A Big Data Implementation on Grid Computing

Mrs. Swetha U
2nd sem M.Tech
SJBIT, Bangalore
swetha2087@gmail.com

Mrs. Prakruthi M K
Asst Prof, Dept of CSE
SJBIT, Bangalore
prakruthi.1012@gmail.com

Abstract: The huge volume and variety of data produced with great velocity is termed as BIG DATA. Thus, Big Data is characterized into three Vs. To handle such a Big data, a java-based software Hadoop an open-source data processing framework was developed which consists of Hadoop kernel, distributed File System, named Hadoop distributed file system(HDFS), fault-tolerant and scalable distributed data processing model and execution environment, named MapReduce and several related instruments. The main problem occurred when studying about Big data is storage capacity and processing power. That is the area where using Grid Technologies can provide help. Grid Computing refers to a special kind of distributed computing. Data storage and data access represent the key of CPU-intensive and data-intensive high performance Grid computing. The WLCG is now the world's largest computing grid. The Worldwide LHC (Large Hadron Collider) Computing Grid – WLCG, created in order to save, distribute and analyze the data generated in the LHC experiments. The main purpose of this article is to present a way of processing Big Data using Grid Technologies.

Keywords- Big Data, Hadoop, HDFS, Grid Technologies, WLCG.

I. INTRODUCTION

Beginning in the early 2000s, Google faced a serious challenge. Its mission — to organize the world's information — meant that it was crawling, copying, and indexing the entire Internet continuously. As the number and size of websites grew and Google's service became more popular, the company was forced to digest an ever-increasing corpus more quickly. No commercially available software could handle the volume of data to be processed, and Google's early, custom-built infrastructure was reaching the limits of its ability to scale. Thus, data that exceeds the processing capacity of conventional database systems is BIG DATA [20].

The hot IT buzzword of 2012, Big data generated from online transactions, emails, videos, audios, images, click streams, logs, posts, search queries, health records, social networking interactions, science data, sensors and mobile phones and their applications. 5 exabytes (10^{18} bytes) of data were created by human until 2003. Today this amount of information is created in two days. In 2012, digital world of data was expanded to 2.72 zettabytes(10^{21} bytes). It is predicted to double every two years, reaching about 8 zettabytes of data by 2015. IBM indicates that every day 2.5 exabytes of data created also 90% of the data produced in last two years. A personal computer holds about 500 gigabytes (10^9 bytes), so it would require about 20 billion PCs to store all of the world's data. In the past, human genome decryption process takes approximately 10 years, now not more than a week. Multimedia data have big weight on internet backbone traffic and is expected to increase 70% by 2013. Only Google has got more than one million servers around the worlds. There have been 6 billion mobile subscriptions in the world and every day 10 billion text messages are sent. By the year 2020, 50 billion devices will be connected to networks and the internet. [11]

All this volume represents a great amount of data that rise challenges when talking about acquiring, organizing and analyzing it. Big Data is an umbrella term describing all these types of information mentioned above. As the name suggests, Big Data refers to a great volume of data, but this is not enough to describe the meaning of the concept. The data presents a great variety, it is usually unsuitable for typical relational databases treatment, being raw, semistructured or unstructured. Also, the data will be processed in different ways, depending on the analysis that needs to be done or on the information that must be found in the initial data. Usually, this big amount of data is produced with great velocity and must be captured and processed quickly (as in the case of real time monitoring). Often, the meaningful and useful information comprised represents a small percent of the initial big volume of data – this means that the data has a low value density [1].

II. BACKGROUND

A. HADOOP

This type of data is impossible to handle using traditional relational database management systems. New innovative technologies were needed and in 2004, Google published an academic paper1 describing its work. Shortly thereafter, a well-known open source software developer named Doug Cutting decided to use the technique it described. Cutting was working on a web crawler called Nutch and was having the same problems with data volumes and indexing speed that had driven Google to develop MapReduce. He replaced the data collection and processing infrastructure behind the crawler, basing his new implementation on MapReduce. He named the new software Hadoop, after a toy stuffed elephant that belonged to his young son.

Hadoop is an open source project and operates under the auspices of the Apache Software Foundation. Hadoop consists of two major components: a file store and a distributed processing system. The file store is called the

Hadoop Distributed File System, or HDFS. HDFS provides scalable, fault-tolerant storage at low cost. The HDFS software detects and compensates for hardware issues, including disk problems and server failure. The second major component of Hadoop is the parallel data processing system called MapReduce. Conceptually, MapReduce is simple. MapReduce includes a software component called the job scheduler. The job scheduler is responsible for choosing the servers that will run each user job, and for scheduling execution of multiple user jobs on a shared cluster.

B. HDFS

At the foundation of Hadoop lies HDFS. The need for it comes from the fact that Big Data is, of course, stored on many machines. HDFS is a block-structured distributed file system designed to hold big amounts of data, in a reliable, scalable and easy to operate way.A Hadoop cluster uses Hadoop Distributed File System (HDFS) to manage its data. HDFS provides storage for the MapReduce job's input and output data. It is designed as a highly fault tolerant, high throughput, and high capacity distributed file system. It is suitable for storing terabytes or petabytes of data on clusters and has flexible hardware requirements, which are typically comprised of commodity hardware like personal computers. The significant differences between HDFS and other distributed file systems are: HDFS's write-once-read-many and streaming access models that make HDFS efficient in distributing and processing data, reliably storing large amounts of data, and robustly incorporating heterogeneous hardware and operating system environments. It divides each file into small fixed-size blocks (e.g., 64 MB) and stores multiple (default is three) copies of each block on cluster node disks. The distribution of data blocks increases throughput andfault tolerance. HDFS follows the master/slave architecture.

The master node is called the Namenode which manages the file system namespace and regulates client accesses to the data. There are a number of worker nodes, called Datanodes, which store actual data in units of blocks. The Namenode maintains a mapping table which maps data blocks to Datanodes in order to process write and read requests from HDFS clients. It is also in charge of file system namespace operations such as closing, renaming, and opening files and directories. HDFS allows a secondary Namenode to periodically save a copy of the metadata stored on the Namenode in case of Namenode failure. The Datanode stores the data blocks in its local disk and executes instructions like data replacement, creation, deletion, and replication from the Namenode. (adopted from Apache Hadoop Project[]) illustrates the HDFS architecture.

A Datanode periodically reports its status through a heartbeat message and asks the Namenode for instructions. Every Datanode listens to the network so that other Datanodes and users can request read and write operations. The heartbeat can also help the Namenode to detect connectivity with its Datanode. If the Namenode does not receive a heartbeat from a Datanode in the configured period of time, it marks the node down. Data blocks stored on this node will be considered lost and the Namenode will automatically replicate those blocks of this lost node onto some other datanodes.

C. MAPREDUCE

Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run in a distributed fashion on a cluster of computers. Each task works on the small subset of the data it has been assigned so that the load is spread across the cluster. The map tasks generally load, parse, transform, and filter data. Each reduce task is responsible for handling a subset of the map task output. Intermediate data is then copied from mapper tasks by the reducer tasks in order to group and aggregate the data. It is incredible what a wide range of problems can be solved with such a straightforward paradigm, from simple numerical aggregations to complex join operations and Cartesian products.

The input to a MapReduce job is a set of files in the data store that are spread out over the Hadoop Distributed File System (HDFS). In Hadoop, these files are split with an input format, which defines how to separate a file into input splits. An input split is a byte oriented view of a chunk of the file to be loaded by a map task.

Each map task in Hadoop is broken into the following phases: record reader, mapper, combiner, and partitioner. The output of the map tasks, called the intermediate keys and values, are sent to the reducers. The reduce tasks are broken into the following phases: shuffle, sort, reducer, and output format. The nodes in which the map tasks run are optimally on the nodes in which the data rests. The output format translates the final key/value pair from the reduce function and writes it out to a file by a record writer. By default, it will separate the key and value with a tab and separate records with a newline character. This can typically be customized to provide richer output formats, but in the end, the data is written out to HDFS, regardless of format. This way, the data typically does not have to move over the network and can be computed on the local machine.

D. GRID COMPUTING

Two of the main problems that occur when studying Big Data are the storage capacity and the processing power. That is the area where using Grid Technologies can provide help. Grid Computing refers to a special kind of distributed computing. A Grid computing system must contain a Computing Element (CE), and a number of Storage Elements (SE) and Worker Nodes (WN). The CE provides the connection with other GRID networks and uses a Workload Management System to dispatch jobs on the Worker Nodes. The Storage Element (SE) is in charge with the storage of the input and the output of the data needed for the job execution.

The Large Hadron Collider(LHC)[13] in Geneva, Switzerland is not only a unique experiment for investigations in particle physics, but also presents an exceptional computing challenge for those tasked with recording and processing huge amounts of data. When originally tasked with the problem, the four detector experiments on the LHC decided to take a decentralized processing approach through using grid computing. The grid formed to handle this task is the Worldwide LHC Computing Grid (WLCG), one of the largest computing grid.

A grid center is composed of computing elements (CE), storage elements (SE) and worker nodes (WN). Basically, the CE manages the resources of the Grid node and manages the jobs launched. The SE offers the storage and data transfer services and the WNs are the servers that offer the processing power. There is also a User Interface (UI). At a higher level, in order to access the Grid resources, a potential user must have a certificate issued by a Virtual Organization (VO) – a set of institutions or people that define rules of accessing and using
the resources in a grid center. There is the administrative part of a Virtual Organization which comprises a Workload Management System that keeps track of the available CEs for the users jobs, a Virtual Organization Membership System (VOMS) and the Logical File Catalog (LFC). These parts can be shared by more Grid centers. A user accesses the UI via SSH (Secure Shell) and he receives a Proxy Certificate (PyC). The user then sends the job written in Job Description Language (JDL) and the PyC to the WMS. The WMS checks the PyC and if the needed resources for the job are available. It, then, sends the job and the PyC to the CE. The CE checks the authenticity of the user again and then sends the job to be processed by a WN. The WN computes the job and then sends the results to the WMS and the state of the job to the CE. The users gather the results using the UI and he can store them on the SE[1].

The Compact Muon Solenoid (CMS)[10] experiment at the LHC, each recognized Grid site, ranging from Tier-0 to Tier-3 in WLCG[9], needs to run a standalone SE to fulfill some common tasks according to CMS Virtual Organisation's (VO) computing needs: moving data between sites and allowing Grid jobs to access data at the local storage. Data storage and access represent the key of CPU-intensive and data-intensive high performance Grid computing. However, the small/medium size Grid sites are in particular constrained to use often commodity hardware which exposes them to hardware failure.

Building a Storage Element (SE)[12] based on Hodoop is a natural choice for CPU-intensive and data intensive Grid computing since Hadoop has many features well fit into the distributed computing architecture: high scalability, reliability, throughput, portability, capability of running on heterogeneous platforms, low maintenance cost, etc. But the integration between Hadoop and existing Grid software and computing models is non-trivial. In addition to the general requirements on the SE, a Virtual Organization (VO) may implement special requirements depending on its data operation mechanism and data access pattern. To accomplish these tasks, some technical specification and preference for the SE technology include but not limited to:

- SE technology must have a credible support model that meets the reliability, availability, and security expectations consistent with the computing infrastructure.
- SE technology must demonstrate the ability to interface with the existing global data transfer system and the transfer technology of SRM tools and FTS as well as demonstrate the ability to interface to the CMS software locally through ROOT.
- The SE must have well-defined and reliable behavior for recovery from the failure of any hardware components. This behavior should be tested and documented.
- The SE must have a well-defined and reliable method of replicating files to protect against the loss of any individual hardware system.
- The SE must have a well-defined and reliable procedure for decommissioning hardware which is being removed from the cluster; this procedure should ensure that no files are lost when the decommissioned hardware is removed.
- The SE must have well-defined and reliable procedure for site operators to regularly check the integrity of all files in the SE.
- The SE must have well-defined interfaces to monitoring systems so that site operators can be notified if there are any hardware or software failures.
- The SE must be capable of delivering at least 1 MB per second per batch slot for CMS applications. The SE must be capable of writing files from the wide area network at a performance of at least 125 MB/s while simultaneously writing data from the local farm at an average rate of 20 MB/s.
- The new SE should be filled to 90% with CMS data. Failures of jobs due to failure to open the file or deliver the data products from the storage systems should be at the level of less than 1 in 10^5 levels.

III. IMPLEMENT A BIG DATA PLATFORM IN A GRID CENTER

Hadoop is written mainly for data transfer within the same datacenter whilst grid computing is mainly developed for distributing the data and the computational power between different sites possibly in different geographical areas [10].

A. HDFS-BASED STORAGE ELEMENT ARCHITECTURE IN OSG[15]

In order to build a SE which provides I/O and other Grid data services to the VO(s), we consider following software are needed to be seamlessly integrated with HDFS to fulfill all the basic required functionalities of the SE.

- FUSE [16]: It is a linux kernel module, allows file systems to be written in userspace and provides POSIX like interface to HDFS. With FUSE mount, the whole file system of the SE will be "local" to the WN, which is crucial for user applications running at the WN to access data at the local storage.

- Globus GridFTP server [17]: It provides WAN transfer between SE(s) or SE and workernode (WN). Due to the fact that HDFS only supports synchronous write, we developed a special plugin to GridFTP. It is able to assemble asynchronous transferred packets to be sequentially written to HDFS, which is necessary for GridFTP running at multiple-stream-transfer mode to achieve optimal performance in file transfer.

- BeStMan server [18]: It provide SRMv2 interface to HDFS. Several plugins are developed to enable selection of GridFTP servers.

There are optional components that may be layered on top of HDFS depending on specific needs of a site or VO's computing strategy, for example XrootD, Apache HTTP server and Fast Data Transfer (FDT)[19].

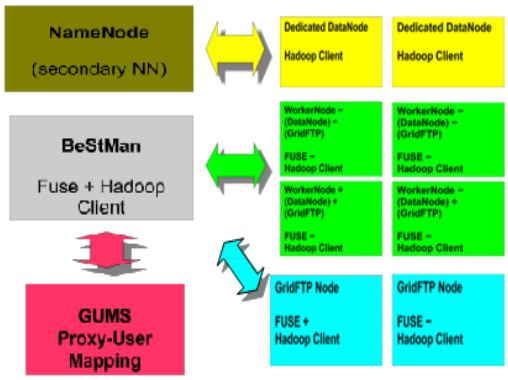

Fig.1. Architecture of HDFS-based Storage Element

The adoption of a generic integrated architecture of HDFS-based SE, as shown in Fig.1., that includes multiple level of data access and several possible ways of sharing those services on limited pieces of hardware, which is shown efficient, scalable and extensible:

- Data access in the local domain: Direct data access via FUSE mount is the primary method to access the data at local domain, since a POSIX compliant file system is necessary for many applications to access data at the SE. The support of this method largely depends on the configuration of the site, since it is not within the GSI authentication and authorization. The export of HDFS to the FUSE mount of a machine is subject to the security consideration of the site. Direct data access via Hadoop client. This method is similar to direct access via FUSE mount, except that it can only be used for file transfer. In the case of data access via FUSE, a site must explicitly authorize the export of the HDFS to the machine to be mounted, while in this case a more open and general access can be offered to a pool of machines for file transfer purpose. The BeStMan server provides a standard data provision and access method via SRMv2 protocol. It works as a frontend of transfer servers and on top of POSIX compliant file system. The GridFTP server also provides a standard data provision and access method. A site can run multiple GridFTP servers to maximize the throughput of the file transfer with load-balance.

- Data access between two SEs: SRM and GridFTP are the standard data transfer mechanism between SEs. Direct data access via FUSE mount between two SEs or a secure remote machine and SE is technically feasible, although it provides the easiest and simplest access to data including file transfer and running applications against data, the data access is limited by the throughput of single FUSE mount, which is more appropriate for user applications interactively accessing data at the SE locally. Direct data access via Hadoop client can also be supported between SEs, but it is also subject to security consideration.

- Data and system security: Generally strong X.509 authentication and authorization are needed for the data access to the SE if creation and deletion of files are involved and initiated from a public host. These operations can be allowed via local FUSE mounted machines, since these machines are under the administration of local domain. For security concern, the remote direct file operation via FUSE or Hadoop client should be prohibited. A GSI-compliant mechanism like using GUMS server can be implemented to map the user X.509 SSL certificates to local user or group accounts which are used by HDFS. According to the roadmap of HDFS, access token, users authentication and data encryption will be supported that provide strong security for HDFS at the local domain in the future.

- Sharing services in the cluster: The computing cluster consists of a number of WorkerNodes (WN). HDFS as a distributed storage solution can make best use of those WNs and form a pool of DataNodes, which lead to the sharing between WN and DataNode. The sharing between GridFTP and

computing nodes (WN and/or DataNode of HDFS) is also feasible. The capability of HDFS supporting high throughout in inter-DataNode file replication and distribution allows a wide choice of configuration for the cluster and won't put physical limit or bottleneck on the service if the service is shared by a sufficient number of distributed hardware.

From computing architecture point of view, the model of sharing data services with computing nodes is an economical choice for sites running commodity hardware to achieve high end performance if the overall throughput is the major concern.

An extra advantage of sharing service lies on the fact that HDFS distributes data to the whole cluster. Applications accessing data will naturally interact with a pool of DateNodes. The performance is mainly determined by the scalability of the SE and load balance among many pieces of hardware mediated by the computing model. For a Grid SE solution, the distributed and shared model is the most flexible and resilient one for serving a large group of users and various applications with their data access patterns.

A set of services implemented by SE, from data transfer to data access, involve various components of the storage system and grid middleware as shown in Fig.2.

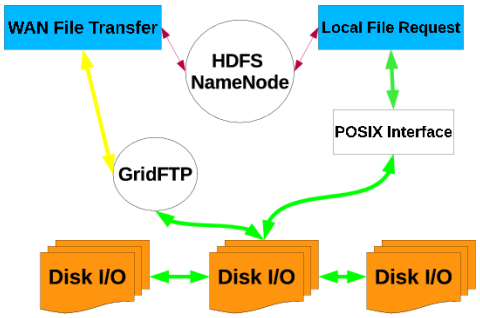

Fig.2. File transfer and data access between various components of the HDFS based SE

An efficient way for jobs to access data within the site's storage system is shown in Fig.3:

- For local access, HDFS has a binding for Linux called FUSE. FUSE allows one to write a mountable file system in userspace (greatly simplifying the development of a file system and removing the need for custom kernels). Hence, worker nodes can mount HDFS and users access it in a similar manner to a NFS server. This transforms HDFS into more "normal Looking" file system for users, as opposed to a more specialized data storage system. In the process of testing and deploying FUSE at a large scale, we have contributed back several bug reports and improvements. To our knowledge, we are the largest and heaviest user of FUSE, as none of the physics applications use the native Hadoop API.

- For SRM access, we have utilized the Berkeley Storage Manager (BeStMan) SRM server. BeStMan's gateway mode implements all of the SRM calls utilized by CMS and ATLAS (but not all the SRM protocol functionalities). This stripped-down implementation has an advantage in that it can be used on any POSIX-like file system; other implementations of the SRM protocol are often tied to a specific storage system. The FUSE mounts which provide data access to jobs on worker nodes also provide metadata access to the SRM server. Hence, we were able to use BeStMan to provide SRM access to HDFS without any initial modifications.

- For GridFTP, we selected the Globus GridFTP server. The Globus server is in wide use elsewhere, and allows for pluggable modules to be implemented interacting with a storage system. Unlike SRM, GridFTP actually writes data in and out of HDFS. Because an HDFS only support appends, and not random writes, we had to implement in-memory data re-ordering and write an interface module between GridFTP and HDFS's C bindings.

B. HDFS-BASED STORAGE ELEMENT ARCHITECTURE IN gLITE

On the WLCG, the protocol used for data transfers between sites is GridFTP and the protocol used for metadata operations is SRM; both are necessary for interoperability between site storages in the Grid. To deploy Hadoop as Grid Storage Element (SE) in Open Science Grid (OSG) sites, a plugin has been developed to allow the interaction between GridFTP server and HDFS storage system, and Berkeley Storage Manager (BeStMan) SRM server has been utilized to allow SRM access.

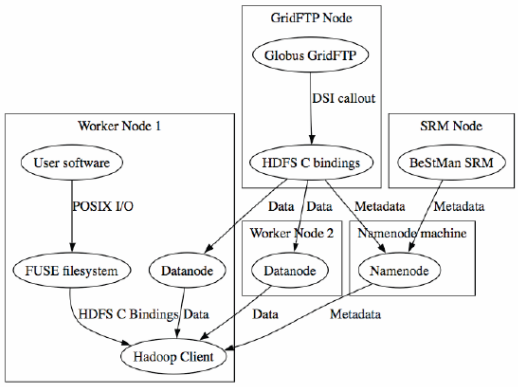

Fig.3 HDFS-BASED SE

For the deployment of Hadoop as SE in gLite, the first choice was to use StoRM as Grid SRM since it is widely deployed in this environment. But investigating this solution, it was noticed that Storm cannot work with File

System not supporting Access Control List (ACL). Since HDFS does not support such control, the BeStMan SRM server was used.

The following steps were required to setup HDFS as a Grid SE in gLite[14] environment:

- setup GridFTP server: HDFS-GridFTP library developed for OSG sites is recompiled in gLite environment and then gLite GridFTP server is started using this library as Data Storage Interface (dsi): globus-gridftp-server -p 2811 -dsi hdfs ;
- setup SRM server: HDFS is mounted using Fuse. Next, BeStMan is installed and configured correctly to be able to authenticate users using Gridmap-file and to manage the experiments storage area in HDFS;
- setup Xrootd service: As interface to HDFS for ROOT files access to allow read and write into HDFS using ROOT software program. The xrootd-hdfs rpm used in OSG site is installed with -nodeps option to bypass the credential check required for OSG sites;
- publish SE Information to Site Berkeley Database Information Index (BDII) to make them available in the Grid.
- A provider script is developed to publish dynamic information to gLite Information service.
- SRM-PING is called to get required information.

The Hadoop-based SE architecture in gLite environment and the interactions between its components are shown in Fig.4.

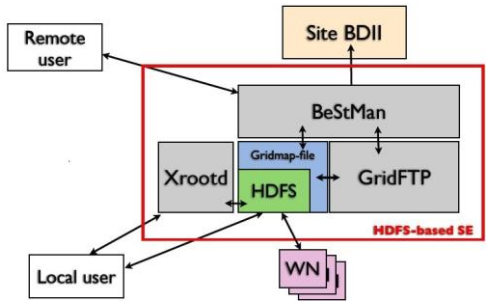

Fig.4. Architecture of Hadoop-based SE in gLite environment and the interactions of the SE with others gLite Grid components. The services involved in the typical interactive accesses by users to SE.

IV. PERFORMANCE OF THE PRESENTLY DEPLOYED HDFS-BASED SES

- It stably delivers 10MB/s to applications in the cluster while the cluster is fully loaded with data processing jobs. This exceeds CMS application's requirement on I/O by an order of magnitude.
- HDFS NameNode serves 2500 request per second as shown in Fig.3.5.3.1, which is sufficient for a cluster of thousands of cores to run I/O intensive jobs.
- It achieves sustained WAN transfer rate at 1,034MB/s, which is sufficient for the data operation of CMS Tier-2 centers including transferring datasets and staging out of user jobs. Fig.5 shows the network throughput of data transfer as a typical data operation with the new SE.
- It can simultaneously process several thousand client's requests at the BeStMan server and achieve sustained endpoint processing rate above 50Hz as shown in Fig.6, which is sufficient for high-rate transfer of gigabytes sized files and uncontrolled I/O requested by random user jobs from across the CMS VO running on more than 30 thousand CPUs mainly provided by over 60 Tier-1 and Tier-2 sites.
- Extremely low file corruption rate is observed, which mainly benefits from robust and fast file replication of HDFS.
- Decommissioning of a DataNode uses less than 1 hour, restarting NameNode in 1 minute, checking the image of file system from memory of NameNode in 10 sec, which are fast and efficient for the operation and system administration.
- In general, we expect that many of these performance characteristics scales with the amount of hardware provided.

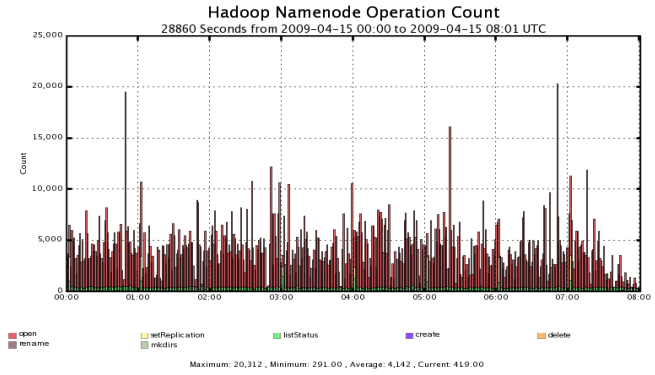

FIG.5. The operation count at the NameNode of HDFS for a period of 8 hours. The operation of NameNode mainly involves file opening, replication, creation, deletion, renaming and etc. File opening (an indication of Grid jobs accessing data at the SE) dominates the operation.

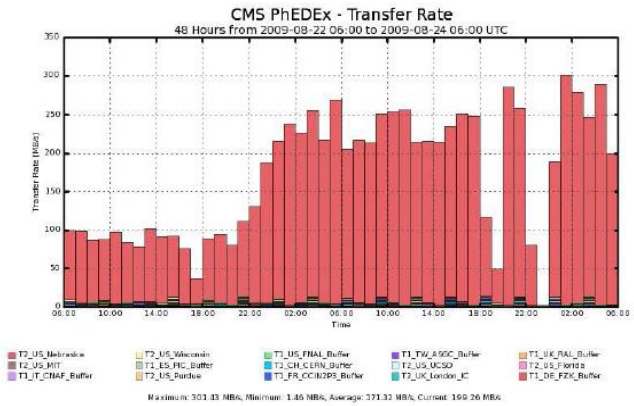

FIG.6. CMS WAN data transfer managed by PhEDEx

V. FAULT TOLERANCE AND SCALABILITY TESTS OF HDFS FOR SMALL/MEDIUM SIZE GRID SITE

One of the main interesting features of Hadoop is the capability to replicate data in order to avoid any data loss both in case of hardware or software problems. The data replication is automatically managed from the system and it is easy to configure the number of the replicas that the system should guarantee. This is important because, the most critical data should be guaranteed with a sufficiently high number of replicas. All the files hosted in the HDFS File System are split in chunk (the size of the chunk could be configured from the site admin), and each file could be registered in a different disk or host.

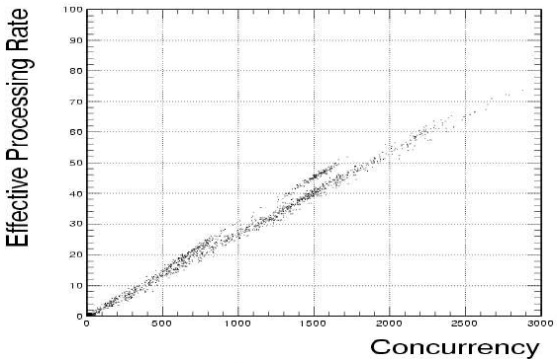

FIG.7. Correlation between Effective Processing Rate and Client Concurrency for small directories

The testing activities was focused on how the replication mechanism works and if this could really be a solution for medium-sized computing centers that have to provide an high available data access infrastructure, without the need of buying an expensive and complex hardware solution.

Several different kinds of failures were simulated in order to check how HDFS reacts to each kind of problem. In particular, the two main topics were metadata failures and data failures. Both issues may happen for different reason, so we have tested: temporary network failure, complete disk failures, host failures, and File System corruption.

Starting from metadata failure: from the design it is clear that the NameNode (that is the metadata server) is the single point of failures in the HDFS file-system. A failure on the metadata, could lead to unavailability of files. Looking at the design of the system it is possible to have a Secondary NameNode that could sync the metadata from the primary NameNode on a different disk.

The Secondary NameNode, indeed, will dump the status of the primary every given amount of seconds that could be configured from the site admin. It has been measured that the dump takes less than one second with hundreds of files to be synchronized. It was tested that if the NameNode gets corrupted or its file-system is deleted, it is easy to import the metadata from the backup on the Secondary NameNode. While the NameNode is not available, every activity is frozen both on the client side as on the DataNode side, but as soon as the NameNode is restarted, all the processes reconnect and the activities proceed.

This is achieved because the HDFS client can be configured to retry every operation in case of failures. In particular, using configuration parameters, it is possible to specify the number of times that the client should retry a write operation before really failing. This is very useful as this provides fault-tolerance also in case of DataNodes failures. Indeed, a client that is writing a file on a specific DataNode, could, in case of failure of that node, retries the write operation on another DataNode as many times as it is configured from the site admin, until a working DataNode will be found and the transfer is completed successfully. For what about the read operation, it has been tested that if a DataNode fails during a read operation, the client gets only a warning and the operation can continue using a node that hosts the replica. This failure automatically triggers a copy of the chunk hosted in the failed node, in order to guarantee that the number of available replicas is always the same. Fig.8. High availability of the DataNode mechanism when reading corrupted data

It has been also tested what happens in case of data corruption. It has been intentionally corrupted a chunk of a file and try to read back that file. The HDFS system automatically understands that there is a problem on that chunk, flags it as failed, reads it from another replica and deletes the corrupted chunk so that it is automatically replicated. This mechanism is shown schematically in Fig.8.

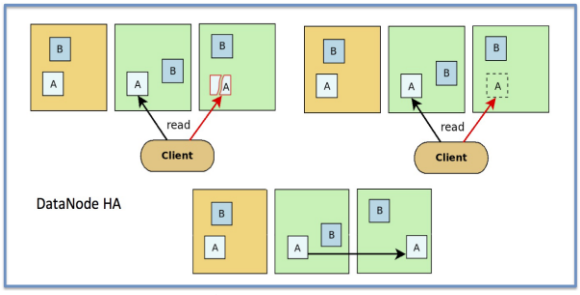

FIG.8. High availability of the datanode mechanism when reading corrupted data

All these features are able to reduce the effort needed to maintain the system in production and this is very important as it means a low Total Cost of Ownership. The INFN-Bari group is also involved in testing the scalability of the HDFS File System, testing a solution with up to 130 different DataNodes. In this test the idea is to prove that a solution based on HDFS could be easily scaled up in terms of datanode in the cluster, by means of simply adding new nodes to it. This was typically called "horizontal scalability" in opposite to the "vertical scalability" that requires that a single, or few nodes are upgraded in order to obtain grater performance from each of those.

From the test it is evident that a system based on HDFS is highly scalable and we can easily get up to two Gbyte/s of aggregate bandwidth among the nodes.

VI. ROOT FILES ANALYSIS WITH HADOOP MAPREDUCE

To take advantage of the parallel processing that Hadoop provides, it is needed to express the ROOT files analysis data ow as a MapReduce job. MapReduce works by breaking the processing into two phases: the “map” phase and the “reduce” phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function. Here only the “map" function is considered to express the common required step to analyze ROOT files, namely read and write of the files.

The input to the map phase is usually one or more text files. The default input format is Text input format. It gives each line in the text files given in input as a text value. The output is the result of the map execution over the input files formatted as key-value pairs. For ROOT files analysis, the input is one or more ROOT files. The Analyzer is usually a C++ analysis code that takes as input the ROOT files names. It opens them for processing by the mean of ROOT primitives call and then produces output ROOT files.

MapReduce and ROOT files Analysis data flows are illustrated in Fig.9.

The implementation of the ROOT files analysis data flow as a MapReduce job requires the support of ROOT files as input to the “map” function. To achieve that, it was required to develop a dedicated input format, named RootFilesInputFormat, inheriting from the super-class FileInputFormat.

FileInputFormat is the base class for all implementations of InputFormat that use files as their data source. It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files. The job of dividing splits into records is performed by the subclasses. The FileInputFormat class calls the RecordReader class to effectively open and read the files contents from the File System.

The RootFilesInputFormat allows the splitting of the input per file instead of line per file. The input files will be then opened and processed inside the Analyzer, implemented as “map” function, producing as result a ROOT file. Within RootFilesInputFormat, the input files contents will not be read and passed as input stream to the “map” function, as it is done for Text files, but only the input split path will be passed to the map function. The “map.input.file” environment variable is set to the input split path by the RootFilesInputFormat object and then read by the “map” function.

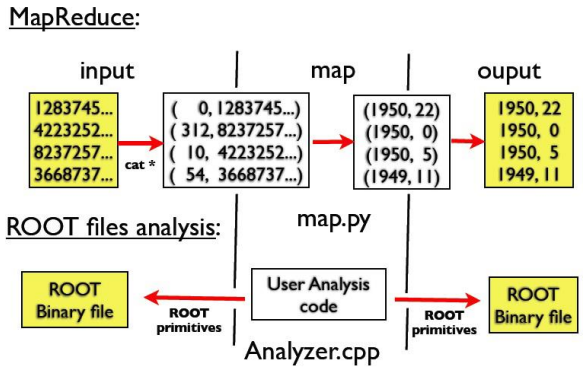

FIG.9. Typical ROOT FILES ANALYSIS and MAPREDUCE data flows: in MAPREDUCE the content of the input files are passed to the “map” function which analyzes it and then produces the output. For the root files analysis, the analyzer gets as input the files names, open it for analysis, and then produces the output.

VII. CONCLUSION

The main advantages offered by Grid computing are the storage capabilities and the processing power and the main advantages of using Hadoop, especially HDFS, are reliability (offered by replicating all data on multiple DataNodes and other mechanism to protect from failure), the scheduler’s ability to collocate the jobs and the data offering high throughput for data for the jobs processed on the grid. Adding the ease of use, ease of maintenance and scalability combining these two technologies seems like a good choice.

Hadoop-based storage solution is established and proved functioning at CMS Tier-2 sites and within the OSG community.

HDFS has been deployed successfully in gLite environment and has shown satisfactory performance in term of scalability and fault tolerance while dealing with small/medium site environment constraints. ROOT files Analysis workflow has been deployed using Hadoop and has shown promising performance.

Hence, by implementing a Hadoop based SE, we take advantage of the WN’s storage capabilities, the Hadoop scheduler’s abilities to send jobs where the needed data is located.

REFERENCES

[1] Garlasu, D. ;Core Technol. Oracle Romania, Bucharest, Romania ; Sandulescu, V. ; Halcu, I. ; Neculoiu, G "A big data implementation based on Grid computing" Published in Roedunet International Conference (RoEduNet), 2013

[2] Garhan Attebury, Andrew Baranovski, Ken Bloom, Brian Bockelman, Doria Kcira, James Letts, Tanya Levshina, Carl Lundestedt, Terrence Martin, Will Maier, Haifeng Pi, Abhishek Rana, Igor Sfiligoi, Alexander Sim, Michael Thomas, Frank Wuerthwein, "Hadoop Distributed File System for the Grid", IEEE Nuclear Science Symposium Conference Record, Orlando, FL, 2009

[3] Brian Bockelman, "Using Hadoop as a Grid Storage Element", IOP Publishing, CSE Conference and Workshop Papers, 2009

[4] H Riahi, G Donvito, L Fano, M Fasi, G Marzulli, D Spiga and A Valentini "Using Hadoop File System and MapReduce in a small/medium Grid site", IOP Publishing, International Conference on Computing in High Energy and Nuclear Physics 2012

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. Commun.ACM, 51(1):107-113, 2008

[6] Yaodong Cheng, "HEP Grid and Hadoop", available at www.hadooper.cn, 2009

[7] Apache hadoop, 2009.
http://hadoop.apache.org/.

[8] Dhruba Borthakur. Hdfs architecture, 2009. http://hadoop.apache.org/core/docs/r0.20.0/hdfs design.html

[9] http://wlcg.web.cern.ch.

[10] The Compact Muon Solenoid Experiment. http://cms.web.cern.ch/cms/index.html.

[11] Collaboration Technologies and Systems (CTS), 2013 International Conference on "Big Data: A review" Sagiroglu, S. ; Dept. of Comput. Eng., Gazi Univ., Ankara, Turkey ; Sinanc, D Publication Year: 2013 , Page(s): 42 - 47 IEEE Conference Publications

[12] Storage Element on Open Science Grid. Available:https://twiki.grid.iu.edu/bin/view/Documentation/AboutStorageElements.

[13] The Large Hadron Collider.
Available: http://lhc.web.cern.ch/lhc.

[14] The gLite File Transfer Service.
Available:http://egee-jra1-dm.web.cern.ch/egeejra1-dm/FTS.

[15] Open Science Grid.
Available: http://www.opensciencegrid.org

[16] File System in Userspace.
Available: http://fuse.sourceforge.net.

[17] GridFTP Protocol.
Available:http://www.globus.org/gridsoftware/data/gridftp.php.

[18] Berkeley Storage Manager.
Available: https://sdm.lbl.gov/bestman.

[19] Fast Data Transfer.
Available: http://monalisa.cern.ch/FDT.

[20] Cloudera Project.
Available: http://www.cloudera.com.

Appendix:

The implementation details of RootFilesInputFormat class and RootFileRecordReader class is given below in Fig. A1 and A2 respectively.

```
package org.apache.hadoop.streaming;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.mapred.InputFormat;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobContext;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.TaskAttemptContext;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.JobConf;
import java.io.IOException;
import org.apache.hadoop.fs.FileSystem;

public class RootFileInputFormat
        extends FileInputFormat<NullWritable, BytesWritable> {
        protected boolean isSplitable(FileSystem fs, Path filename) {
                return false;
        }
```

Fig.A.1 IMPLEMENTATION OF RootFileInputFormat CLASS.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.TaskAttemptContext;
import org.apache.hadoop.mapred.FileSplit;
import java.io.IOException;
import org.apache.hadoop.io.IOUtils;

class RootFileRecordReader implements RecordReader<NullWritable, BytesWritable> {
...
public RootFileRecordReader(FileSplit fileSplit, Configuration conf) throws IOException {
...
public boolean next(NullWritable key, BytesWritable value) throws IOException {
        if (!processed) {
                byte[] contents = new byte[0];
                Path file = fileSplit.getPath();
                FSDataInputStream in = null;
                value.set(contents, 0, contents.length);
                conf.set("map.input.file", conf.get("mapred.input.dir") + "/" + file.getName());
                processed = true;
                return true;
                }
        return false;
}
...
```

Fig.A.2 IMPLEMENTATION OF RootFileRecordReader CLASS.