks for the commit log and the data directories). Figure 4a represents a measurement of the total update transactions per second received by all six P-BEAST gathering instances from the 87 IS servers of interest. An update transaction corresponds to an information object being transferred over the network to P-BEAST. The plot shows the irregular traffic that the Information Service generates with an average of 9,200 updates per second. This input load is transformed by filtering and batching into

a load presented to Cassandra depicted in figure 4b. On average 28.3 batches/sec are inserted. With a measured average of 226KB of data per batch, the Cassandra cluster thus handles an average of 6.4 MB/sec of input data.

The Cassandra meta data can have a significant impact on the required storage space. The storage space needed for the ATLAS partition during a test was estimated based on standard formulas [24] for calculating Cassandra overhead for rows and name value pairs (essentially for the native indexes offered, bloom filters etc). The formulas described in table 1 were fed with several statistics gathered during 15 minute tests: the total number of rows created during the test (TNR), the total number of name value pairs inserted (TNC), the average number of name value pairs per row (NC), the average sizes of row keys and of names (RKS).

Table 1: Cassandra storage space: 100 sec vs 1000 sec time bins.

Measurement	100 sec bins	1000 sec bins
Raw data inserted[Sum (size of name + size of value)]	1.5 GB	1.3 GB
Name value pairs overhead [TNC x 15]	287.9 MB	398.49 MB
Row header overhead [TNR x 23]	103.5 MB	51.93 MB
Row bloom filter[TNR*(2 + (8 + ceiling((NC * 4 + 20) / 8)))]	67.94 MB	42.2 MB
SSTable index[TNR * (10 + RKS)]	783.5 MB	386.5 MB
SSTable bloom filter[(TNR * 15 + 20)/8]	8.4 MB	4.2 MB
Base Storage (BS)	2.7 GB	2.2 GB
Raw data inserted (% BS)	54%	60%
Column Overhead (% BS)	10%	18%
Row overhead (%BS)	4%	2%
SSTableOverhead (%BS)	29%	18%

These calculations were performed in order to test the impact of various Cassandra schema parameters on the storage efficiency. The schema employed uses Cassandra rows as time bins. The time bins are implemented in the following manner: for each new field of an incoming information object, a Cassandra row key is computed by concatenating the 8th (for 100 sec bins) or 7th (for 1000 sec bins) most significant digits of the object time stamp, to the object name and the field name. The effect of computing the row keys in this way is that the value of a certain field is inserted in the same row if two successive updates are received within the same time bin, essentially stacking name value pairs in the same row.

By varying the length of the bins from 100 second to 1000 second, the number of rows is reduced while the average number of name value pairs per row is increased. The results of storage space estimation is depicted in table 1. The same test configuration described earlier was used to perform these tests. The figures in the table clearly show that by enabling longer time bins the row and SSTable[3] overhead is reduced by almost half. This change also has a positive effect on retrieval query performance because it improves data locality (the name-value pairs in one row are always stored on the same machine in the cluster).

Despite metadata optimizations, the amount of raw data being stored is still high. In order to further reduce the intake of information smoothing filters are applied in addition to the default one (duplicates filter). Figure 5 shows the estimated base storage requirements using the same techniques described before. The data rates are lower because the tests were performed on a more limited infrastructure. The procedure adopted was to run the system, in the same configuration (6 P-BEAST gathering agents and 3 node Cassandra cluster), in 4 separate 12 minute tests.

[3] An SSTable is the on disk data representation in Cassandra.

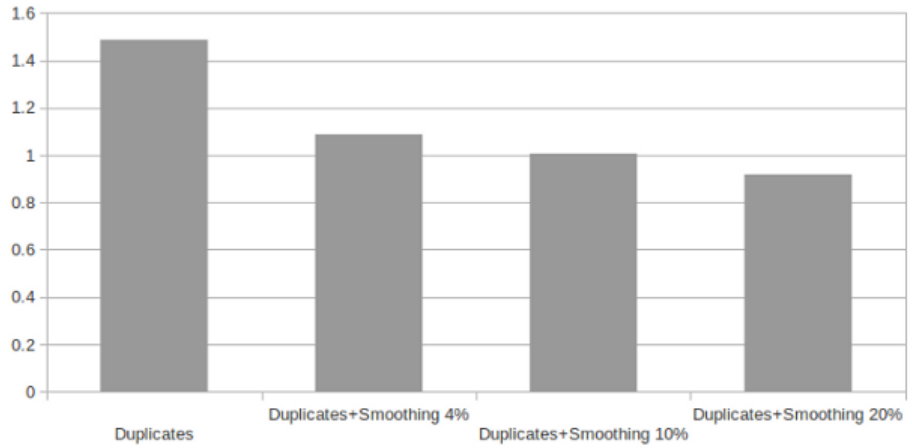


Figure 5: Estimated Cassandra storage space for different degrees of smoothing.

Each test had increasingly restrictive smoothing applied, ranging from no smoothing to 20% smothing. The effect of smoothing is to decrease the storage space up to 38%.

6. Conclusions

P-BEAST is a system that maintains an archive of operational monitoring data which would otherwise be lost. Access to this history helps data flow experts debug problems occurring within the infrastructure and make analysis of past behavior.

The writing part of P-BEAST has been tested in the production environment. On-going tests within the TDAQ infrastructure indicate that P-BEAST is capable of absorbing the load produced by the 87 servers of interest of the ATLAS TDAQ Information Service, thanks to the intermediate buffering and to Cassandra's extremely high write performance. The results presented here show the data rates handled and the possible optimizations that can be performed with respect to storage space occupancy. The raw input data rates can be further controlled by applying smoothing. The smoothing factors are completely configurable which allow probing the limits of relevancy of applying this kind of filtering for different sources of data in the future.

Further work will involve finishing insertion tests/optimizations and the long term storage space estimates. These will feed into the proposal for acquisition of new machines to deploy the Cassandra cluster on. Retrieval performance measurements will follow as well as development of the reading clients.

References

[1] Jenni P, Nessi M, Nordberg M and Smith K 2003 *ATLAS high-level trigger, data-acquisition and controls:Technical Design Report* (Geneva: CERN) pp 6–9, 23–31 http://cdsweb.cern.ch/record/616089/
[2] Jenni P, Nessi M, Nordberg M and Smith K 2003 *ATLAS high-level trigger, data-acquisition and controls:Technical Design Report* (Geneva: CERN) pp 161–162 http://cdsweb.cern.ch/record/616089/
[3] Group O M 2011 Corba http://www.corba.org/
[4] CERN 2011 Cern advanced storage manager http://castor.web.cern.ch/castor/
[5] Harwood A and Magnoni L 2011 *The ADAM project: a generic web interface for retrieval and display of ATLAS TDAQ information* Proc. Int. Ws. on Advanced Computing And Analysis Techniques In Physics Research (Uxbridge, UK)
[6] Prof Edlich S 2011 Nosql http://nosql-database.org/
[7] Foundation T A S 2011 Cassandra http://cassandra.apache.org/
[8] Lakshman A and Malik P 2010 *SIGOPS Oper. Syst. Rev.* **44** 35–40 ISSN 0163-5980
[9] Foundation T A S 2011 Hbase http://hbase.apache.org/
[10] 10gen 2011 Mongodb http://www.mongodb.org/

[11] Gray J 1981 *Proc. Int. Conf. on Very Large Data Bases - Volume 7* VLDB '81 (VLDB Endowment) pp 144–154
[12] Oetiker T 2011 Rrd tool http://oss.oetiker.ch/rrdtool/
[13] Graphite 2011 Graphite http://graphite.wikidot.com/
[14] OpenTSDB 2011 Opentsdb http://opentsdb.net/
[15] Systems K 2011 Kdb http://kx.com/
[16] NCSA 2011 Hdf5 http://www.hdfgroup.org/HDF5/
[17] Cooper B F, Silberstein A, Tam E, Ramakrishnan R and Sears R 2010 *Proc. of the 1st ACM Sym. on Cloud computing* SoCC '10 (New York, NY, USA: ACM) pp 143–154 ISBN 978-1-4503-0036-0
[18] Hewitt E *Cassandra: The Definitive Guide* (O'Reilly Media, Inc.)
[19] Cloudkick 2011 Cassandra blog http://www.cloudkick.com/blog/
[20] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P and Vogels W 2007 *SIGOPS Oper. Syst. Rev.* **41** 205–220 ISSN 0163-5980
[21] Chang F, Dean J, Ghemawat S, Hsieh W C, Wallach D A, Burrows M, Chandra T, Fikes A and Gruber R E 2008 *ACM Trans. Comput. Syst.* **26** 4:1–4:26 ISSN 0734-2071
[22] Welsh M, Culler D and Brewer E 2001 *SIGOPS Oper. Syst. Rev.* **35** 230–243 ISSN 0163-5980
[23] Foundation T A S 2011 Thrift http://thrift.apache.org/
[24] Todd B 2011 Cassandra storage sizing http://btoddb-cass-storage.blogspot.com/