Development of noSQL data storage for the ATLAS PanDA Monitoring System

M Potekhin on behalf of the ATLAS Collaboration

Brookhaven National Laboratory, Upton, NY11973, USA

E-mail: potekhin@bnl.gov

Abstract. For several years the PanDA Workload Management System has been the basis for distributed production and analysis for the ATLAS experiment at the LHC. Since the start of data taking PanDA usage has ramped up steadily, typically exceeding 500k completed jobs/day by June 2011. The associated monitoring data volume has been rising as well, to levels that present a new set of challenges in the areas of database scalability and monitoring system performance and efficiency. These challenges are being met with a R&D effort aimed at implementing a scalable and efficient monitoring data storage based on a noSQL solution (Cassandra). We present our motivations for using this technology, as well as data design and the techniques used for efficient indexing of the data. We also discuss the hardware requirements as they were determined by testing with actual data and realistic loads.

1. Introduction

PanDA is a Workload Management System built around the concept of Pilot Frameworks [1]. In this approach, workload is assigned to successfully activated and validated Pilot Jobs, which are lightweight processes that probe the environment and act as a 'smart wrapper' for the payload. This 'late binding' of workload jobs to processing slots prevents latencies and failures in slot acquisition from impacting the jobs, and maximizes the flexibility of job allocation to globally distributed and heterogeneous resources [2]. The system was developed in response to computing demands of the ATLAS collaboration at the LHC, and for a number of years now has been the backbone of ATLAS data production and user analysis.

A central and crucial component of the PanDA architecture is its database (hosted on an Oracle RDBMS), which at any given time reflects the state of both pilot and payload jobs, as well as stores a variety of vital configuration information. It is driven by the *PanDA Server*, which is implemented as an Apache-based Python application and performs a variety of brokerage and workload management tasks, as well as advanced pre-emptive data placement at sites, as required by the workload.

By accessing the central database, another PanDA component – the *PanDA Monitoring System* (or simply *PanDA Monitor*) – offers to its users and operators a comprehensive and coherent view of the system and job execution, from high level summaries to detailed drill-down job diagnostics. Some of the entities that are monitored in the system are:

- Pilot Job Generators (schedulers)
- Queues (abstractions of physical sites)
- Pilot Jobs (or simply "pilots")
- Payload Jobs (the actual workload)

All of these are related to each other by reference, i.e. an instance of the Pilot Job Generator is submitting pilots to a particular queue, a queue in turn can contain any number of pilots in various stages of execution, and payload jobs are mapped to successfully activated, running pilots. Payload jobs are also associated with their input/output datasets (details of data handling in PanDA go beyond the scope of this paper). The purpose of the PanDA Monitoring System is to present these objects and their logical associations to the user with optimal utility.

2. Characteristics of the data and current design of the PanDA monitor

As already mentioned, the PanDA Monitor handles a variety of data, and perhaps the most important and widely accessed objects stored in its database are entries describing payload jobs. We used this type of data as the most relevant case study in considering our database strategies going forward. Each job entry is mapped onto a row in the Oracle table, with roughly a hundred attributes (columns).

The monitor is currently implemented as a Python application embedded in the Apache server. Database access is done via the *cx_Oracle* Python library. The Oracle server is handling loads of the order of 10^3 queries of varying complexity per second. There are more than 500,000 new entries generated daily, which results in multi-terabyte accumulated data load. Despite ongoing query optimization effort, there are capability and performance limits imposed by available resources of the current Oracle installation and nature of the queries themselves.

Since RDBMS features such as the ability to perform join operation across several tables can be costly in terms of performance, most of the widely accessed data stored in the PanDA Oracle database is de-normalized. For that reason, there is slight redundancy in the data and joins are not used in the application.

For performance and scalability reasons, the data stored in Oracle is partitioned into "live" and "archive" tables. The former contains the state of live objects (such as job being queued or executed in one of the clouds), while the latter contains a much larger set of data which is final (read-only, such as parameters of fully completed jobs) and has been migrated from the live table after a certain period of time after becoming static. In addition, we note that monitoring applications do not require guaranteed "hard consistency" of the data, which would be another crucial motivation to use a RDBMS. In other words, the time window between the instants when information is updated in the database by its client application, and when this is reflected in results of subsequent queries, is not required to be zero (which is achieved in RDBMS by making the "write" operation synchronous).

3. Motivation for noSQL and choice of platform

As explained above, we actually don't have compelling reasons to store the monitoring data in a RDBMS. In situations like this, a variety of so-called *noSQL* database solutions are often employed by major Web services that require extreme degrees of availability, resilience and scalability. Examples include Google, Amazon, Facebook, Twitter, Digg and many others, as well as "brick and mortar" businesses. In order to leverage this new technology, we initiated a R&D project aiming at applying this approach to elements of the PanDA monitoring system to make it future-proof and gain relevant experience going forward.

The "noSQL" term applies to a broad class of database management systems that differ from RDBMS in some significant aspects. These systems include such diverse categories as Document Store, Graph Databases, Tabular, Key-Value Store and others. Based on the patterns observed in queries in PanDA, a conclusion was made that either a Key-Value or a Tabular Store would be an adequate architecture in the context of PanDA monitoring system. A significant portion of all queries done in PanDA is performed using a unique identifier each entry has, therefore representing a classic case of Key-Value query. Other types of queries are done by indexing relevant columns, which is a general approach that will still work with noSQL solutions which support indexing.

We considered two of the most popular platforms: Apache Cassandra (Key-Value) and Apache Hbase (Tabular). The latter relies on Hadoop Distributed Filesystem. To cut on the learning curve, and for ease of operation, it was decided to choose Cassandra [3], which does not have such dependency.

In addition, Cassandra was already deployed and undergoing evaluation by ATLAS personnel, giving us leverage in quickly acquiring our own R&D platform.

4. Cassandra characteristics

A quick description of Cassandra would be as follows: it's a truly distributed, schema-less, eventually consistent, highly scalable key-value storage system.

The *key*, in this context, can be of almost any data type, as Cassandra handles it as an array of bytes. The *value* is a collection of elements called *columns* (again, stored as arrays of bytes). The column is the smallest unit of data in Cassandra and consists of a *name*, *value* and a timestamp (used by Cassandra internally). This gives the developer flexibility in modeling objects in the data. Aggregation of *columns* with the same key is often called a row, and it can be thought of as a dictionary describing an object. A collection of rows pertaining to the same application is termed a *column family* and serves as a very distant approximation of what is called a table in the traditional RDBMS – note however that Cassandra rows can be drastically different from each other inside a single column family, while rows in a RDBS table follow the same schema.

By design, the *write* operation is durable: once a write is completed (and this becomes known to the client requesting this procedure), data will survive many modes of hardware failure. While an instance of Cassandra can be successfully run on a single node, its advantages are best utilized when using a cluster of nodes, possibly distributed across more than one data center, which brings the following features:

- Incremental and linear scalability – capacity can be added with no downtime
- Automatic load balancing
- Fault tolerance, with data replication within the cluster as well as across data centers

Cassandra also features built-in caching capabilities which will not be discussed here for sake of brevity. The application is written in Java, which makes it highly portable. There are client libraries for Cassandra, for almost any language platform in existence. In our case, we chose *Pycassa* Python module to interface Cassandra.

5. Data Design and migration scheme

Design of PanDA data for storage in the Cassandra system underwent a few iterations which included storage of data as *csv* (comma-separated values) data chunks, "bucketing" of data in groups with sequential identifiers (to facilitate queries of serially submitted jobs) etc. After performance evaluation, a simple scheme was adopted where a single job entry maps onto a Cassandra row. The row key then is the same as used as primary index in the Oracle database, which is the unique ID automatically assigned to each job by the PanDA system.

At the time of this writing, parts of the archived data in PanDA are mirrored on Amazon S3 system in *csv* format for backup and R&D purposes, grouped in segments according to the date of the last update. This body of data was used as input for populating a Cassandra instance as it effectively allows us to avoid placing an extra load on the production RDBMS during testing, is globally and transparently available and does not require DB-specific tools to access data (all data can be downloaded by the client application from a specific URL using tools like *curl*, *wget* etc).

In this arrangement, a multithreaded client application (written in Python) pulls the data from Amazon, optionally caches it on the local file system, parses the *csv* format, and feeds it to the Cassandra cluster.

6. Characteristics of the cluster used in the project

In order to achieve optimal performance, Cassandra makes full use of the resources of the machine on which it is deployed. It is therefore not optimal to place a Cassandra system on a cluster of virtual machines, since there is nothing to be gained from sharing redundant resources. For that reason, after initial testing on a 4-VM cluster at CERN, a dedicated cluster was built at Brookhaven National

Laboratory, with the following characteristics: 3 nodes with 24 cores, 48GB RAM and 1TB of attached SSD storage each. Its first version relied on rotational media (6 spindles per node in RAID0 configuration), but testing showed that it didn't scale up to performance levels of the PanDA Oracle instance, therefore storage was upgraded to SSDs. Main results of testing will be given below. The Oracle cluster used in PanDA had a total spindle count of 110 (Raptor-class rotational media). Its performance would doubtless benefit from using SSDs as well, but clearly at this scale this wouldn't be economical.

It's worth noting that Cassandra use of resources is radically different between read and write operations – the former is I/O bound, while the latter is CPU intensive. Depending on application, it's important to provide sufficient resources in each domain.

Monitoring load on the nodes as well as client machines that load and index data on the cluster is an important part of regular operations, troubleshooting and maintenance. We used the Ganglia system to accomplish that.

7. Indexing

Efficient use of data makes it necessary to create an effective indexing scheme in the data. Analysis of actual queries done against PanDA database shows that in addition to a simple row key query they often contain selection based in values in a few columns. This knowledge can be utilized in building a Cassandra-based application, by creating composite indexes mapped to such queries. This can be done in one of two ways:

1. Relying on "native" indexing in Cassandra
2. Creating additional lookup tables by a separate client application

If one goes with the former, it effectively necessitates creation of composite columns in the column families. This is akin to duplication of parts of data and leads to inflation of disk space associated with the data. On the other hand, there is no extra code to be written or run against the database, as the cluster will handle indexing transparently and asynchronously.

Producing indexes "by hand" (the second option) gives the operator complete control over when and how the data is indexed, and saves disk space since no extra columns are attached to column families. In the end, driven by consideration of referential integrity when retiring parts of data, it was decided to use the former option in creating indexes, trading disk space for ease of operation and maintenance.

Indexes were designed as follows: we identified queries most frequently used in the PanDA monitor. As an example, there are queries that select data based on the following job attributes: *computingsite, prodsourcelabel, date*. A new column is created in each row, with a name that indicates its composite nature: "**computingsite+prodsourcelabel+date**". The value of such column may then look like "ANALY_CERN+user+20110601". This augmentation of data can be done either at loading stage, or by running a separate process at a later time, at discretion of the operator. When and if this column is declared as index for Cassandra (which is typically done via its CLI), the cluster will start building corresponding data structures, automatically and asynchronously. To utilize the index, client application must determine that the user's request contains requisite query elements, and then a query is run on the cluster by selecting values of the composite column according to the user's request. Seven indexes constructed in such a way cover a vast majority of all queries done in the Monitor. The remaining small fraction can be handled by using individual "single" column indexes in combination, since this is also allowed in Cassandra. It won't be as fast as a composite index since it will involve iterations over a collection of objects instead of a seek and fetch, however preliminary testing shows that the performance will still be acceptable, given relative scarcity of such queries.

8. Results of testing

A few client applications, written in Python, were created for purposes of data loading and indexing, as well as performance and scalability tests. The Cassandra cluster was populated with real PanDA data, and the data load was equal to one year worth of job records, covering the period of June 2010 to

May 2011, in order to be approximately the same as in testing done with previous configurations. Based on observed query patterns, two main categories of performance and scalability tests were performed, along with comparisons with analogous queries against the original Oracle database:

- Random seek test, whereby a long series of queries were extracting individual data for randomly selected jobs from the past year data sample
- Indexed queries, when a number of representative complex queries were performed using composite indexes as described in the section above

In order to judge the scalability of the system, queries of both classes were run concurrently in multiple client threads, simulating real operational conditions of the database. Our capability to fully stress the cluster was limited to approximately 10 threads, by the available CPU resources on the client machine, however the results extracted provide useful guidance nevertheless.

In the random seek test, Cassandra's time per extracted entry was 10 ms, with 10 concurrent clients. This translates into 1000 queries per second, which is about twice the rate currently experienced by our Oracle database. An analogous test done against Oracle (from a Python client) yielded roughly 100ms per query.

In the case of indexed queries, Cassandra's time per entry was 4ms. This result is counterintuitive at first, when compared to the random query metric, because indexed queries result in additional disk operations (the index must be read before the actual data extraction). However this result makes sense if one considers that multiple rows extracted in the query were packaged more efficiently in the network layer during data transmission from the cluster to the client, i.e. effective overhead per entry was lower. A comparable Oracle test returned a range of values from 0.5ms to 15 ms.

The queries that were run for comparison purposes against Oracle and Cassandra were effectively identical in scope and logic, i.e. full content of the row was pulled from the database in each case.

When increasing the load further, we found that the Cassandra cluster will adequately handle loads of at least 1500 queries per second, which is approximately 3 times more than current query load in the PanDA Monitor. Actual timing results were less relevant because of the client machine limitations mentioned above, i.e. we were not able to reach stress limits of the Cassandra cluster at this time.

9. Conclusions

We identified a "noSQL" database system Cassandra as a promising technology platform to ensure scalability and performance of the PanDA monitor database, operating under conditions of consistently high data and query load. A few versions of data design and indexing were evaluated. Hardware requirements were confirmed via testing of the Cassandra cluster under realistic conditions and benchmarking it against Oracle RDBMS. A simple but effective technique of composite index creation was employed to guarantee high performance in most popular queries generated in the PanDA monitor. Quantitative scalability and performance test results are favorable and pave the way for integration of this new noSQL data store with PanDA monitor.

References

[1] Maeno T 2008 PanDA: distributed production and distributed analysis system for ATLAS *J. Phys.: Conf. Ser.* **119** 062036

[2] Nilsson P 2008 Experience from a pilot based system for ATLAS *J. Phys.: Conf. Ser.* **119** 062038

[3] Hewitt E 2010 *Cassandra: The definitive guide* (O'Reilly Media)