Studies of Big Data metadata segmentation between relational and non-relational databases

2015 J. Phys.: Conf. Ser. 664 042023

(http://iopscience.iop.org/1742-6596/664/4/042023)

Journal of Physics: Conference Series **664** (2015) 042023 doi:10.1088/1742-6596/664/4/042023

Studies of Big Data metadata segmentation between relational and non-relational databases

M V Golosova[1], M A Grigorieva[1], A A Klimentov[2], E A Ryabinkin[1], G Dimitrov[3] and M Potekhin[2]

[1] National Research Centre "Kurchatov Institute", Moscow, Russia
[2] Brookhaven National Laboratory, Upton, NY, USA
[3] CERN, Geneva, Switzerland

Email: marina.golosova@cern.ch, maria.grigorieva@cern.ch

Abstract. In recent years the concepts of Big Data became well established in IT. Systems managing large data volumes produce metadata that describe data and workflows. These metadata are used to obtain information about current system state and for statistical and trend analysis of the processes these systems drive. Over the time the amount of the stored metadata can grow dramatically. In this article we present our studies to demonstrate how metadata storage scalability and performance can be improved by using hybrid RDBMS/NoSQL architecture.

1. Introduction

Scientific computing becomes more and more data intensive nowadays. Experimental science produces vast volumes of data (hundreds of PBs) to be managed and analyzed. In High Energy Physics (HEP) data are distributed around the world in more than 120 centers. In order to provide transparent access and data integrity, to manage the data processing workflow and balance utilization of computing resources, large experiments use dedicated software services – Distributed Data Management (DDM) and Workload Management Systems (WMS).

These systems produce metadata that describe actions performed on jobs, stored data and other entities. One of the most successful systems developed in HEP is PanDA (acronym for Production and Distributed Analysis) [1], used by thousands of physicists in the ATLAS (A Toroidal LHC ApparatuS) [2] experiment at the LHC.

PanDA was adopted ATLAS-wide in 2008 for all computing applications and is currently used to manage all experiment-specific workflow for data processing, simulation and analysis. It provides location transparency of computing resources and experimental data, processing yearly more than 200 million jobs on over 140 thousand job slots worldwide. Recently, the system started to evolve in the direction of unification and integration of new computing resource types. The project carried out at National Research Centre "Kurchatov Institute" (NRC KI) aims to provide higher level of services for workload and data management, reusing existing components of PanDA when possible. MegaPanDA, the system being developed, will make PanDA WMS available for non-HEP, extend it beyond the LHC Computing Grid [3][4] and integrate with Distributed Data Management service.

A central component of PanDA architecture is a database, which at any given time reflects the state of job payload and stores a variety of vital configuration metadata. Its growth rate has significantly increased over the last years: from 500 thousand jobs per day in 2011 up to 2 million nowadays. Our

research is focused on scalability and performance improvements of the PanDA metadata store by using heterogeneous (hybrid) storage consisting of both relational and non-relational databases.

2. PanDA architecture and associated metadata

PanDA utilizes a concept of pilots, lightweight wrappers for the job payload: they occupy computing slots at target sites, report back to the central scheduler and pull jobs to be executed. Information about all current jobs is stored in PanDA central queue and the input data for them are managed by ATLAS DDM. The latter currently stores more than 160 PB of experimental and simulated data distributed across more than 120 sites all over the world.

2.1. Metadata storage

Each submitted job is tracked by the PanDA database. While the job is being executed, its records are updated in real-time to reflect different job state transitions, errors and other events. PanDA server does no metadata updates after the job is done, but the system doesn't throw away information about finished jobs as it is useful for statistical analysis of recent workflows, detection of faulty resources, prediction of future usage patterns and other analytical activities.

Storing all the metadata is a hard task: PanDA manages large computing system handling around 1.5 million jobs per day. Since the expansion of main database tables heavily affects daily work of the system, it was decided to move "historical" part of metadata to its own tables: storage engine splits job records into active and archive parts. Both of them are used by the Web-based application called "PanDA monitor" to visualize the state of current and historical jobs, tasks, datasets and perform run-time and retrospective analysis of failures on all of the used computing resources.

2.2. Historical metadata and NoSQL

Presently relational database management system (RDBMS, either Oracle or MySQL) is used for both parts of the storage and after a certain period of time metadata are migrated from active to archive part. The latter is used only for analytical activities that need no real-time processing capabilities. Nevertheless, as its volume grows the underlying software and hardware stack for the database engine hits its limits and this severely harms analytical possibilities of PanDA.

Since the archived part possesses "write once, read many" property, we have investigated a modern class of database technologies commonly referred to as NoSQL databases. These technologies are focused on scalability and data availability, sometimes at the expense of consistency and/or atomicity. They suggest BASE[1] standard instead of ACID[2], which is guaranteed by currently-used RDBMS. While NoSQL can't fully replace relational databases for the active metadata, it seems to be quite good for the archive part, where strong consistency is not so essential as sheer performance and scalability.

Main focus of the current work is two-fold. First, we wanted to prototype NoSQL-based archive part for PanDA metadata storage, the concept of hybrid backend. Second, we need to understand if the specifics of NoSQL approach are worth the trouble of introducing it into such a system: will the benefits (like horizontal scalability, lower hardware and operational costs, ability to handle higher data volumes) outweight problems associated with careful request planning, data denormalization and additional complexity of system code.

3. Hybrid Metadata Storage framework

Metadata segmentation between relational and non-relational databases yields two main tasks: synchronization between active and archived parts and PanDA adaptation to the new storage backends. We had prototyped a heterogeneous framework – Hybrid Metadata Storage (HMS) that

[1] Basic Availability, Soft-state, Eventual consistency
[2] Atomicity, Consistency, Isolation, Durability

consists of two corresponding parts. First of them, called "HMS/sync", performs data migration; the second, called "HMS/access", adds NoSQL-related logics to the PanDA monitor.

The very first question was the choice of NoSQL technology and implementation. We decided to look only at mature NoSQL projects to avoid having experimental HMS prototype on the experimental database backend. Currently main players in the general-purpose NoSQL database field are Apache Cassandra, HBase and MongoDB. For prototyping we had briefly analyzed existing experience for these software products and chose Cassandra, since it is sufficiently simple and robust, provides all needed functionality for metadata storage and is field-proven.

Nevertheless, we had also studied architecture and characteristics of HBase and MongoDB focusing on applicability of our approach to these products. Below we will summarize our findings, focusing on Apache Cassandra.

3.1. Cassandra

Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple data centers with asynchronous masterless replication allowing low latency operations for all clients [5].

Cluster architecture. Cassandra has the Google-like architecture that takes for granted that system and hardware failures can and will occur. It addresses the problem of failures by employing a peer-to-peer distributed system across homogeneous nodes where data is distributed among all cluster participants, and each server every second gossips its status information to the neighbors.

Cassandra also supports multi-datacenter replication with clusters that can either be used for active fallback or to split various activity types (writing, analytics, searching) between different clusters. For the current needs of HMS, the former scenario is foreseen to be useful.

Data storage and access. Every record in database is identified by a unique (composite) primary key that is divided in two parts. The mandatory one is called partition key; it governs the physical location of the record. The rest of the components are called clustering keys – they determine location of the record within the partition acting as indexes for sorting (Fig. 1).

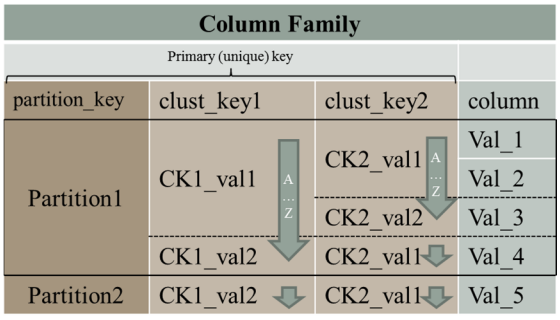

Figure 1: Cassandra column family

Such a structure dictates a data access algorithm. Since the physical location of the record is determined by its partition key, one must explicitly specify this key to get related data (or it would cost a full scan of all stored data, which is unreasonable for a system being engineered to work with Big Data).

Cassandra also wants fast searches within partitions. For key-ordered data the fastest bulk request is a range query – thus only requests for continuous ranges of clustering keys are permitted: querying by any of the clustering keys without specifying values of the preceding ones will produce an error.

Data are stored on disks in SSTable (Sorted String Table) files along with the index of associated partition keys. SSTables are immutable: they are written only once and are never modified in-place, allowing Cassandra to eliminate random I/O to a large extent, greatly improving query speed.

When a client writes data to the cluster, they go to an in-memory structure called memtable, which resembles a write-back cache. Every node stores a commit log capturing write activity to ensure transaction durability – in case of a failure memtable will be recovered next time the server goes up. Once the memtable is full, it is flushed to disk as an SSTable. Collection of SSTables can be compacted: immutability leads to obsolete rows stored inside old SSTables. Garbage collection

removes them, occasionally merging multiple SSTables to bigger ones eliminating the need to crawl over many files on each query.

3.2. HBase

HBase is a column-based NoSQL database being very similar to Cassandra in terms of data storage and access organization (since they have a common ancestor, Google BigTable [6]). But their approaches to clustering differ significantly: HBase has master/slave architecture with distributed coordination provided by ZooKeeper and stores data files in HDFS [7], a distributed file system.

For our prototype HBase was considered to be more complicated in installation and operations, but since the main concept of data organization is the same, everything done for Cassandra can be ported to HBase.

3.3. MongoDB

MongoDB is a document-oriented NoSQL database with a rich set of search and analytics tools, providing possibility to organize data almost without regard to future requests. Still, without indices MongoDB will perform full scans, so for fast requests an appropriate index is needed. Though as indices store values only for indexed fields, to get the rest of the data MongoDB will have to read all the matching documents from disk or memory (for already cached documents). It can be inefficient for reading a large volumes of data: index provides fast search of matching document, but does not offer efficient data access [8].

4. Hybrid Metadata Storage Framework implementation

To build the prototype we have identified the slowest part of the PanDA monitor to adapt it for the hybrid storage. The well-known offender is the “Errors” page: it delivers information about jobs that have not finished correctly in a given period of time, combining this information into 4 summaries and a histogram (failed jobs vs. time). This page is vital for identifying problematic sites and/or bad jobs. Initially its generation took considerable time even for requests like “show all errors for today”. So we decided to start with this page to see what can and should be done to improve its usability.

4.1. HMS/access implementation

PanDA monitor uses Django framework that currently doesn't support NoSQL database backends. Fortunately there is Django-nonrel project that extends database abstraction layer with NoSQL interface and offers plugins for Cassandra, MongoDB and Google App Engine [9]. So first we had completely moved PanDA monitor to Django-nonrel.

Then we modified Django view generating “Errors” page to display only one of the summaries and a histogram obtaining metadata from Cassandra. Other summaries have the same data organization, query types and comparable number of entries, so it is trivial to extend our approach to them and estimate the total page generation time from results for just one summary. As we haven’t copied additional tables into the archive, HMS/access sends some requests to the SQL part. But it is a realistic scenario, as the former tables do not grow much and their proper place is in the actual part (that belongs to the SQL-based backend).

NoSQL storage schema. Currently we have in Cassandra a main archive column family (CF) “Jobs” that duplicates SQL table “JobsArchived”. It is used to build several secondary request-specific CFs: “day_site_errors” (summary) and “day_errors_30m” (histogram).

To improve query performance, we added data aggregation. Instead of fetching all records for failed jobs and then creating error summaries in Web application, we precalculate summaries for 30 min. intervals and then put them to the new NoSQL table “day_mtime_site_errors_cnt_30m” (summary with aggregation). Such optimization is useful even for relational database, but for NoSQL, where we have no requirements for normalized schema, it is a natural procedure to perform.

Cassandra has a limitation on the partition size (partition can't store more than 2 billion cells[3]), so we had to think carefully about partition key (PtK). For "Jobs" it was not a problem, as we used pandaID for the PtK, but for secondary CFs we decided to partition by date. It is a safe approach as long as PanDA handles less than 1 billion jobs per day (even for worst case when all jobs are failed) and there's less than 14 M of possible combinations of the site name and error code (that gives us the ability to handle around 10000 sites if total count of distinct error types stays below 1000).

Primary key for "Jobs" is pandaID (which is also a PtK); this CF stores all the metadata fields (more than 90). The only possible query is by pandaID.

"day_site_errors" has compound primary key: date (PtK), computingsite, errcode, pandaID. It stores one more column: errdiag (error description). Possible queries are: by a given day; by the day and (a set of) computing site(s); by the day, computing site and (a set of) error code(s). It is also possible to get information for a given pandaID (specifying all the preceding keys), but it would be easier to query "Jobs" column family.

"day_mtime_site_errors_cnt_30m" also has a compound primary key: date (PtK), base_mtime, computingsite, errcode, errdiag. It stores two counters: err_count (number of errors) and job_count (number of errored jobs). Possible queries for this CF are: by given day (it gives number of errors within every 30 min. interval); by (a set of) 30 min. interval(s); by the interval and (a set of) computing sites; etc.

Primary key for "day_errors_30m" is a combination of date (PtK) and base_mtime. This CF stores number of the errors for each 30 min. time bin. Possible queries here are: number of jobs with errors within certain bin or throughout a given day.

4.2. HMS/sync implementation

When all the data is kept in a single (relational) database, we can use its internal procedures to move data from active to archive part. But when the archive part goes into NoSQL database, there's a need for synchronization mechanisms: first, we need to copy already existing archive (metadata for ~900 M of jobs), and then to keep the archive up-to-date with respect to the new finished jobs.

For these purposes a Python library called "Storage" was developed. It provides unified wrappers for NoSQL and SQL databases (hiding details about database engine actually used), allowing a smooth transition from one NoSQL implementation to another one (in case we find out later that Cassandra doesn't fit requirements of PanDA, or for any other reason it will be decided to use another NoSQL engine).

Synchronization mechanism implemented in "Storage" synchronization is time-based. The flow diagram for synchronization and precalculation (synchronization of the tables within the NoSQL database) is shown on a Fig. 2.

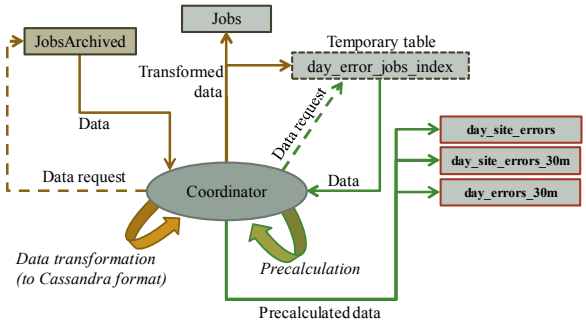

Figure 2: Synchronization and precalculation process

"Coordinator" starts the process of initial synchronization by querying SQL server and stores results in the internal database-neutral format. Then it transforms received set of data to the form Cassandra can consume and sends it to the various NoSQL tables ("Jobs" and temporary index CFs used for precalculation). This process is repeated until there's no more data in SQL matching the initial synchronization parameters (e.g. time interval).

Precalculation goes in almost the same way: coordinator node (not necessarily the same as

[3] **Cell** is a unique combination of values of all clustering keys and a one of the rest of the fields (partition key isn't included, since it uniquely determines row where these cells are stored). Basically the number of cells per row is equal to the number of non-empty columns for its primary key plus 1 for the hidden timestamp cell.

above) sends the request to the already-populated temporary index CFs, transforms the data for the request-specific CFs and their aggregated counterparts and subsequently flush them to the persistent NoSQL tables.

Import of 1 M rows from Oracle took ~16 min, so we can synchronize around 40 M records in a day. It allows us to scale up to 10 times comparing to the current rate of historical metadata flow from daily PanDA production activities. Precalculation for 1 M rows took ~6 min, thus we are on the bright side of the future scaling picture too.

Both actions can be done in a single export/precalculation cycle and then will not require any temporary CFs. But since we transfer large volumes of data via the network, it is better to run synchronization closer (network-wise) to the Oracle server, and precalculation – closer to the Cassanda cluster. Also, such separation allows us to do both procedures in a pipeline-like fashion to further speed up things.

Estimated time of daily synchronization for ATLAS (presently ~1.5M jobs per day) is ~35 min.

5. Prototype testing and results

5.1. General considerations

We has measured time for various parts of PanDA monitor page generation logics. Tests were not limited to stand-alone database queries: data transformation cycle was included in the measurements. It was necessarily due to significant differences between DB schemas in SQL and NoSQL. Obtained measurements have statistical significance: all tests were performed at least 10 times and then analyzed to have normal distribution with small dispersion (since in fully-deterministic system we ought to have just spot-on results).

5.2. Layout of a testbed

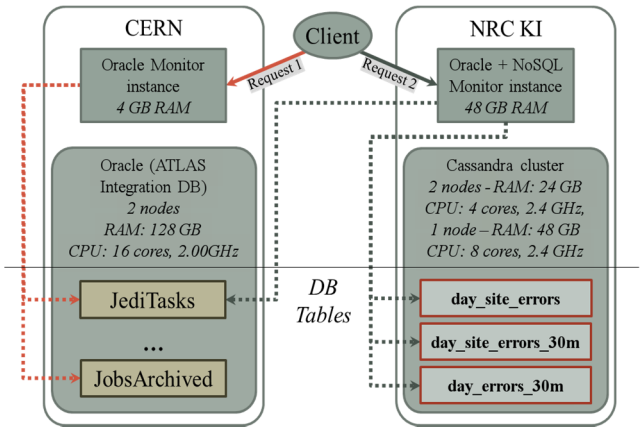

Figure 3: Scheme of the test set-up

To ensure database access locality we had installed two instances (Fig. 3): one was deployed at CERN and used pure SQL metadata backend (ATLAS Oracle-based Integration Database replicated from main ATLAS production database). The second instance was set up at NRC KI in the closest proximity of Cassandra cluster. Both instances were running the same (modified) source code for PanDA monitor, but almost all the modifications were related to the "HMS/access" NoSQL parts, so for CERN's instance it was effectively running the same code as usual production instances do.

5.3. Obtained results

In the first test we have checked the scalability of NoSQL: we studied the dependence of query execution and page generation time on the total amount of data stored in Cassandra. Measurements were performed for different number of rows matching the query.

We had created two test sets: 7.8 M and 13.8 M rows in request-specific CFs (from 30 M and 62 M rows in the full archive CFs).

Test results (Fig. 4) show that query execution

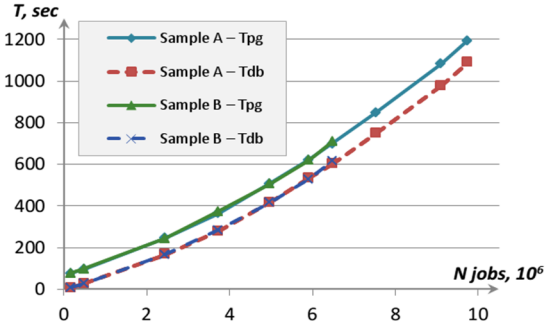

Figure 4: Page generation (T_{pg}) and database request (T_{db}) time

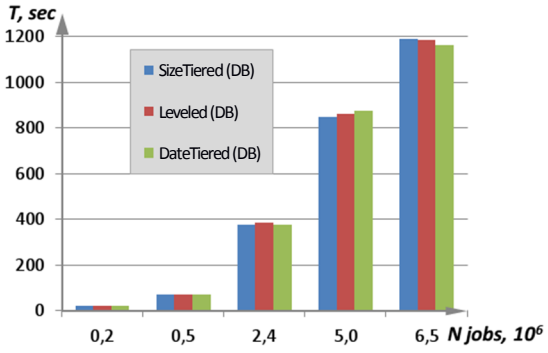

Figure 5: Comparison of compaction strategies

takes the same time for both sets, and T_{pg} is almost a linear function of the number of jobs matching the request.

Next the compaction strategies test was performed. Cassandra has a number of algorithms to manage SSTable files on disk called compaction strategies [10]. To check if the choice of the strategy has any effect on the read performance, we had created three keyspaces with same data but different compaction types. Tests revealed that for up to 7 M of jobs matching the request there's no significant difference (less than 5% for the first requests and less than 1% for the last, most massive requests) in read performance (Fig. 5).

This is a very good result, since the completion of size-tiered compaction requires up to the size of the largest SSTable to be free at each node. For the system working with large amounts of data it can be critical. So we can use leveled compaction (that requires less additional space) without sacrificing query performance.

The next test quantifies the effect of data aggregation. Results show that for 30 min. aggregation interval generation of "Errors" page takes almost 4 times less (Fig. 6). The speed-up was mostly achieved by reducing the number of records stored in database (and, thus, read by application): non-aggregated table had 7.8 M rows, and aggregated held only 1.9 M. And requests for a longer periods (months, years) can use more compact (daily, weekly) summaries to make things even faster.

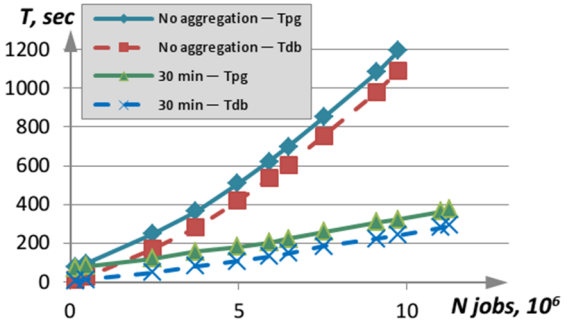

Figure 6: Page generation (T_{pg}) and DB request (T_{db}) times for non-aggregated and aggregated data

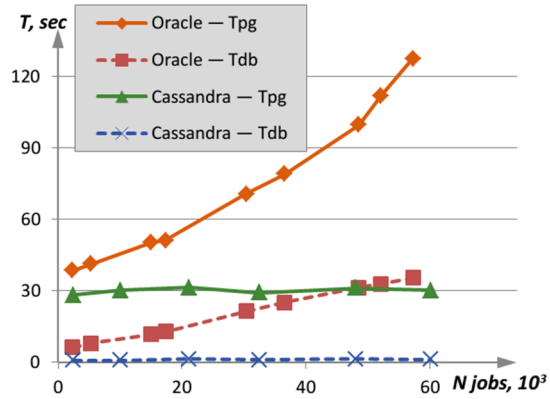

Figure 7: Performance of NoSQL archive prototype

Finally, we have compared performance of the two PanDA monitor instances (Fig. 7); for NoSQL flavor precalculated and aggregated data were used.

One can see that request and page generation times for NoSQL archive are almost constant (for a studied range of data), while theoriginal PanDA monitor slows down significantly. Basing on the results of scalability test we can predict that NoSQL part will not stay constant for a wider data range, but still results look promising.

6. Conclusions and Future Work

This work was devoted to finding ways of improving the analytical abilities of PanDA monitor software taking into account fundamental properties of historical job metadata. Using Apache Cassandra, the storage engine that can reliably handle large amounts of data and scales horizontally sufficiently well for our task, we had studied the possibility of using BigTable-based NoSQL technologies exploiting specifics of the stored information and the way PanDA monitor works with it. A concept of hybrid metadata storage was architectured and to prove its feasibility we had ported the worst-behaving part of the monitor to that framework.

Our tests yield the following conclusions:

- Cassandra shows a good scalability (for the month-worth timescale we were testing at), which is very important for systems like PanDA, where the archive database hosts metadata for more than 900 M jobs today and grows every day for 1-2 M of records.
- Column-based NoSQL storage, such as Cassandra or HBase, may seem to have too severe restrictions on data organization and querying, but query adoption is possible and it just makes us to think carefully about the data layout, basing on what do we want to use them for.
- It may look like we have to store an excessive number of copies for the same data, but actually it's almost the same as if we had a number of complicated indices on one table in SQL or document-oriented NoSQL engine.
- Carefully organized data and other elaborations (that come very natural with NoSQL design practices), such as advance data aggregation and precalculation, provide significant performance improvement without adding much complexity to the resulting system.

This makes us to believe that the idea of creating NoSQL-based metadata archive for PanDA proved to be a healthy one, but still there are a lot of things to be done. Our future work items are:

- to implement functionality that will cover the rest of SQL-based logics for analytics;
- to perform more scalability and real-world usage tests on a wider range of data;
- to verify our assumptions on how other NoSQL engines will perform in place of Cassandra;
- to integrate the resulting code with the production PanDA monitor instances.

Acknowledgements

We wish to thank all our colleagues from the ATLAS experiment, PanDA team and CERN IT.
This work was funded in part by the Russian Ministry of Science and Education under contract № 14.Z50.31.0024. Testbed resources at NRC KI are supported as a part of the centre for collective usage (project RFMEFI62114X0006, funded by Ministry of Science and Education of Russia).

References

[1] Maeno T. et al. 2012 Evolution of the ATLAS PanDA Production and Distributed Analysis System, *Proc. Int. Conf. on Computing in High Energy and Nuclear Physics, J. Phys. Conf. Ser.*
[2] ATLAS Collaboration 2008, The ATLAS Experiment at the CERN Large Hadron Collider, *J. Inst.* 3 S08003
[3] Panitkin S. et al. 2015 Integration of PanDA workload management system with Titan supercomputer at OLCF *Proc. Int. Conf. on Computing in High Energy and Nuclear Physics, J. Phys. Conf. Ser.*
[4] Berezhnaya A. 2015 et al Integration of Russian Tier-1 Grid Center with High Performance Computers at NRC-KI for LHC experiments and beyond HENP. *Proc. Int. Conf. on Computing in High Energy and Nuclear Physics, J. Phys. Conf. Ser.*
[5] Apache Cassandra, http://planetcassandra.org
[6] Chang, Fay, et al. 2008 Bigtable: A distributed storage system forstructured data. *ACM Transactions on Computer Systems* 26.2: 4
[7] Shafer J. and Rixner S. 2010 The Hadoop distributed filesystem: balancing portability and performance *Proc. IEEE International Symposium on Performance Analysis of Systems & Software*, pp.122-133
[8] MongoDB, http://docs.mongodb.org/master/MongoDB-indexes-guide.pdf
[9] Django-nonrel, https://github.com/django-nonrel
[10] Apache Cassandra compaction types:
http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra
http://www.datastax.com/dev/blog/datetieredcompactionstrategy