Journal of Physics: Conference Series **664** (2015) 042002 doi:10.1088/1742-6596/664/4/042002

Scale out databases for CERN use cases

Zbigniew Baranowski, Maciej Grzybek, Luca Canali, Daniel Lanza Garcia, Kacper Surdy

European Organization for Nuclear Research (CERN), CH-1211, Geneve 23, Switzerland

Abstract. Data generation rates are expected to grow very fast for some database workloads going into LHC run 2 and beyond. In particular this is expected for data coming from controls, logging and monitoring systems. Storing, administering and accessing big data sets in a relational database system can quickly become a very hard technical challenge, as the size of the active data set and the number of concurrent users increase. Scale-out database technologies are a rapidly developing set of solutions for deploying and managing very large data warehouses on commodity hardware and with open source software. In this paper we will describe the architecture and tests on database systems based on Hadoop and the Cloudera Impala engine. We will discuss the results of our tests, including tests of data loading and integration with existing data sources and in particular with relational databases. We will report on query performance tests done with various data sets of interest at CERN, notably data from the accelerator log database.

1. Introduction

CERN and high energy physics in general, have developed over the years many techniques to store and manage large amounts of data. Experiments at the Large Hadron Collider [1], for example, have generated around 30 Petabytes of data annually during run 1.

At the same time Terabytes of data are produced and stored daily by various monitoring, measurement and control systems that supports the operations of LHC itself and of the many online systems for the LHC experiments.

Unlike physics data, which are stored on tape and on disk-based file systems, controls data coming from LHC subsystems are inserted into relational databases. Thanks to the structured data model and usage of indexes this model has the advantage of allowing quick access to particular data of interest. Also building visualization and management application on top of data stored in a Relational Database Management System (RDBMS) is relatively easy to implement, deploy and maintain. At the same time storing data on a dedicated database installations means that those systems are critical for LHC and have to be highly available, reliable and provide the needed performance level. A potential problem when deploying very large databases on many of the available RDBMS engines, for the use cases of running analytic workloads is with the performance of sequential I/O. In those cases sequential access to the database has an intrinsic bottleneck at the level of the storage connection. To work around such bottleneck the deployment of specialized hardware may be necessary to scale up the database hardware infrastructure and reach the required performance levels.

1.1. Very large relational database system at CERN

The largest production database currently deployed on a RDBMS engine at CERN is the LHC logging service [2], which is a consolidated system for data coming from over 30 subsystems that are part of the LHC infrastructure (like Cryogenics, Vacuum, Quench Protection System). This database has grown to a size of over 250TB from 2008 till 2015. The data rate is expected to increase during the coming years, as new data sources have been added during the recent LHC maintenance period. As a result, the current insertion rate into the database is 50 thousands data points per second, which makes it growing by more than 100GB every day. Some estimation says that after restart of LHC in 2015 the LHC logging service will be adding 90TB annually.

The SCADA systems [3] that control and monitor all the most important hardware elements of LHC and experiments are also important sources of data for relational databases deployed at CERN. Historical values of all data points are stored in the databases at a relatively high rate. For example the Quench Protection System [4] can generate 150 thousand insertions every second. This results in data growth of about 2TB per day. Data are kept for 40 days and are subsequently dropped after being processed, filtered out and logged in the LHC logging system. Analysing data in real time has the additional challenge that queries should be performed with a very low footprint in order not to interfere with the performance of the critical write workload to the database.

The time series data being produced by controls and monitoring systems and stored in Oracle databases at CERN will typically make use of special structures called Index-Organized Tables (IOT) [5]. This amounts to storing the table key and data in a B+tree index structure and has the advantage of reducing the amount of I/O requests on the storage when retrieving data by a key, therefore increasing the performance and scalability of the system. In order to keep the IOT segments to a manageable size another technique is used: partitioning [6]. Very often data that has been produced within a single day are stored in a dedicated partition. Typically the index key for the time series data in these databases is compounded of two columns: an identifier (variable id) and a timestamp. This means that the data blocks in an index are optimally stored for the retrieval certain variable values from a specified time range. This is because values of the same 'variable id' will be stored by the database engine in close physical proximity and also ordered by time.

1.2. Ad-hoc data analysis

Single-variable look-ups are very fast when using IOTs, as described above. However this approach does not provide any performance advantage when performing ad-hoc analytical, statistical or report queries that normally have to process multiple or even all variables in a data set. Mainly because such data access requires sequential scanning of the full data set instead of traversing through an index for each variable in the data set. Sequential scanning of a full partition delivers better performance when reading large amounts of data as in such case the storage receives aggregated asynchronous multi-block IO requests that can be handled in an optimal way by a database and in the end deliver very high data reading throughput.

1.3. Sequential data access on CERN infrastructure

Sequential data scan operations deliver high throughput for data processing and are normally bound by limits of a database server or storage hardware configuration or whatever medium is between both of them for data transport. At CERN most of the database installations are clustered using Oracle Real Application Clusters [7] technology. The use of clusters (typically of two or three nodes) has an important advantage for availability of the database by allowing many maintenance operations to be performed in a seemingly transparent way. However this approach implies significant limitations on the data access throughput due to the requirement of using shared storage among the cluster nodes. In particular the speed of the connection between the database server nodes and the storage can be a bottleneck in this architecture, especially when the requirements for throughput increase. In the computing infrastructure currently deployed for database services, connectivity to storage is deployed on 10 Gigabit Ethernet, which limits the throughput that a single node can support to about 1GB per

second. Moreover the storage units themselves have internal limits on the throughput (of the order of a few GB per second, depending on the model). This obviously has its consequences on the speed at which big data stored in a database can be processed. One Gigabyte per second in contrast to hundreds of Gigabytes stored every day is not a speed that can deliver performance in many of the related data analytic use cases. Of course this situation can be improved by scaling up all the components including the network. However there are always technological upper boundaries for each single element of the shared architectures that will not let to go beyond certain throughput values.

In this paper we will discuss potential solution for achieving highly scalable performance (beyond 1GB/s) for SQL (and not only) queries by integrating relational databases with platform using the shared nothing architecture, notably Hadoop.

2. Hadoop-based solution for scale out

Hadoop [8] is currently the most representative platform in a new wave of scalable systems for dealing with 'Big Data' problems. Thanks to simplified data models and specialized architectures these systems are able to process data at very high speed and at the same time to offer a variety of interfaces to the data.

2.1. Hadoop as an open platform

One of the peculiarities of Hadoop is that it is not a single product, but rather a framework or an ecosystem that consist of many components that can work together or may co-exists independently. The core component of the Hadoop ecosystem is the Hadoop Distributed File System (HDFS). HDFS provides a file store layer for all other components. Such open architecture centralizes around a file system and makes Hadoop unique compared to other platforms in the 'Big Data arena'. There is a large and thriving developer community that contributes to the Hadoop project by adding components that extend the system with new features and functionalities or just improve the already existing ones.

Currently Hadoop offers multiple ways for storing and processing data located on HDFS. The most popular data analytic approach available on Hadoop is MapReduce [9]. Apache Spark [10] is a modern, improved and more efficient successor of the MapReduce. Imperative interfaces are available both for Spark and MapReduce. Typical languages used in this area include Java, Scala and Python. The SQL-language interface for declarative data processing is also available and implemented in a few specialized engines. Apache Hive [11] was one of the first implementation of an engine for running SQL on Hadoop by translating queries into MapReduce jobs. A newer and more efficient generation of SQL engines for massive parallel processing (MPP) include Cloudera Impala [12], Apache Spark SQL [13] and the extended Hive functionality obtained by running Hive on Tez framework [14]. The mentioned mainstream data processing engines can work with multiple data file formats that are available on HDFS, this lets users choose the most suitable approach for their data and application design. It is not uncommon to store data in a semi-structured text format like CSV. However more sophisticated and demanding systems are very likely to use binary data formats like SequenceFiles [15], RCFiles [16] (or the improved version, OCR [17]), Avro [18] or Parquet [19]. Finally data files on HDFS can be compressed in order to reduce their data volume.

A great variety of available techniques and tools for data processing and storage make Hadoop a strong solution for consolidating and handling data coming from many sources and for running analytic workloads.

2.2. Shared nothing for scale out

The Hadoop architecture was designed to deliver scalable performance: the more machines in a Hadoop cluster, the better the overall performance of data processing. Shared nothing architecture is a key for achieving scalability and ultimately the needed levels of performance: data are partitioned into equal shards and distributed evenly across all nodes of a Hadoop cluster. Whenever data stored on HDFS is to be processed, all the processing logic is distributed and applied locally on all the data shards. In a subsequent step the products of such processing are consolidated and/or aggregated to

output the final result. This approach of running the initial data processing locally on server nodes can reduce considerably the amount of data that has to be shipped between cluster machines. This is a clear advantage for scalability in comparison with the shared storage architecture.

As an example of the potential of this architecture we would like to mention the tests done at CERN in 2013 [20] which showed near-linear scalability of a Hadoop cluster of 50 commodity servers. Each node of the cluster had 3 local disks attached and the setup scaled up to the overall throughput for sequential I/O to almost 9 GB/s. Overall, Hadoop is a very good candidate for building solutions for data warehousing and analytic workloads given its flexible architecture capable of handling diverse workloads in combination with the scale-out capability that provides the required performance level.

3. Using of Hadoop for accelerating data analytics at CERN

Given the variety of available solutions a series of tests and investigations have been performed to find a right set of technologies suitable for processing large time series datasets of interest for CERN use cases. The format in which the data is stored in the file system, i.e. in HDFS in Hadoop, impacts the performance of the analytic framework. Similarly, the choice of compression algorithm affects both storage size and performance by reducing the amount of data that have to be read and transferred. This comes as a trade-off with an additional load on the CPU for compression and decompression operations.

3.1. Data formats

Several data formats representing different design approaches are available in the Hadoop ecosystem. The most basic data format is the *CSV* file, where each row is stored as one line with fields separated by a delimiter, usually a comma. The simplicity of this format makes it very easy to read/write from/to it and to use it as a format to exchange data between applications. *SequenceFile* is another file format, which contains binary-encoded sequence of key-value pairs. *Avro* is a data serialization system. When referring to Avro we mean serialization format for persistent data. The format stores binary data in files together with a schema defined in JSON. *Parquet* stores data in column-oriented arrangement. Files are composed of row groups. Within a row group all values of a given column are stored sequentially. Column and file metadata is located after the data to allow single-pass writing.

3.2. Compression

On all the formats mentioned in the previous paragraph compression algorithms can be applied. The obvious reason for using compression is to lower space usage. In the case of massive processing of data compression can be a mean to accelerate computing as well. When the data is compressed reading time can be lowered at the expanse of increased demand for CPU cycles or memory. With the respect to that comprehensive performance tests have been done – taking into consideration not only compression rates but also decompression speed. The tests have been done using *bzip2* [21], *snappy* [22] and with *no compression*. bzip2 has proven to provide good compression ratios. Snappy was designed to strike a compromise between having good compression ratios and fast compression and decompression operations.

3.3. Data partitioning

Data partitioning can be used to improve the performance of storage-related operations. Horizontal partitioning allows to optimize data access by allowing pruning operations based on the partitioning condition. List-based partitioning, for example, is implemented by clustering data according to a list of discrete values. In the Hadoop platform, for example, partitioning is typically adopted by placing files into separate folders depending on the value of one or more partitioning fields. Respectively, vertical (column based) partitioning allows to read only values from selected columns. This can be achieved by using columnar-oriented storage formats like Parquet. Architecting data layouts where both partitioning techniques are used in combination can deliver the best results thanks to improved selectivity of the data.

3.4. Query engines

For data stored in Hadoop a selection of tools mentioned in section 2.1 can be used. In our evaluation Hive, Impala, and Spark-SQL have been tried.

3.5. Test results and data format comparisons

In this paragraph we report on a series of tests on data formats. Data used for the tests was stored as tuples composed of: source identifier, timestamp, and value. As numerical data is the most commonly used type of data for control systems, we limit our tests to numeric value fields. This is also a choice that allowed to simplify the implementation of the tests.

Table 1 contains a comparison of file sizes for 8 days of numerical data imported from the LHC logging database. The original data volume stored in compressed IOT structures on RDBMS system was 650GB. Avro and Parquet showed to be 2.2 to 2.8 times more space efficient than CSV and SequenceFile for storing uncompressed data. As expected storing data in plain text in CSV format is very inefficient. The cause of large space consumption of SequenceFile is different. In uncompressed SequenceFile the key is repeated for each stored record. The size difference became less substantial after enabling compression. Avro format created the smallest snappy compressed files. CSV appeared to be the most favourable for bzip2 compression.

	no compression	snappy	bzip2
CSV	1240 GB	331 GB	109 GB
SequenceFile	1545 GB	265 GB	117 GB
Avro	542 GB	226 GB	171 GB
Parquet	558 GB	288 GB	-

Table 1. File sizes of 8 days of LHC log numerical data as a function of the data format and compression algorithm used

Table 2 contains the measurements of the run time for querying the data set described above and in Table 1 (i.e. 8 days of numerical data imported from the LHC logging database). The test query performs a full scan of the data set and applies some additional data filtering. At the end the final result set is ordered by time.

The comparison of the execution times on the data sets compressed by snappy with the uncompressed data, illustrates that the beneficial effects of compression. The resulting reduction of the data set size and consequently the reduction of the I/O operations in most cases justify the additional computing complexity introduced by compression. For all the tested formats the execution time was lower on the snappy compressed files then on the uncompressed ones, with the biggest benefit observed on Parquet. A different case can be seen when comparing snappy and bzip2 compressed Avro files. Even though the bzip2 compressed file is smaller, the additional algorithm complexity slows the query execution time to the point that compression is no more beneficial for performance.

	no compression	snappy	bzip2
CSV	757 s	687 s	-
SequenceFile	682 s	572 s	1800 s
Avro	216 s	113 s	118 s
Parquet	328 s	117 s	-

Table 2. Measured query execution times for 8 days of LHC log numerical data as a function of the data format and compression algorithm used

Table 3 shows throughput measurements obtained by running the same test query while varying the SQL engine and cluster size. The data format used is Parquet.

	4 nodes	8 nodes	12 nodes	16 nodes
Impala	0.68 GB/s	1.33 GB/s	1.74 GB/s	2.30 GB/s
Hive	0.34 GB/s	0.55 GB/s	0.73 GB/s	1.22 GB/s
Spark SQL	1.00 GB/s	1.64 GB/s	2.23 GB/s	2.83 GB/s

Table 3. Measured average throughput when reading 8 days of LHC log numerical data as a function of the storage engine type and the cluster size.

The results show that all tested technologies are able to scale out, as the average throughput was increasing with the increasing number of nodes in the cluster. Spark SQL delivered the best overall performance just above Cloudera Impala in our tests and significantly above the performance of Apache Hive. It is important to notice that in all our tests the workload was CPU-bound. We explain the better results obtained by Spark SQL as consequence of Spark SQL being the less CPU-hungry solution for the tested query. Additional tests with servers with more CPU power (namely more cores) are not available at the time of this writing. Notably the results of these tests are promising as they already show higher throughput for sequential I/O than the current generation of shared storage-based RDBMS deployed in production.

3.6. Bucketing for an efficient data access

Even though the Hadoop platform offers high scalable throughput for data access and processing there is a family of queries that will execute relatively slowly compared to RDBMS systems. This applies to the cases when only one or few variables are being accessed. Having data indexed gives a great advantage in such a situations and lets RDBMS systems to access data efficiently by limiting IO requests to the storage. Since SQL on Hadoop solutions like Hive, Impala or Spark SQL do not have indexing features they always have to read whole data partitions. This approach requires a lot of unnecessary I/O requests as later most of the data read will be filter out leaving only a small subset of interest. For example queries on data having 100GB partitions will take long even though the intention is to retrieve few megabytes out of it. The accelerator log service for example has daily partitions of 40 GB when stored in Parquet or Avro formats and reading it by Impala or SQL spark takes around 10 seconds.

Introducing fine-grained partitioning by ‘variable id’ in addition to year, month and day would reduce significantly the size of a single partition and would guarantee efficient data access. However depending on number of variable ids in the data set this solution would produce hundreds of millions of very small files yearly, as each partition is stored in a separate directory and file. Having such large number of objects in a file system would impact negatively the performance. In particular the HDFS Namenode server would require a great amount of server memory for storing the file system map. Such a situation is strongly discouraged by the Hadoop developer communities.

In order to overcome this problem the ‘bucketing solution’ for data organization has been investigated and developed as part of this research. The idea behind the bucketing solution is to group data for multiple variable ids into a single partition, based on the value of a grouping/hashing function. The advantage is to obtain good performance with the finer-grain partitioning, while keeping a low footprint on the HDFS Namenode. The grouping in our implementation of the bucketing solution is obtained with a modulo function: *mod(variable_id, x)*. The partitions obtained this way are in practice sub-partitions of the time-based daily partitons. This method gives a full control over the number of partitioned created every day by adjusting argument ‘*x*’ in the *mod* function.

When grouping daily data into 10 buckets (by using *mod(variable_id,10)*) for the LHC logging service, a significant performance improvement is observed for single-variable data selection: only 4GB instead of 40GB needs to be read from daily partitions, which results in reduction of query execution time by a factor 10 (1s instead of 10s in our example).

3.7. Benefits from a columnar store

Parquet data file format is a structured binary standard for efficient storing of big data sets. Notably Parquet provides columnar organization of data, similarly to RCFiles and ORC formats [16][17]. This means that data for each column is stored contiguously. Thanks to that, data processing can be more efficient when column pruning can be used to minimize the amount of data to be read from storage.

Tests with physics data loaded on Hadoop **Error! Reference source not found.** have proven the efficiency of using columnar store for multi-column data sets. The tests discussed in [23] used a table join resulting in a data set of 1400 columns, but where only 50 columns were of interest in the query. When the source tables were stored in the Avro format the execution of the joining query had to read the full data set, 110 GB. For the same test using the Parquet format, the query read only 4 GB out of 92 GB. This had direct impact on the query execution time; in case of using the Parquet format the measured query execution was 16 seconds, for the case of Avro (row-based storage) the measured execution time was about three times longer: 52 seconds.

4. Integration of scale out databases with current infrastructure

Most of the data of interest for this paper are currently stored in relational databases and have to be loaded into am Hadoop system before they can be accessed and processed. Therefore data integration solutions for Hadoop and RDBMS are very important and have been addressed in our tests. There are already several popular solutions for data loading into HDFS from heterogeneous sources, including RDBMS.

4.1. Apache Sqoop

Apache Sqoop [24] is a project for bulk data loading between relational database systems and HDFS. It is implemented as a MapReduce job and does not only allow storing data into in files on HDFS but also supports direct data insertions into tables defined by Hive, HBase or Accumulo. Source data input can be specified in many ways including listing table names or by providing the output of a SQL query. In order to implement continuous data synchronization between a database and Hadoop, Sqoop jobs have to run periodically as there is no feature of synchronous data replication.

One of the main parameters for Sqoop performance is the degree of parallelism. In our tests we have found that this can significantly impact of the overall extraction throughput. Each Sqoop map task opens a dedicated connection with a database and queries an even portion of the data. This operation is CPU bound on the client side. Sqoop mappers processing is mainly on CPU when data from a source databases are being shipped and stored at the final location on HDFS. In order to speed up the data retrieval Sqoop can be configured to start multiple mapper tasks at the same time. Increasing the number of mappers improves the overall throughput proportionally. It is important also to avoid overloading the source RDBMS systems by running too many concurrent sessions. A compromise between Sqoop loading performance and corresponding load applied on the source database has to be established by starting with a low load and increasing the number of concurrent Sqoop mappers while monitoring the database load. For example with the LHC logging data extraction when using 31 mappers the consolidated throughput was 1.3 Mrows/s (40MB/s) when using just a single mapper was 174 Krows/s (5MB/s).

4.2. Apache Flume

Unlike Sqoop, Apache Flume [25] uses streaming data flows for collecting, aggregating, and moving large amounts of data to HDFS. Flume includes some out-of-the-box sources such as Avro, Thrift or logs, channels as in-memory or file and sinks (destinations) such as file, logger and IRC. It is worth emphasising data loading with Flume is a continuous process. So once it is configured and started it will intercept all changes done to source files and will stream them to HDFS without need of scheduling the transfer as a job, as in the case of Sqoop for example. Flume works as a background process.

Flume can be used as a data integration solution for all systems that are not loading data directly to a RDBMS but that are initially producing files in a text format: logs, CSV, etc.

4.3. GoldenGate for real time data replication to HDFS

Sqoop is a good solution for a first stage of the data integration process where initial load of the whole data set is performed. Then depending on the system requirements periodical incremental loading can be used in order to keep the Hadoop copy up to date. Alternatively a continuous data synchronization mechanism can be used instead. This approach is challenging because the information about data changes stored in transaction logs are not easy to retrieve from a RDBMS in a real time. However there are tools like Oracle GoldenGate [26] that are able to perform log mining and data change propagation in near real-time. Given the fact that HDFS is just a file system where data can be stored using different file formats and data organization, there is no a dedicated replicator solution for applying data changes and it has to be implemented by a user on his own.

Unlike Sqoop, this solution apply each transaction in almost real-time to text files and that files are read as well in almost real-time and copied to HDFS. The fact that it works in real-time comes with a problem with the creation of Parquet files, since this kind of files must be created from a large set of rows to get advantage of the features of Parquet files. In addition, Oracle GoldenGate cannot be adapted or modified to create other kind of files different that text files, so that files must be transformed. The transformations (casting and partitioning) can be applied by Hive or Impala engines when the data is being imported from the staging table to the final table. This approach has been tested at CERN and worked with data flows at the speed of 25 thousands row changes per second.

5. Conclusions

The Hadoop platform and its ecosystem of components are a proven solution for data warehousing and in particular they provide scalable systems with high throughput that is needed for storing and processing large amounts of data coming from controls systems at CERN.

Cloudera Impala is a storage engine on top of HDFS that has demonstrated very good scalability in our tests. Spark is another engine that can be used on top of Hadoop and that has shown very good scalability for our use cases. Moreover Impala and Spark SQL allow to query data using the SQL interface, which is a natural fit for many of the data warehouse use cases

From our tests we find that the choice of the storage engine is very important for the overall performance of the system. In particular Parquet and Avro data formats can be recommended for their data compression capabilities. Partitioning is also a key for query performance. For example time-based partitioning can provide substantial query speedup for time series data. Vertical partitioning, implemented for example by Parquet's columnar storage, can also provide very important improvements in performance. Finally data movement is an important topic for all data warehouse projects. Sqoop has proven to be a very good tool for moving data for bulk ingestion from database sources. Oracle GoldenGate can be used for near real-time ingestion. Flume is also a common solution for data streaming.

6. Future work

The project will move in Q3 2015 to the deployment into a larger cluster with more modern hardware. This will allow to load the full data set for the accelerator log and for controls data. Further development will be done towards a production service, in particular to ensure that the needed quality of service, performance and availability can be provided. Another important area of work will be to keep up with the constant and fast pace of evolution of the technology while maintaining the required service levels. Additional work on tuning the performance of near real-time replication solutions is also foreseen. The authors believe the technology and solutions showcased by this work open the path for many further explorations of data warehousing and data analysis on the Hadoop ecosystem that can be useful for many other areas of data management at CERN and for HEP in general.

Acknowledgements
The authors would like to warmly thank many people that were involved and contributed to the project, especially the users community:
C. Roderick, P. Sowinski, J. Wozniak from the Beams department of CERN and M. Berges, P. Golonka, A. Voitier from the Engineering department of CERN;
D. Duellmanm and R. Toebbicke from the CERN IT Storage group (IT-DSS);
all colleagues from the CERN IT Database group (IT-DB) including: E. Grancher, M. Limper , G. Tenaglia, M. Marquez and A. Marin

References

[1] The Large Hadron Collider: http://home.web.cern.ch/topics/large-hadron-collider
[2] Roderic C, Billen R, Gaspar Aparicio R C, Grancher E, Khodabandeh A, Seguera Chinchilla N, 2009, *The LHC Logging Service : Handling terabytes of on-line data* (CERN-ATS-2009-099)
[3] PVSS: http://www.pvss.com/Fachartikel/PI_Control_Engineering_Jan2010.pdf
[4] R. Denz, K. Dahlerup-Petersen, F. Formenti, K. H. Meß, A. Siemko, J. Steckert, L. Walckiers, J. Strait, 2009, *Upgrade of the protection system for superconducting circuits in the LHC* (CERN-A TS-2009-008)
[5] Index-Organized tables in Oracle RDBMS: http://docs.oracle.com/cd/E25054_01/server.1111/e25789/indexiot.htm
[6] Partitioning concept in Oracle RDBMS: http://docs.oracle.com/cd/E11882_01/server.112/e25523/partition.htm
[7] Oracle Real Application Clusters (RAC): http://docs.oracle.com/cd/B28359_01/rac.111/b28254/admcon.htm#i1058057
[8] Hadoop: http://hadoop.apache.org
[9] J. Dean and S. Ghemawat, 2004, *MapReduce: Simplified Data Processing on Large Clusters* (OSDI'04: Sixth Symposium on Operating System Design and Implementation)
[10] Apache Spark: https://spark.apache.org
[11] Ashish Thusoo et al, 2009, *Hive: a warehousing solution over a map-reduce framework* (VLDB Endowment, Volume 2 Issue 2 Pages 1626-1629)
[12] Cloudera Impala: http://impala.io
[13] Apache Spark SQL: https://spark.apache.org/sql/
[14] Apache Tez: http://tez.apache.org
[15] SequenceFile: http://wiki.apache.org/hadoop/SequenceFile
[16] RCFiles: https://hive.apache.org/javadocs/r0.12.0/api/org/apache/hadoop/hive/ql/io/RCFile.html
[17] ORC files: http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-.0.0.2/ds_Hive/orcfile.html
[18] Avro files: http://avro.apache.org/docs/1.3.0/
[19] Parquet files: http://parquet.apache.org
[20] Baranowski Z, Canali L, Grancher E, 2013, *Sequential data access with Oracle and Hadoop: a performance comparison* (*Journal of Physics: Conference Series*, 513(4):042001)
[21] bzip2: http://www.bzip.org
[22] Snappy: http://code.google.com/p/snappy
[23] Limper M , 2014, *An SQL-based approach to Physics anaysis* (Journal of Physics: Conference Series **513** (2014) 022022)
[24] Apache Sqoop: http://sqoop.apache.org
[25] Apache Flume: https://flume.apache.org
[26] Oracle GoldenGate: http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html